CSCI 4155: Machine Learning and Robotics

Thomas P. Trappenberg
Dalhousie University

Acknowledgements

These lecture notes are closely based on several sources, which I commend as further readings. In particular, much of the machine learning part is derived from the lecture nodes by Andrew Ng from Stanford University, specifically his lecture notes of 'CS221: Artificial Intelligence: Principles and Techniques' from the Winter 2009, and some notes from his course 'CS229: Machine Learning' from the Fall 2008. The full set of these lecture notes and video links to his lectures are available on his web site (`http://robotics.stanford.edu/∼ang`).

Andrew's material is augmented with some material from slides by Stuart Russel and Peter Norvig for their book *Artificial Intelligence: A Modern Approach*, 2nd edition, Prentice Hall, 2003. This book covers machine learning in the later chapters, and is also the standard textbook for AI more generally. One of the best books more specifically dedicated to machine learning is Christopher Bishop's, *Pattern Recognition and Machine Learning*, Springer 2006. For reinforcement learning I highly recommend the book by Richard Sutton and Andrew Barto, *Reinforcement Learning: An Introduction*, MIT press, 1998. A modern and well written introduction to robotics is given by S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*, MIT Press 2005.

Several people have contributed considerably to this lecture notes. In particular, Leah Brown and Ian Graven have created most of the original Lego Mindstorm implementations and tutorials, Patrick Connor made several contributions, and Chris Maxwell helped with some implementation issues.

Contents

Part I

Introduction and background

1 Machine Learning and the probabilistic framework

Both, machine learning and robotics have been important topics in the area of artificial intelligence. The introductory chapters in this first section outline some history and the direction of the course and review some material which are important as background in the first tutorials. This includes probability theory, programming with Matlab, and how to use the Lego MIndstorm.

1.1 Some history

Artificial Intelligence (AI) has many sub-discipline such as search, knowledge representation, expert systems, etc. This course will focus on **Machine Learning (ML)** where some aspects of an algorithm rely on learning from examples or feedback from the environment to find solutions to problems.

AI has long been tightly interwoven with ML. For example, Samuel's checkers program from the 1950s was able to learn from experience and thereby outperforming its creator. Basic Neural Networks, such as Bernhard Widrow's ADALINE (1959), Karl Steinbuch's Lernmatrix (around 1960), Rosenblatt's Perceptron (late 1960s), and Richard Bellman's Dynamic Programming (1953), have created a lot of excitement during the 1950s and 60s. This has been paralleled in the progress of understanding biological systems such s Donald Hebb's influential book *The Organization of Behavior* (1943) and Eduardo Caianiello's influential paper *Outline of a theory of thought-processes and thinking machines* (1961). However, after Marvin Minsky and Seymore Papert published their book *Perceptrons* in 1969 with a proof that simple perceptrons can not learn all problems, the field quickly subsided and shifted towards expert systems. Neural Networks became again popular in the mid 1980 after the backpropagation algorithms was popularized by David Rumelhart, Geoffrey Hinton and Ronald Williams (1986).

Since then, important progress has been made, in particular through the more rigorous mathematical formulations and the embedding of such methods with stochastic methods. In particular, important learning theories become widely known and developed further after Vladimir Vapnik published his book *The Nature of Statistical Learning Theory* in 1995. Bayesian methods have also transformed the field (Judea Pearl, 1985).

Fig. 1.1 Some AI pioneers AI.From top left to bottom right: Alan Turing, Arthur Lee Samuel, Marvin Minsky, Richard Bellman, Frank Rosenblatt, Geoffrey Hinton.

1.2 Stochastic modeling

In this course we discuss in some detail three different types of learning systems which are commonly categorized as supervised learning, unsupervised learning, and reinforcement learning. **Supervised learning** is characterized by given explicit examples that the system should learn (**memorize**) and from which the system should **generalize** to previously unseen examples. The supervision is usually given in the form of labels, $y^{(i)}$ for training patterns with features $\mathbf{x}^{(i)}$. This is usually contrasted with **unsupervised learning**, which aims to discover structure in data without explicit labels. **Reinforcement learning** is somewhat in between as the learning is guided only by some feedback on the quality of actions produced by the earning system rather than specifying what is right or wrong as can be done in supervised systems. We will also discuss hybrid systems.

Traditional AI has provides many useful approaches for solving specific problems. For example, search algorithms can be used to navigate mazes or to find scheduling solutions, and such strategies should first be considered, However, learning systems are usually more general and can be applied to situations for which closed solutions are not be known. Also, experience has shown that applications often fail at some point since systems change over time or when an agent encounters situations for which the system has not been designed. Learning systems are generally thought to be more robust. Indeed, a main objective of learning systems is to build systems that generalize well to new situation, that is, learning systems should be able to generate sensible solutions to previously unseen situations. We will discuss this point in detail in Chapter 5.

Another well known problem in building practical applications is that systems are generally **unreliable** and that environments are **uncertain**. For example, we a program might read data files in a specific format, but some users supplies a corrupted file. Productions software is often lengthy only to consider all kind of situations that could occur, and we now realize that considering all possibilities is often inpossible. This problem will become very apparent in robotics where sensors are often unreliable and estimation techniques are limited either because of limited resources or limited data. Even actions can be noisy.

While the area of machine learning has a long history as outlined above, there has been major advances made over the last two decades. Much of the progress was made possible by the realization of researchers in many related areas that the language of (Bayesian) probability theory is very useful in this area. This language is appropriate since is acknowledges the fundamental limitations we have in real world applications (such as limited computational researches or inaccurate sensors). The language of probability theory has certainly helped to unify much of related areas and improved communication between researchers.

Our aim of learning machines is to learn from the environment, either through instructions, by reward or punishment, or just by exploring the environment and learning about typical objects or temporal sequences. Our ultimate goal is to model the world and to use such models to make 'smart' decisions in order to maximize reward.

The next two chapters are outlining some of the tools we will use to explore and describe this area. Chapter 2 is a brief introduction to the Matlab programming environment that we will mainly use in the following, Chapter 3 shows you how to use the Lego NXT and how to control them with Matlab. This chapter also includes a discussion of the important concept of a configuration space and how to plan movements with a robot arm. Chapter 4is a refresher on basic probability theory. The rest of the course is then divided into supervised, unsupervised, and reinforcement learning.

2 Programming with Matlab

This chapter is a brief introduction to programming with the Matlab programming environment. We assumes thereby little programming experience, although programmers experienced in other programming languages might want to scan through this chapter. MATLAB is an interactive programming environment for scientific computing. This environment is very convenient for us for several reasons, including its interpreted execution mode, which allows fast and interactive program development, advanced graphic routines, which allow easy and versatile visualization of results, a large collection of predefined routines and algorithms, which makes it unnecessary to implement known algorithms from scratch, and the use of matrix notations, which allows a compact and efficient implementation of mathematical equations and machine learning algorithms. MATLAB stands for matrix laboratory, which emphasizes the fact that most operations are array or matrix oriented. Similar programming environments are provided by the open source systems called **Scilab** and **Octave**. The Octave system seems to emphasize syntactic compatibility with MATLAB, while Scilab is a fully fledged alternative to MATLAB with similar interactive tools. While the syntax and names of some routines in Scilab are sometimes slightly different, the distribution includes a converter for MATLAB programs. Also, the Matlab web page provides great videos to learn how to use Matlab at `http://www.mathworks.com/demos/matlab/...` `...getting-started-with-matlab-video-tutorial.html`.

2.1 The MATLAB programming environment

MATLAB[1] is a programming environment and collection of tools to write programs, execute them, and visualize results. MATLAB has to be installed on your computer to run the programs mentioned in the manuscript. It is commercially available for many computer systems, including Windows, Mac, and UNIX systems. The MATLAB web page includes a set of brief tutorial videos, also accessible from the **demos** link from the MATLAB desktop, which are highly recommended for learning MATLAB.

As already mentioned, there are several reasons why MATLAB is easy to use and appropriate for our programming need. MATLAB is an interpreted language, which means that commands can be executed directly by an interpreter program. This makes the time-consuming compilation steps of other programming languages redundant and allows a more interactive working mode. A disadvantage of this operational mode is that the programs could be less efficient compared to compiled programs. However, there are two possible solution to this problem in case efficiency become a concern. The first is that the implementations of many MATLAB functions is very efficient

[1]MATLAB and Simulink are registered trademarks, and MATLAB Compiler is a trademark of The MathWorks, Inc.

and are themselves pre-compiled. MATLAB functions, specifically when used on whole matrices, can therefore outperform less well-designed compiled code. It is thus recommended to use matrix notations instead of explicit component-wise operations whenever possible. A second possible solutions to increase the performance is to use the MATLAB compiler to either produce compiled MATLAB code in `.mex` files or to translate MATLAB programs into compilable language such as C.

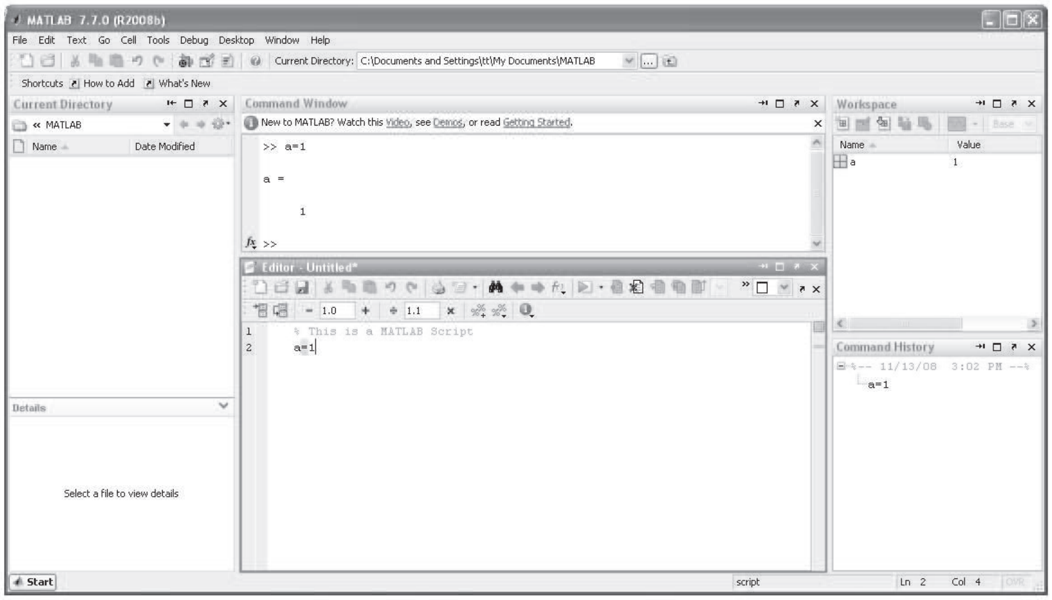
A further advantage of MATLAB is that the programming syntax supports matrix notations. This makes the code very compact and comparable to the mathematical notations used in the manuscript. MATLAB code is even useful as compact notation to describe algorithms, and it is hence useful to go through the MATLAB code in the manuscript even when not running the programs in the MATLAB environment. Furthermore, MATLAB has very powerful visualization routines, and the new versions of MATLAB include tools for documentation and publishing of codes and results. Finally, MATLAB includes implementations of many mathematical and scientific methods on which we can base our programs. For example, MATLAB includes many functions and algorithms for linear algebra and to solve systems of differential equations. Specialized collections of functions and algorithms, called a 'toolbox' in MATLAB, can be purchased in addition to the basic MATLAB package or imported from third parties, including many freely available programs and tools published by researchers. For example, the MATLAB Neural Network Toolbox incorporates functions for building and analysing standard neural networks. This toolbox covers many algorithms particularly suitable for connectionist modelling and neural network applications. A similar toolbox, called NETLAB, is freely available and contains many advanced machine learning methods. We will use some toolboxes later in this course, including the LIBSVM toolbox and the MATLAB NXT toolbox to program the Lego robots.

2.1.1 Starting a MATLAB session

Starting MATLAB opens the MATLAB desktop as shown in Fig. 2.1 for MATLAB version 7. The MATLAB desktop is comprised of several windows which can be customized or undocked (moving them into an own window). A list of these tools are available under the **desktop menu**, and includes tools such as the **command window**, **editor**, **workspace**, etc. We will use some of these tools later, but for now we only need the **MATLAB command window**. We can thus close the other windows if they are open (such as the **launch pad** or the **current directory window**); we can always get them back from the **desktop** menu. Alternatively, we can undock the command window by clicking the arrow button on the command window bar. An undocked command window is illustrated on the left in Fig. 2.2. Older versions of MATLAB start directly with a command window or simply with a MATLAB command prompt >> in a standard system window. The command window is our control centre for accessing the essential MATLAB functionalities.

2.1.2 Basic variables in MATLAB

The MATLAB programming environment is interactive in that all commands can be typed into the command window after the command prompt (see Fig. 2.2). The

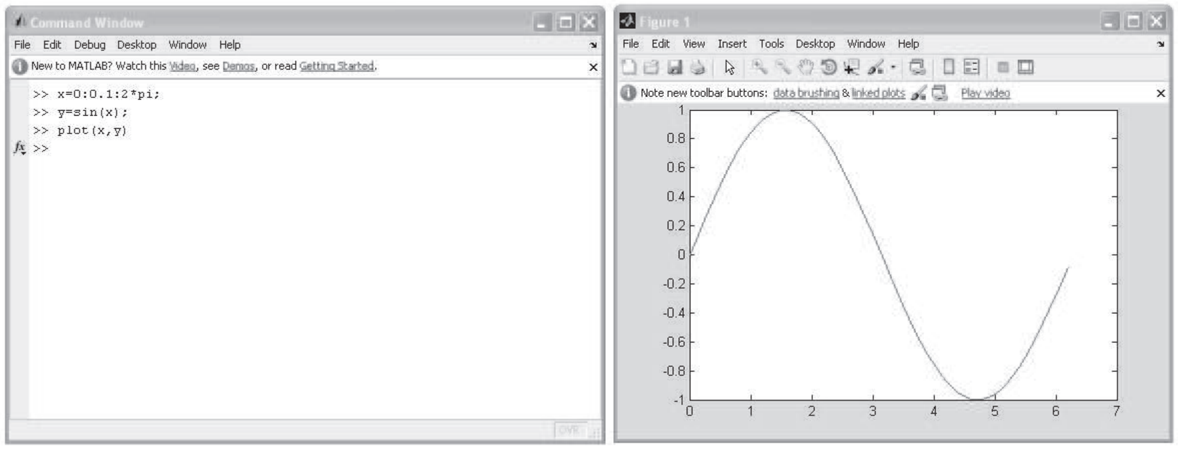

MATLAB 7.7.0 (R2008b)
File Edit Text Go Cell Tools Debug Desktop Window Help
Current Directory: C:\Documents and Settings\tt\My Documents\MATLAB
Shortcuts How to Add What's New
Current Directory
« MATLAB
Name Date Modified
Details
Select a file to view details
Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
Editor - Untitled*
Workspace
Name Value
a 1
Command History
Start
script Ln 2 Col 4

```
>> a=1

a =

     1

>>
```

```
% This is a MATLAB Script
a=1
```

```
%-- 11/13/08  3:02 PM --%
a=1
```

Fig. 2.1 The MATLAB *desktop window* of MATLAB Version 7.


Command Window
File Edit Debug Desktop Window Help
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
Figure 1
File Edit View Insert Tools Desktop Window Help
Note new toolbar buttons: data brushing & linked plots Play video
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6
7

```
>> x=0:0.1:2*pi;
>> y=sin(x);
>> plot(x,y)
>>
```

Fig. 2.2 A MATLAB *command window* (left) and a MATLAB *figure window* (right) displaying the results of the function `plot_sin` developed in the text.

commands are interpreted directly, and the result is returned to (and displayed in) the command window. For example, a variable is created and assigned a value with the = operator, such as

```
>> a=3
```

```
a =

     3
```

Ending a command with semicolon (`;`) suppresses the printing of the result on screen. It is therefore generally included in our programs unless we want to view some intermediate results. All text after a percentage sign (`%`) is not interpreted and thus treated as comment,

```
>> b='Hello World!'; % delete the semicolon to echo Hello World!
```

This example also demonstrates that the type of a variable, such as being an integer, a real number, or a string, is determined by the values assigned to the elements. This is called **dynamic typing**. Thus, variables do not have to be declared as in some other programming languages. While dynamic typing is convenient, a disadvantage is that a mistyped variable name can not be detected by the compiler. Inspecting the list of created variables is thus a useful step for debugging.

All the variables that are created by a program are kept in a buffer called **workspace**. These variable can be viewed with the command `whos` or displayed in the **workspace** window of the MATLAB desktop. For example, after declaring the variables above, the `whos` command results in the responds

```
>> whos
  Name      Size            Bytes  Class     Attributes

  a         1x1                 8  double
  b         1x12               24  char
```

It displays the name, the size, and the type (class) of the variable. The size is often useful to check the orientation of matrices as we will see below. The variables in the workspace can be used as long as MATLAB is running and as long as it is not cleared with the command `clear`. The workspace can be saved with the command `save` **filename**, which creates a file **filename**`.mat` with internal MATLAB format. The saved workspace can be reloaded into MATLAB with the command `load` **filename**. The load command can also be used to import data in ASCII format. The workspace is very convenient as we can run a program within a MATLAB session and can then work interactively with the results, for example, to plot some of the generated data.

Variables in MATLAB are generally matrices (or data arrays), which is very convenient for most of our purposes. Matrices include scalars (1×1 matrix) and vectors ($1 \times N$ matrix) as special cases. Values can be assigned to matrix elements in several ways. The most basic one is using square brackets and separating rows by a semicolon within the square brackets, for example (see Fig. 2.2),

```
>> a=[1 2 3; 4 5 6; 7 8 9]

a =

     1     2     3
     4     5     6
     7     8     9
```

A vector of elements with consecutive values can be assigned by column operators like

```
>> v=0:2:4

v =

     0     2     4
```

Furthermore, the MATLAB desktop includes an **array editor**, and data in ASCII files can be assigned to matrices when loaded into MATLAB. Also, MATLAB functions often return matrices which can be used to assign values to matrix elements. For example, a uniformly distributed random 3×3 matrix can be generated with the command

```
>> b=rand(3)

b =

    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214
```

The multiplication of two matrices, following the matrix multiplication rules, can be done in MATLAB by typing

```
>> c=a*b

c =

    3.2329    4.5549    2.9577
    8.5973   10.9730    6.8468
   13.9616   17.3911   10.7360
```

This is equivalent to

```
   c=zeros(3);
   for i=1:3
      for j=1:3
         for k=1:3
            c(i,j)=c(i,j)+a(i,k)*b(k,j);
         end
      end
   end
```

which is the common way of writing matrix multiplications in other programming languages. Formulating operations on whole matrices, rather than on the individual components separately, is not only more convenient and clear, but can enhance the programs performance considerably. Whenever possible, operations on whole matrices should be used. This is likely to be the major change in your programming style when converting from another programming language to MATLAB. The performance disadvantage of an interpreted language is often negligible when using operations on whole matrices.

The transpose operation of a matrix changes columns to rows. Thus, a row vector such as v can be changed to a column vector with the MATLAB transpose operator ('),

```
>> v'

ans =

     0
     2
     4
```

which can then be used in a matrix-vector multiplication like

```
>> a*v'

ans =

    16
    34
    52
```

The inconsistent operation a*v does produce an error,

```
>> a*v
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Component-wise operations in matrix multiplications (*), divisions (/) and potentiation ^ are indicated with a dot modifier such as

```
>> v.^2

ans =

     0     4    16
```

The most common operators and basic programming constructs in MATLAB are similar to those in other programming languages and are listed in Table 2.1.

2.1.3 Control flow and conditional operations

Besides the assignments of values to variables, and the availability of basic data structures such as arrays, a programming language needs a few basic operations for building loops and for controlling the flow of a program with conditional statements (see Table 2.1). For example, the **for loop** can be used to create the elements of the vector v above, such as

```
>> for i=1:3; v(i)=2*(i-1); end
>> v

v =
```

Table 2.1 Basic programming contracts in MATLAB.

Programming construct	**Command**	**Syntax**
Assignment	=	`a=b`
Arithmetic	add	`a+b`
operations	multiplication	`a*b` (matrix), `a.*b` (element-wise)
	division	`a/b` (matrix), `a./b` (element-wise)
	power	`a^b` (matrix), `a.^b` (element-wise)
Relational	equal	`a==b`
operators	not equal	`a~=b`
	less than	`a<b`
Logical	AND	`a & b`
operators	OR	`a‖b`
Loop	for	`for` **index=start:increment:end** statement `end`
	while	`while` **expression** statement `end`

Conditional command	if statement	`if` **logical expressions**
		statement
		`elseif` **logical expressions**
		statement
		`else`
		statement
		`end`

Function		`function [x,y,...]=`**name**`(a,b,...)`

```
     0     2     4
```

Table 2.1 lists, in addition, the syntax of a while loop. An example of a conditional statement within a loop is

```
>> for i=1:10; if i>4 & i<=7; v2(i)=1; end; end
>> v2

v2 =

     0     0     0     0     1     1     1
```

In this loop, the statement `v2(i)=1` is only executed when the loop index is larger than 4 and less or equal to 7. Thus, when `i=5`, the array `v2` with 5 elements is created, and since only the elements `v2(5)` is set to 1, the previous elements are set to 0 by default. The loop adds then the two element `v2(6)` and `v2(7)`. Such a vector can also be created by assigning the values 1 to a specified range of indices,

```
>> v3(4:7)=1

v3 =
     0     0     0     1     1     1     1
```

A 1×7 array is thereby created with elements set to 0, and only the specified elements are overwritten with the new value 1. Another method to write compact loops in MATLAB is to use vectors as index specifiers. For example, another way to create a vector with values such as `v2` or `v3` is

```
>> i=1:10

i =

     1     2     3     4     5     6     7     8     9    10

>> v4(i>4 & i<=7)=1

v4 =

     0     0     0     0     1     1     1
```

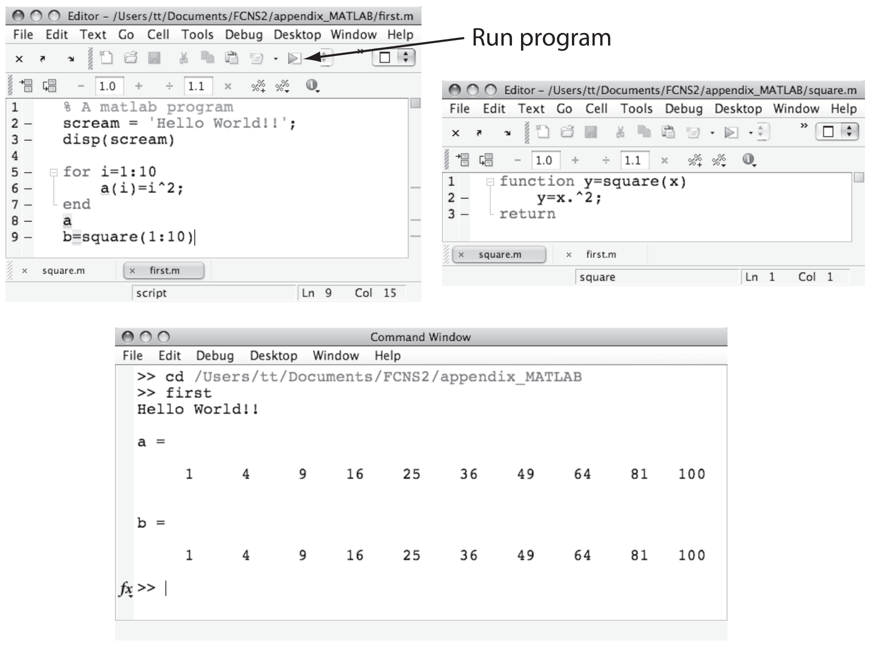
2.1.4 Creating MATLAB programs

If we want to repeat a series of commands, it is convenient to write this list of commands into an ASCII file with extension '`.m`'. Any ASCII editor (for example; WordPad, Emacs, etc.) can be used. The MATLAB package contains an editor that has the advantage of colouring the content of MATLAB programs for better readability and also provides direct links to other MATLAB tools. The list of commands in the ASCII file (e.g. **prog1**`.m`) is called a **script** in MATLAB and makes up a MATLAB program. This program can be executed with a run button in the MATLAB editor or by calling the name of the file within the command window (for example, by typing **prog1**). We assumed here that the program file is in the current directory of the MATLAB session or in one of the search paths that can be specified in MATLAB. The MATLAB desktop includes a 'current directory' window (see desktop menu). Some older MATLAB versions have instead a 'path browser'. Alternatively, one can specify absolute path when calling a program, or change the current directories with UNIX-style commands such as `cd` in the command window (see Fig. 2.3).

Functions are another way to encapsulate code. They have the additional advantage that they can be pre-compiled with the MATLAB Compiler$^{\mathrm{TM}}$ available from MathWorks, Inc. Functions are kept in files with extension '`.m`' which start with the command line like

```
function y=f(a,b)
```

where the variables `a` and `b` are passed to the function and `y` contains the values returned by the function. The return values can be assigned to a variable in the calling MATLAB


Editor – /Users/tt/Documents/FCNS2/appendix_MATLAB/first.m
File Edit Text Go Cell Tools Debug Desktop Window Help
Run program
square.m
first.m
script Ln 9 Col 15
Editor – /Users/tt/Documents/FCNS2/appendix_MATLAB/square.m
File Edit Text Go Cell Tools Debug Desktop Window Help
square Ln 1 Col 1
Command Window
File Edit Debug Desktop Window Help

```
% A matlab program
scream = 'Hello World!!';
disp(scream)

for i=1:10
    a(i)=i^2;
end
a
b=square(1:10)
```

```
function y=square(x)
    y=x.^2;
return
```

```
>> cd /Users/tt/Documents/FCNS2/appendix_MATLAB
>> first
Hello World!!

a =

     1     4     9    16    25    36    49    64    81   100


b =

     1     4     9    16    25    36    49    64    81   100

fx >> |
```

Fig. 2.3 Two editor windows and a command window.

script and added to the workspace. All other internal variables of the functions are local to the function and will not appear in the workspace of the calling script.

MATLAB has a rich collection of predefined functions implementing many algorithms, mathematical constructs, and advanced graphic handling, as well as general information and help functions. You can always search for some keywords using the useful command `lookfor` followed by the keyword (search term). This command lists all the names of the functions that include the keywords in a short description in the function file within the first **comment lines** after the function declaration in the function file. The command `help`, followed by the function name, displays the first block of comment lines in a function file. This description of functions is usually sufficient to know how to use a function. A list of some frequently used functions is listed in Table 2.1.4.

2.1.5 Graphics

MATLAB is a great tool for producing scientific graphics. We want to illustrate this by writing our first program in MATLAB: calculating and plotting the sine function. The program is

```
x=0:0.1:2*pi;
y=sin(x);
plot(x,y)
```

The first line assigns elements to a vector $\mathbf{x}$ starting with $x(1) = 0$ and incrementing the value of each further component by 0.1 until the value 2π is reached (the variable

Name	Brief description	Name	Brief description
abs	absolute functions	mod	modulus function
axis	sets axis limits	num2str	converts number to string
bar	produces bar plot	ode45	ordinary differential equation solver
ceil	round to larger interger	ones	produces matrix with unit elements
colormap	colour matrix for surface plots	plot	plot lines graphs
cos	cosine function	plot3	plot 3-dimensional graphs
diag	diagonal elements of a matrix	prod	product of elements
disp	display in command window	rand	uniformly distributed random variable
errorbar	plot with error bars	randn	normally distributed random variable
exp	exponential function	randperm	random permutations
fft	fast Fourier transform	reshape	reshaping a matrix
find	index of non-zero elements	set	sets values of parameters in structure
floor	round to smaller integer	sign	sign function
hist	produces histogram	sin	sine function
int2str	converts integer to string	sqrt	square root function
isempty	true if array is empty	std	calculates standard deviation
length	length of a vector	subplot	figure with multiple subfigures
log	logarithmic function	sum	sum of elements
lsqcurevfit	least mean square curve	surf	surface plot
	fitting (statistics toolbox)	title	writes title on plot
max	maximum value and index	view	set viewing angle of 3D plot
mix	minimum value and index	xlabel	label on x-axis of a plot
mean	calculates mean	ylabel	label on y-axis of a plot
meshgrid	creates matrix to plot grid	zeros	creates matrix of zero elements

Table 2.2 MATLAB functions used in this course. The MATLAB command `help cmd`, where `cmd` is any of the functions listed here, provides more detailed explanations.

`pi` has the appropriate value in MATLAB). The last element is $x(63) = 6.2$. The second line calls the MATLAB function `sin` with the vector `x` and assigns the results to a vector `y`. The third line calls a MATLAB plotting routine. You can type these lines into an ASCII file that you can name `plot_sin.m`. The code can be executed by typing `plot_sin` as illustrated in the **command window** in Fig. 2.2, provided that the MATLAB session points to the folder in which you placed the code. The execution of this program starts a **figure window** with the plot of the sine function as illustrated on the right in Fig. 2.2.

The appearance of a plot can easily be changed by changing the attributes of the plot. There are several functions that help in performing this task, for example, the function `axis` that can be used to set the limits of the axis. New versions of MATLAB provide window-based interfaces to the attributes of the plot. However, there are also two basic commands, `get` and `set`, that we find useful. The command `get(gca)` returns a list with the axis properties currently in effect. This command is useful for finding out what properties exist. The variable `gca` (get current axis) is the **axis handle**, which is a variable that points to a memory location where all the attribute variables are kept. The attributes of the axis can be changed with the `set` command. For example, if we want to change the size of the labels we can type `set(gca,'fontsize',18)`.

There is also a handle for the current figure `gcf` that can be used to get and set other attributes of the figure. MATLAB provides many routines to produce various special forms of plots including plots with error-bars, histograms, three-dimensional graphics, and multi-plot figures.

2.2 A first project: modelling the world

Suppose there is a simple world with a creature that can be in three distinct states, sleep (state value 1), eat (state value 2), and study (state value 3). An agent, which is a device that can sense environmental states and can generate actions, is observing this creature with poor sensors, which add white (Gaussian) noise to the true state. Our aim is to build a model of the behaviour of the creature which can be used by the agent to observe the states of the creature with some accuracy despite the limited sensors. For this exercise, the function `creature_state()` is available on the course page on the web. This function returns the current state of the creature. Try to create an agent program that predicts the current state of the creature. In the following we discuss some simple approches.

A simulation program that implements a specific agent a with simple world model (a model of the creature), which also evaluates the accuracy of the model, is given in Table 2.3. This program, also available on the web, is provided in file `main.m`. This program can be downloaded into the working directory of MATLAB and executed by typing `main` into the command window, or by opening the file in the MATLAB editor and starting it from there by pressing the icon with the green triangle. The program reports the percentage of correct perceptions of the creature's state.

Line 1 of the program uses a comment indicator (`%`) to outline the purpose of the program. Line 2 clears the workspace to erase all eventual existing variables, and sets a counter for the number of correct perceptions to zero. Line 4 starts a loop over 1000 trials. In each trial, a creature state is pulled by calling the function `creature_state()` and recording this state value in variable `x`. The sensory state `s` is then calculated by adding a random number to this value. The value of the random number is generated from a normal distribution, a Gaussian distribution with mean zero and unit variance, with the MATLAB function `randn()`.

We are now ready to build a model for the agent to interpret the sensory state. In the example shown, this model is given in Lines 8–12. This model assumes that a sensory value below 1.5 corresponds to the state of a sleeping creature (Line 9), a sensory value between 1.5 and 2.5 corresponds to the creature eating (Line 10), and a higher value corresponds to the creature studying (Line 11). Note that we made several assumptions by defining this model, which might be unreasonable in real-world applications. For example, we used our knowledge that there are three states with ideal values of 1, 2, and 3 to build the model for the agent. Furthermore, we used the knowledge that the sensors are adding independent noise to these states in order to come up with the decision boundaries. The major challenge for real agents is to build models without this explicit knowledge. When running the program we find that a little bit over 50% of the cases are correctly perceived by the agent. While this is a good start, one could do better. Try some of your own ideas . . .

Table 2.3 Program main.m

```
% Project 1: simulation of agent which models simple creature
clear; correct=0;

for trial=1:1000
    x=creature_state();
    s=x+randn();

    %% perception model
    if (s<1.5) x_predict=1;
    elseif (s<2.5) x_predict=2;
    else x_predict=3;
    end

    %% calculate accuracy
    if (x==x_predict) correct=correct+1; end
 end

disp(['percentage correct: ',num2str(correct/1000)]);
```

. . . Did you succeed in getting better results? It is certainly not easy to guess some better model, and it is time to inspect the data more carefully. For example, we can plot the number of times each state occurs. For this we can write a loop to record the states in a vector,

```
>> for i=1:1000; a(i)=creature_state(); end
```

and then plot a histogram with the MATLAB function `hist()`,

```
>> hist(a)
```

The result is shown in Fig. 2.4. This histogram shows that not all states are equally likely as we implicitly assumed in the above agent model. The third state is indeed much less likely. We could use this knowledge in a modified model in which we predict that the agent is sleeping for sensory states less than 1.5 and is eating otherwise. This modified model, which completely ignores study states, predicts around 65% of the states correctly. Many machine learning methods suffer from such 'explaining away' solutions for imbalanced data, as further discussed in Chapter **??**.

It is important to recognize that 100% accuracy is not achievable with the inherent limitations of the sensors. However, higher recognition rates could be achieved with better world (creature + sensor) models. The main question is how to find such a model. We certainly should use observed data in a better way. For example, we could use several observations to estimate how many states are produced by function `creature_state()` and their relative frequency. Such parameter estimation is a basic form of learning from data. Many models in science take such an approach by proposing a parametric model and estimating parameters from the data by model fitting. The main challenge with this approach is how complex we should make the model. It is much easier to fit a more complex model with many parameters to example data, but the

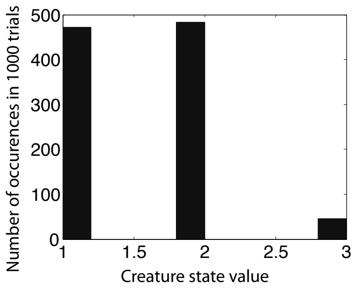


Fig. 2.4 The MATLAB *desktop window* histogram of states produced by function `creature_state()` from 1000 trials.

increased flexibility decreases the prediction ability of such models. Much progress has been made in machine learning by considering such questions, but those approaches only work well in limited worlds, certainly much more restricted than the world we live in. More powerful methods can be expected by learning how the brain solves such problems.

2.3 Alternative programming environments: Octave and Scilab

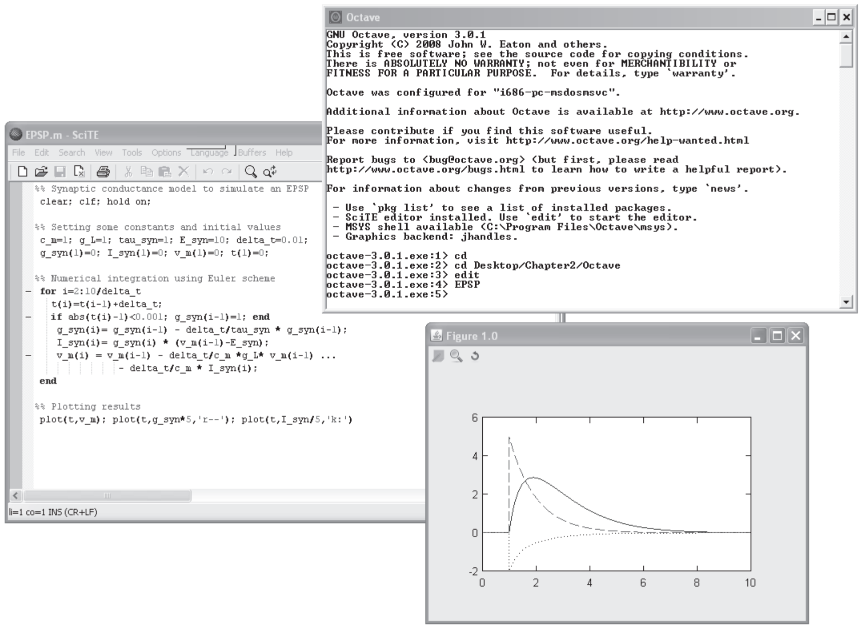
We briefly mention here two programming environments that are very similar to Matlab and that can, with certain restrictions, execute Matlab scripts. Both of these programming systems are open source environments and have general public licenses for non-commercial use.

The programming environment called **Octave** is freely available under the GNU general public license. Octave is available through links at http://www.gnu.org/software/octave/. The installation requires the additional installation of a graphics package, such as gnuplot or Java graphics. Some distributions contain the SciTE editor which can be used in this environment. An example of the environment is shown in Fig. 2.5

Scilab is another scientific programming environment similar to MATLAB. This software package is freely available under the CeCILL software license, a license compatible to the GNU general public license. It is developed by the Scilab consortium, initiated by the French research centre INRIA. The Scilab package includes a MATLAB import facility that can be used to translate MATLAB programs to Scilab. A screen shot of the Scilab environment is shown in Fig. 2.6. A Scilab script can be run from the execute menu in the editor, or by calling `exec("filename.sce")`.

2.4 Exercises

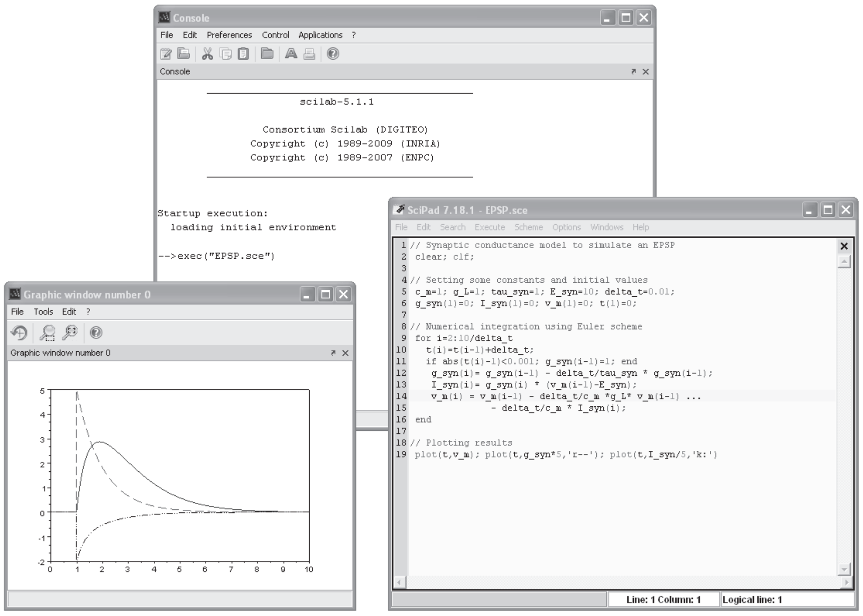
1. Write a Matlab function that takes a character string and prints out the character string in reverse order. reverses program
2. Write a Matlab program that plots a two dimensional gaussian function.



```
GNU Octave, version 3.0.1
Copyright (C) 2008 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTIBILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type `warranty'.

Octave was configured for "i686-pc-msdosmsvc".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/help-wanted.html

Report bugs to <bug@octave.org> (but first, please read
http://www.octave.org/bugs.html to learn how to write a helpful report).

For information about changes from previous versions, type `news'.

 - Use `pkg list' to see a list of installed packages.
 - SciTE editor installed. Use `edit' to start the editor.
 - MSYS shell available (C:\Program Files\Octave\msys).
 - Graphics backend: jhandles.

octave-3.0.1.exe:1> cd
octave-3.0.1.exe:2> cd Desktop/Chapter2/Octave
octave-3.0.1.exe:3> edit
octave-3.0.1.exe:4> EPSP
octave-3.0.1.exe:5>
```

```
%% Synaptic conductance model to simulate an EPSP
 clear; clf; hold on;

%% Setting some constants and initial values
 c_m=1; g_L=1; tau_syn=1; E_syn=10; delta_t=0.01;
 g_syn(1)=0; I_syn(1)=0; v_m(1)=0; t(1)=0;

%% Numerical integration using Euler scheme
 for i=2:10/delta_t
   t(i)=t(i-1)+delta_t;
   if abs(t(i)-1)<0.001; g_syn(i-1)=1; end
    g_syn(i)= g_syn(i-1) - delta_t/tau_syn * g_syn(i-1);
    I_syn(i)= g_syn(i) * (v_m(i-1)-E_syn);
    v_m(i) = v_m(i-1) - delta_t/c_m *g_L* v_m(i-1) ...
                   - delta_t/c_m * I_syn(i);
 end

%% Plotting results
 plot(t,v_m); plot(t,g_syn*5,'r--'); plot(t,I_syn/5,'k:')
```

Fig. 2.5 The Octave programming environment with the main console, and editor called *SciTE*, and a graphics window.



```
              scilab-5.1.1

          Consortium Scilab (DIGITEO)
        Copyright (c) 1989-2009 (INRIA)
        Copyright (c) 1989-2007 (ENPC)


Startup execution:
  loading initial environment

-->exec("EPSP.sce")
```

```
// Synaptic conductance model to simulate an EPSP
 clear; clf;

// Setting some constants and initial values
 c_m=1; g_L=1; tau_syn=1; E_syn=10; delta_t=0.01;
 g_syn(1)=0; I_syn(1)=0; v_m(1)=0; t(1)=0;

// Numerical integration using Euler scheme
 for i=2:10/delta_t
   t(i)=t(i-1)+delta_t;
   if abs(t(i)-1)<0.001; g_syn(i-1)=1; end
    g_syn(i)= g_syn(i-1) - delta_t/tau_syn * g_syn(i-1);
    I_syn(i)= g_syn(i) * (v_m(i-1)-E_syn);
    v_m(i) = v_m(i-1) - delta_t/c_m *g_L* v_m(i-1) ...
                 - delta_t/c_m * I_syn(i);
 end

// Plotting results
 plot(t,v_m); plot(t,g_syn*5,'r--'); plot(t,I_syn/5,'k:')
```

Fig. 2.6 The Scilab programming environment with *console*, and editor called *SciPad*, and a graphics window.

3 Basic robotics with Lego NXT

3.1 Introduction

"Robotics is the science of perceiving and manipulating the physical world through computer-controlled devices"[2]. We use the word **robot** or **agent** to describe a system which interacts actively with the environment through **sensors** and **actuators**. An agent can be implemented in software or hardware, and we often use the term **robot** more specifically for a hardware implementation of an agent, though this does generally include software components. **Active interactions** means that the robot's position in the environment or the environment itself changes though the action of the robot. **Mobile robots** are a good example of this and are mainly used in the examples of this course. In contrast, a vision system that uses a digital camera to recognize objects is not a robot in our definition.

The word 'robot' was first used in print in a novel by Isaac Asimov (1941), and the first industrial robots were installed in 1961 (Unimate). Robots are now invaluable in manufacturing, and research is ongoing to make robots more autonomous and to make those machines more robust and able to work in hostile environments. Robotics has many components, including mechanical and electrical engineering, computer or machine vision, and even behavioral studies have become prominent in this field.

In this course, we will demonstrate many of the discussions, algorithms and associated problems with the help of computer simulations and robots. Computer simulations are a great way to experiment with many of the ideas, but physical implementations have often the advantage to show more clearly the challenges in real applications.

We will use the LEGO mindstorm system in this course. Besides having common LEGO building blocks to construct different designs, this system consist of a microprocessor, called the **brick**, which is programmable and which controls the sensors and actuators. The actuators are stepping motors that can be told to run for a specific duration, a specific number of rotations, or with a specific speed. Our tool kit also includes several sensors, a light sensor that can be used to measure the wavelength of reflecting light and also small distances, an ultrasonic sensor to measure larger distances, a touch sensor, and a microphone. The motors can also be used to sense some externally applied movements.

We will use a basic tribot design as shown in Fig.3.1 for most of the explorations in this course. We will build the basic tribot in the following tutorial and will outline an example of a program to avoids walls and then work on a program so that the tribot can follow a line. After this we will use a robot arm configuration to discuss the important concept of a configuration space and demonstrate path-planing.

[2]Probabilistic Robotics, Sebastian Thrun, Wolfram Burgard, and Dieter Fox, MIT press 2006

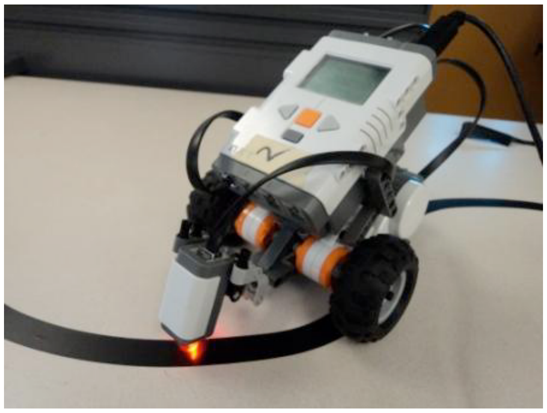


Fig. 3.1 Basic Lego Robot with microprocessor, two motors, and a light sensor.

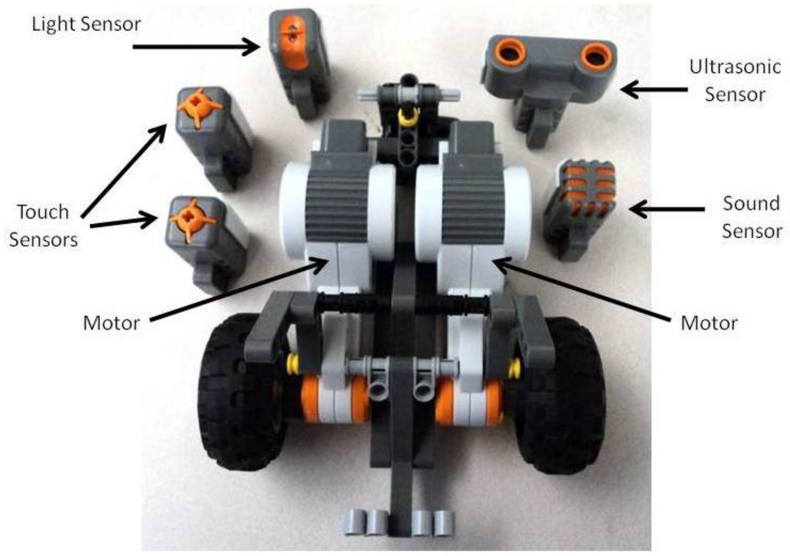
3.2 Building a basic robot with Lego NXT

3.2.1 Building the tribot

1. **Charge the battery**
   Before constructing the robot, plug the battery pack into the NXT brick controller and plug it into an outlet to recharge the battery.

2. **Construct the NXT**
   Follow the instruction booklet included with the Lego kit to construct the robot. There are mini programs illustrated in the manual that can be implemented directly on the NXT brick. Although this is not the method that will be used to control the NXT, you may implement them if you have time in order to get an idea of how the sensors and motors function. The instruction manual is organized as follows:

Section	Pages
NXT brick base construction	8-23
Sound sensor mounting	24-27
Ultrasonic sensor mounting	28-31
Light sensor mounting	32-39
Touch sensor mounting	40-45



Sensors & constructed NXT brick base

3.2.2 Mindstorm NXT toolbox instalation

We will use some software to control the Lego actuator and gather information from their sensors within the Matlab programming environment. To enable this we need to install software developed at the German university called ‘RWTH Aachen’, which in turn uses some other drivers that we need to install. Most of the software should be installed in our Lab, but we will outline briefly some of the installation issues in case you want to install them under your own system or if some problems exists with the current installation. The following software installation instructions are adapted from RWTH Aachen University’s NXT Toolbox website:
http://www.mindstorms.rwth-aachen.de/trac/wiki/Download4.03

1. **Check NXT Firmware version**
   Check what version of NXT Firmware is running on the NXT brick by going to "Settings" > "NXT Version". Firmware version ("FW") should be 1.28. If it does not, it needs to be updated (Note: The NXT toolbox website claims version 1.26 will work, however it will not)
   To update the firmware:
   The Lego Mindstorms Education NXT Programming software is required to update the firmware. In the NXT Programming software, look under "tools" > "Update NXT Firmware" > "browse", select the firmware’s directory, click "download".

2. **Install USB (Fantom) Driver (Windows only)**
   If the Lego Mindstorms Education NXT Programming software is already on your computer, this should already be installed. Otherwise, download it from: http://mindstorms.lego.com/support/updates/
   If you run into problems with the Fantom Library on windows go to this site for help. http://bricxcc.sourceforge.net/NXTFantomDriverHelp.pdf
   If you have Windows 7 Starter edition the standard setup file will not run properly. To install the Fantom Driver go into Products and then LEGO_NXT_Driver_32 and run LegoMindstormsNXTdriver32.
   The fantom USB driver seem not to work on the Mac, but we will anyhow use the bluetooth connections.
3. **Download the Mindstorms NXT Toolbox 4.03:**
   Download: http://www.mindstorms.rwth-aachen.de/trac/wiki/Download
   - Save and extract the files anywhere, but do not change the directory structure.
   - The folder will appear as "RWTHMindstormsNXT"
4. **Install NXT Toolbox into MATLAB**
   In MATLAB: "File" > "SetPath" > "Add Folder", and browse and select "RWTHMindstormsNXT" - the file you saved in the previous step.
   - Also add the "tools" folder, which is a subdirectory of the RWTHMindstormsNXT folder.
   - Click "save" when finished.
5. **Download MotorControl to NXT brick**
   Go to http://bricxcc.sourceforge.net/utilities.html for the download. Use the USB cable for this step

Windows: Download NeXTTool.exe to RWTHMindstormsNXT/tools/MotorControl. Under RWTHMindstormsNXT/tools/MotorControl, double click TransferMotorControlBinaryToNXT, click "Run", and follow the onscreen instructions. If this fails, try using the NBC compiler (download from http://bricxcc.sourceforge.net/nbc/) instead of the NeXTTool; again save it under the MotorControl folder.

Mac: Download the NeXTTools for Mac OS X. Run the toolbar and open the XNT Explorer (the globe in the toolbar). With the arrow key at the top, transfer the file MotorControl21.rxe to the brick.

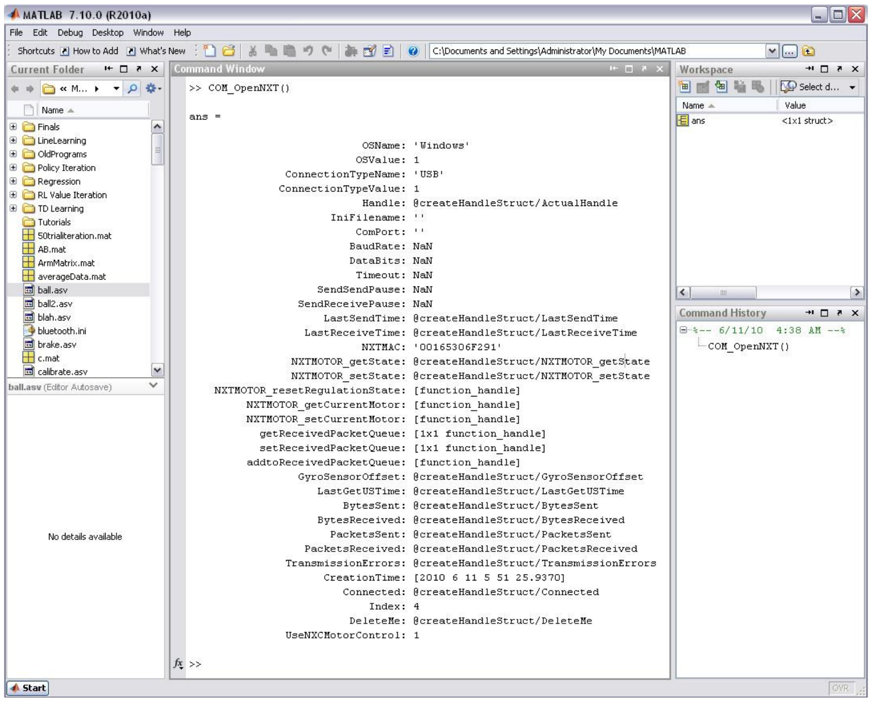
6. **Setting up a Bluetooth connection**

   To connect to the NXT via bluetooth you must first turn on the bluetooth in the NXT and make sure that the visibility is set to on. Then use the bluetooth device on your computer to search for your specific NXT. By default the name is NXT, but as a first step we will rename each brick.

   Create a connection between the computer and the NXT. When you create the connection between the NXT and the bluetooth device the NXT will ask for a passkey (usually either 0000 or 1234 on the NXT screen and press the orange button. The computer will then ask for the same passkey. To test the connection, type the command `COM_OpenNXT('bluetooth.ini');` in the Matlab command window. The command should run without any red error messages.

If there is an error check to see if the COMPort the Matlab code is looking for is the same as the one used in the connection made between the bluetooth device and the NXT. Also turning the NXT off and back on again can help. After every failed `COM_OpenNXT('bluetooth.ini');` command type `COM_CloseNXT('all');` to close the failed connection for a clean new attempt. To switch on a debug mode enter the command `DebugMode on` before entering the command `COM_OpenNXT('bluetooth.ini');` . Also, make sure that the bluetooth.ini file is present. There are sample files for Windows ad Linux (Mac) in the main RWTH toolbox folder. Also, check if the port name is correct by typing `ls -ltr /dev` in a terminal window.

7. **Does it work?**
   In MATLAB, enter the commands below into the command window. The command should execute without error and the NXT should play a sound.
   ```
   h=COM_OpenNXT('bluetooth.ini');
   COM_SetDefaultNXT(h);
   NXT_PlayTone(400,300);
   ```



```
>> COM_OpenNXT()

ans =

                       OSName: 'Windows'
                      OSValue: 1
           ConnectionTypeName: 'USB'
          ConnectionTypeValue: 1
                       Handle: @createHandleStruct/ActualHandle
                  IniFilename: ''
                      ComPort: ''
                     BaudRate: NaN
                     DataBits: NaN
                      Timeout: NaN
                SendSendPause: NaN
             SendReceivePause: NaN
                 LastSendTime: @createHandleStruct/LastSendTime
              LastReceiveTime: @createHandleStruct/LastReceiveTime
                       NXTMAC: '00165306F291'
            NXTMOTOR_getState: @createHandleStruct/NXTMOTOR_getState
            NXTMOTOR_setState: @createHandleStruct/NXTMOTOR_setState
NXTMOTOR_resetRegulationState: [function_handle]
     NXTMOTOR_getCurrentMotor: [function_handle]
     NXTMOTOR_setCurrentMotor: [function_handle]
       getReceivedPacketQueue: [1x1 function_handle]
       setReceivedPacketQueue: [1x1 function_handle]
     addtoReceivedPacketQueue: [function_handle]
             GyroSensorOffset: @createHandleStruct/GyroSensorOffset
                LastGetUSTime: @createHandleStruct/LastGetUSTime
                    BytesSent: @createHandleStruct/BytesSent
                BytesReceived: @createHandleStruct/BytesReceived
                  PacketsSent: @createHandleStruct/PacketsSent
              PacketsReceived: @createHandleStruct/PacketsReceived
           TransmissionErrors: @createHandleStruct/TransmissionErrors
                 CreationTime: [2010 6 11 5 51 25.9370]
                    Connected: @createHandleStruct/Connected
                        Index: 4
                     DeleteMe: @createHandleStruct/DeleteMe
           UseNXCMotorControl: 1

>>
```

A list of Matlab commands from the NXT toolbox can be found in Appendix A. Also, there are some examples included with the RWTH Mindstorm's NXT Toolbox, under RWTHMindstormsNXT/demos.

3.3 First examples

The following exercises are intended to explore how to use RWTH's Mindstorms NXT Toolbox.

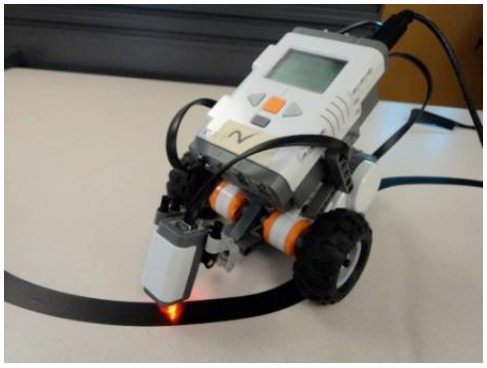
3.3.1 Example 1: Wall avoidance

The following is a simple example of how to drive a robot and use the ultrasonic sensor. The robot will drive forward until it is around 20 cm away from a barrier (i.e. a wall), stop, beep, turn right, and continue moving forward. The robot will repeat this 5 times. Attach the Ultrasonic sensor and connect it to port 1. The study and run the following program.

```
COM_CloseNXT('all');                                  %cleans up workspace
close all;
clear all;
hNXT=COM_OpenNXT('bluetooth.ini');                    % initiates NXT, hNXT is an arbitrary name
COM_SetDefaultNXT(hNXT);                              %sets default handle

OpenUltrasonic(SENSOR_1);

forward=NXTMotor('BC';); &% setting motors B          C to drive forward
forward.Power=50;
forward.TachoLimit=0;
turnRight=NXTMotor('B'); &% setting motor B to turn right
turnRight.Power=50;
turnRight.TachoLimit=360;
for i= 1:5
    while GetUltrasonic(SENSOR_1)>20
        forward.SendToNXT();                          %sends command for robot to move forward
                                                      %TachoLimit=0; no need for a WaitFor() statement
    end %while
    forward.Stop('brake');                            %robot brakes from going forward
    NXT_PlayTone(400,300);                            %plays a note
    turnRight.SendToNXT;                              %sends the command to turn right
    turnRight.WaitFor;                                %TachoLimit is not 0; WaitFor() statement required
end %for
turnRight.Stop('off');                                %properly closes motors
forward.Stop('off');
CloseSensor(SENSOR_1);                                %properly closes the sensor
COM_CloseNXT(hNXT);                                   % properly closes the NXT
close all;
clear all;
```

3.3.2 Example 2: Line following

The next exercise is writing your own program that uses readings from its light sensor to drive the NXT and follow a line.



Setup:

1. Mount light sensor, facing downwards on front of NXT and plugged into Port 3.
2. Mount a switch sensor on the NXT, plugged into Port 2.
3. Use a piece of dark tape (i.e. electrical tape) to mark a track on a flat, light coloured surface. Make sure the tape and the surface are coloured differently enough that the light sensor returns reasonably different values between the two surfaces.
4. Write a program so that the tibot follows the line.

3.4 Configuration space

3.4.1 Abstracting the states of a system

Very important for most algorithms to control a robot is the description of its state or the possible states of the system. The physical state of an agent can be quite complicated. For example, the Lego components of the tribot can have different positions and can shift in operation; the battery will have different levels of charge during the day, lightening conditions change, etc. Thus, description of the physical state would need a lot of variables, and such a description space would be very high dimensional, which is one important source of the computational challenges we face. To manage this problem we need to consider abstractions.

Abstraction is an important concept in computer science and science in general. To abstract means to simplify the system in a way that it can be described in as simple terms as possible to answer the specific questions under consoderation. This philosophy is sometimes called the **principle of parsimony**, also known as **Occam's razor**. Basically, we want a model as simple as possible, while still capturing the main aspects of the system that the model should capture.

For example, let us consider a robot that should navigate around some obstacles from point A to point B as shown in Fig.3.2A. The simplifications we make in the following is that we take consider the movement in only in a two-dimensional (2D) plane. The robot will have a position described by an x and y coordinate, and a heading direction described by an angle α. Note that we ignore for now the physical extension of the robot, that is, we consider it here only as a point object. If this is a problem, we can add an extension parameter later. We also ignore many of the other physical parameters mention above. The current state of robot is hence described by three real numbers, $[x, y, \alpha]$, and the space of all possible values is called the **configuration space** or **state space**. An obstacle, as shown in Fig.3.2A with the grey area, can be represented as state values that are not allowed by the system.

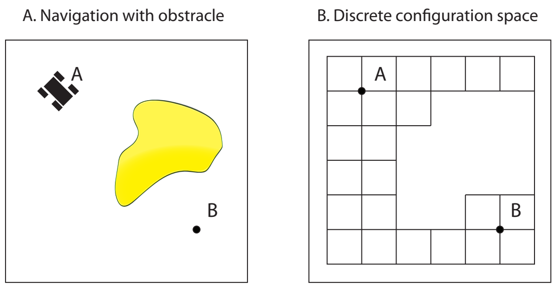


Fig. 3.2 (A) A physical space with obstacles in which an agent should navigate from point A to point B. (B) Descretized configuration space.

We have reduced the description of an robot navigation system from a nearly infinite physical space to a three-dimensional (3D) configuration space by the process of abstraction. However, there are still an infinite number of possible states in the state space if the state variables are real numbers. We will shortly see that we have often to search the state space, and an infinite space then often becomes problematic. A common solution to this problem is to make further abstractions and to allow the state variables to have only a finite number of states. Thus, we commonly **discretize** the state space. An example is shown in Fig.3.2B, where we used a grid to represent the possible values of the robot positions. Such a discretization is very useful in practice, and we can also make the discretization error increasingly small by decreasing the grid parameter, Δx, which describes the minimal distance between two states. We will use this grid discretization for planing a path through the configuration space later in this chapter.

3.4.2 Robot Arm State Space Visualizer

The goal of the following exercise is to graph the state space in which a double jointed robot arm is able to move. To do this, use two motors of the NXT Lego kit as joins of this arm, similarly to the one pictured in Fig.3.3. Also attach a touch sensor that we will use as a button in the following. Secure it on the table so that you can move its pointing finger around by turning the externally moving the motors. We will use the ability of measuring this movement in the motors by the Lego system. Also, use a box, a coffee mug or some other items to create some obstacles for the arm to move around.

Use the Matlab program in Table 3.1 to visualize the state space. Start the Matlab program with the robot arm touching one of the obstacles. Then move the arm all around one of the obstacles. This will allow the NXT to create set of data that will display which combinations of angles cannot be travelled through by the robot arm. To map more than one obstacle without mapping the angles in between the two obstacles, push the button. The button pauses the program for 4 seconds so that you can move

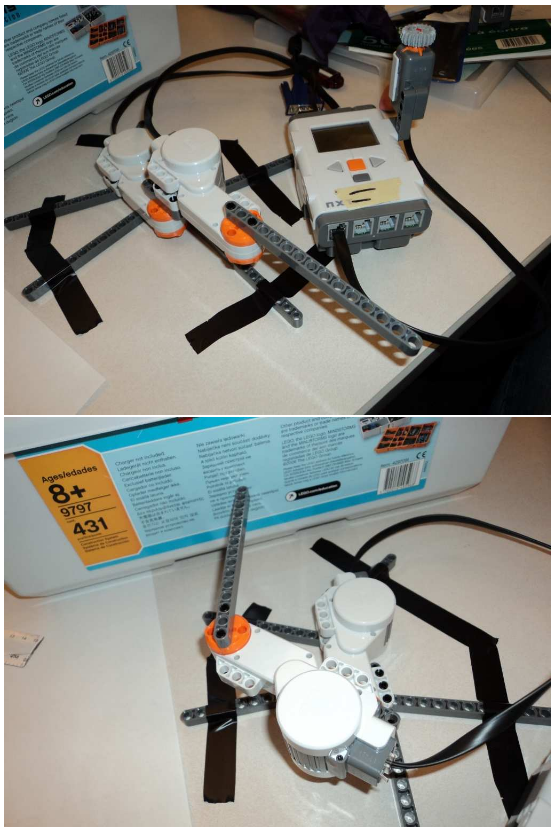


Fig. 3.3 (A) Robot arm with two joints. (B)

the arm from one obstacle to the next without mapping the angles in between the two.

3.4.3 Exercise

1. Write a program that produces a map of the state space in form of a matrix which has zeros of entries that can be reached and ones as entries for states that have obstacles in it.

3.5 Planning

Planing a movement is basic requirement in robotics. In particular, we want to find the **shortest path** from a starting position to a goal position. The previous exercise provides us with a **map** that we can use to plan the movement. One of the elements of the array is dedicated as the starting position, and one as the goal position. We now need to write a search algorithm that finds the shortest path.

There are many search algorithms that we could utilize for this task. For example, a basic search algorithm is **depth-first-search** that choses a neighboring node, than

Table 3.1 State space visualization program

```
for counter = 1:10000
    %The angles for both wheels are read
    bottomDeg=bottom.ReadFromNXT.Position;
    topDeg=top.ReadFromNXT.Position;
    %the angles are then stored in angleWheel
    angleWheel(1,counter)=bottomDeg;
    angleWheel(2,counter)=topDeg;
    %This plot will show all the angles read from the
    %wheels, because hold is on
    plot(topDeg,bottomDeg,'o');
    %If the switch is pushed then pause the angle reader
    if GetSwitch(SENSOR_1)==1
        disp('paused');
        pause(4);
        disp('unpaused');
    end
end
```

moves to the next, etc, until it reaches a goal. If the algorithm ends up in a dead-end, than the algorithm tracks back to the last node with another options and goes down this node. The **breadth-first-search** algorithms searches instead first if each of the neighboring nodes of a current node is the goal state before advancing to the next level. Of course, both need to keep track of which routs have been tried already to ensure that we do not continuously try failed paths. These algorithms would sooner or later find a path between the starting position and the goal if it exists since these algorithms would try out all possible paths. However, these algorithms do not grantee to find the shortest path, and these algorithms are also usually not so efficient.

To find better solutions we could take more information into account. For example, in the grid world, we can use some distance measure between the current state and the goal, such as the **Euclidean distance**

$$g(\text{current}) = \sqrt{(x_{\text{goal}} - x_{\text{current}})^2 + (y_{\text{goal}} - y_{\text{current}})^2}, \tag{3.1}$$

to guide the search[3]. That is, if we chose the next node to expand, we would chose the node that has the smallest distance to the goal node. Such a **heuristic search algorithm** is sometimes called a **gready best-first search**. Valid heuristics, which are estimations of the expected cost that must not underestimate the real cost, can greatly accelerate search performance. We can take such strategy a step further by also taken into account in the cost of reaching the current state from the starting state,

$$h(\text{current}) = \sqrt{(x_{\text{start}} - x_{\text{current}})^2 + (y_{\text{start}} - y_{\text{current}})^2}, \tag{3.2}$$

[3]This distance measure is also called the L_2 norm. Other measures such as the Manhattan distance, the Minkowski distance, or the L_1 norm also frequently used.

so that the total heuristic cost of node 'current' is

$$f(\text{current}) = h(\text{current}) + g(\text{current}). \tag{3.3}$$

The algorithm that uses this heuristics while taking track of paths that are still 'open' or that are 'closed' since they lead to dead-ends, is called an A* star search algorithm. An Matlab example and visualization of the A* search algorithm is provided on the course web page.

3.5.1 Exercise: Robot Arm State Space Traversal

The goal of this exercise is to make the double jointed NXT arm progress through the state space. You need the double-joined robot arm and the obstacles from the state space visualization exercise above.

- Make a program which can move the arm from the starting coordinate to the ending coordinate while avoiding the obstacles in the state space.

4 Basic probability theory

As outlined in Chapter 1, a major milestone for the modern approach to machine learning is to acknowledge our limited knowledge about the world and the unreliability of sensors and actuators. It is then only natural to consider quantities in our approaches as **random variables**. While a regular variable, once set, has only one specific value, a random variable will have different values every time we 'look' at it (draw an example from the distributions). Just think about a light sensor. We might think that an ideal light sensor will give us only one reading while holding it to a specific surface, but since the peripheral light conditions change, the characteristics of the internal electronic might change due to changing temperatures, or since we move the sensor unintentionally away from the surface, it is more than likely that we get different readings over time. Therefore, even internal variables that have to be estimated from sensors, such as the state of the system, is fundamentally a random variable.

Having a random variable does not mean that we can not say anything about the variable. Some numbers might be more likely than others when applying the sensor. This knowledge, how likely each value is for a random variable x, is captured by the probability density function $pdf(x)$.

Most of the systems discussed in this course are **stochastic models** to capture the uncertainties in the world. Stochastic models are models with random variables, and it is therefore useful to remind ourselves about the properties of such variables. This chapter is a refresher on concepts in probability theory. Note that we are mainly interested in the language of probability theory rather than statistics, which is more concerned with hypothesis testing and related procedures.

4.1 Random numbers and their probability (density) function

Probability theory is the theory of **random numbers**. We denoted such numbers by capital letters to distinguish them from regular numbers written in lower case. A random variable, X, is a quantity that can have different values each time the variable is inspected, such as in measurements in experiments. This is fundamentally different to a regular variable, x, which does not change its value once it is assigned. A random number is thus a new mathematical concept, not included in the regular mathematics of numbers. A specific value of a random number is still meaningful as it might influence specific processes in a deterministic way. However, since a random number can change every time it is inspected, it is also useful to describe more general properties when drawing examples many times. The frequency with which numbers can occur is then the most useful quantity to take into account. This frequency is captured by the mathematical construct of a **probability**.

We can formalize this with some compact notations. We speak of a **discrete random variable** in the case of discrete numbers for the possible values of a random number. A **continuous random variable** is a random variable that has possible values in a continuous set of numbers. There is, in principle, not much difference between these two kinds of random variables, except that the mathematical formulation has to be slightly different to be mathematically correct. For example, the **probability function**,

$$P_X(x) = P(X = x) \tag{4.1}$$

describes the frequency with which each possible value x of a discrete variable X occurs. Note that x is a regular variable, not a random variable. The value of $P_X(x)$ gives the fraction of the times we get a value x for the random variable X if we draw many examples of the random variable.[4] From this definition it follows that the frequency of having any of the possible values is equal to one, which is the normalization condition

$$\sum_x P_X(x) = 1. \tag{4.2}$$

In the case of continuous random numbers we have an infinite number of possible values x so that the fraction for each number becomes infinitesimally small. It is then appropriate to write the probability distribution function as $P_X(x) = p_X(x)\mathrm{d}x$, where $p_X(x)$ is the **probability density function** (pdf). The sum in eqn 4.2 then becomes an integral, and normalization condition for a continuous random variable is

$$\int_x p_X(x)\mathrm{d}x = 1. \tag{4.3}$$

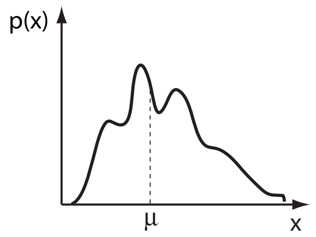
We will formulate the rest of this section in terms of continuous random variables. The corresponding formulas for discrete random variables can easily be deduced by replacing the integrals over the pdf with sums over the probability function. It is also possible to use the δ-function, outlined in Appendix **??**, to write discrete random processes in a continuous form.

4.2 Moments: mean, variance, etc.

In the following we only consider independent random values that are drawn from identical pdfs, often labeled as iid (independent and identically distributed) data. That is, we do not consider cases where there is a different probabilities of getting certain numbers when having a specific number in a previous trial. The static probability density function describes, then, all we can know about the corresponding random variable.

Let us consider the arbitrary pdf, $p_X(x)$, with the following graph:

[4] Probabilities are sometimes written as a percentage, but we will stick to the fractional notation.



Such a distribution is called **multimodal** because it has several peaks. Since this is a pdf, the area under this curve must be equal to one, as stated in eqn 4.3. It would be useful to have this function parameterized in an analytical format. Most pdfs have to be approximated from experiments, and a common method is then to fit a function to the data. However, sometimes it is sufficient to know at least some basic characteristics of the functions. For example, we might ask what the most frequent value is when drawing many examples. This number is given by the largest peak value of the distribution. A more common quantity to know is the expected arithmetic average of those numbers, which is called the **mean**, **expected value**, or **expectation value** of the distribution, defined by

$$\mu = \int_{-\infty}^{\infty} x f(x) \mathrm{d}x. \tag{4.4}$$

In the discrete case, this formula corresponds to the formula of calculating an arithmetic average, where we add up all the different numbers together with their frequency. Formally, we need to distinguish between a quantity calculated from random numbers and quantities calculated from the pdfs. If we treat the pdf as fundamental, then the arithmetic average is like an estimation of the mean. This is usually how it is viewed. However, we could also be pragmatic and say that we only have a collection of measurements so that the numbers are the ‘real’ thing, and that pdfs are only a mathematical construct. While this is mainly a philosophical debate, we try to be consistent in calling the quantities derived from data ‘estimates’ of the quantities defined through pdfs.

The mean of a distribution is not the only interesting quantity that characterizes a distribution. For example, we might want to ask what the **median** value is for which it is equally likely to find a value lower or larger than this value. Furthermore, the spread of the pdf around the mean is also very revealing as it gives us a sense of how spread the values are. This spread is often characterized by the standard deviation (std), or its square, which is called **variance**, σ^2, and is defined as

$$\sigma^2 = \int_{-\infty}^{\infty} (x-\mu)^2 f(x) \mathrm{d}x. \tag{4.5}$$

This quantity is generally not enough to characterize the probability function uniquely; this is only possible if we know all moments of a distribution, where the nth **moment about the mean** is defined as

$$m^n = \int_{-\infty}^{\infty} (x-\mu)^n f(x) \mathrm{d}x. \tag{4.6}$$

The **variance** is the second moment about the mean,

$$\sigma^2 = \int_{-\infty}^{\infty} (x - \mu)^2 f(x)\mathrm{d}x. \tag{4.7}$$

Higher moments specify further characteristics such as the kurtosis and skewness of the distribution. Moments higher than this have not been given explicit names. Knowing all moments of a distribution is equivalent in knowing the distribution precisely, and knowing a pdf is equivalent in knowing everything we could know about a random variable.

In case the distribution function is not given, moments have to be estimated from data. For example, the mean can be estimated from a sample of measurements by the **sample mean**,

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \tag{4.8}$$

and the variance either from the **biased sample variance**,

$$s_1^2 = \frac{1}{n}\sum_{i=1}^{n}(\bar{x} - x_i)^2, \tag{4.9}$$

or the **unbiased sample variance**

$$s_2^2 = \frac{1}{n-1}\sum_{i=1}^{n}(\bar{x} - x_i)^2. \tag{4.10}$$

A statistic is said to be biased if the mean of the sampling distribution is not equal to the parameter that is intended to be estimated. Knowing all moments uniquely specifies a pdf.

4.3 Examples of probability (density) functions

There is an infinite number of possible pdfs. However, some specific forms have been very useful for describing some specific processes and have thus been given names. The following is a list of examples with some discrete and several continuous distributions. Most examples are discussed as one-dimensional distributions except the last example, which is a higher dimensional distribution.

4.3.1 Bernoulli distribution

A Bernoulli random variable is a variable from an experiment that has two possible outcomes: success with probability p; or failure, with probability $(1 - p)$.

Probability function:
$P(\text{success}) = p; P(\text{failure}) = 1 - p$
mean: p
variance: $p(1 - p)$

4.3.2 Multinomial distribution

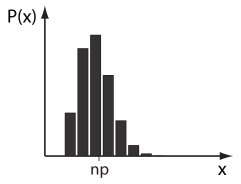
This is the distribution of outcomes in n trials that have k possible outcomes. The probability of each outcome is thereby p_i.

Probability function:
$P(x_i) = n! \prod_{i=1}^{k} (p_i^{x_i} / x_i!)$
mean: np_i
variance: $np_i(1-p_i)$

An important example is the Binomial distribution ($k = 2$), which describes the the number of successes in n Bernoulli trials with probability of success p. Note that the binomial coefficient is defined as

$$\binom{n}{x} = \frac{n!}{x!(n-x)!} \tag{4.11}$$

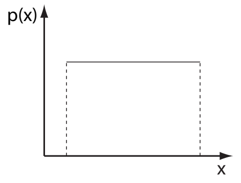
and is given by the MATLAB function `nchoosek`.



Probability function:
$$P(x) = \binom{n}{x} p^x (1-p)^{n-x}$$
mean: np
variance: $np(1-p)$

4.3.3 Uniform distribution

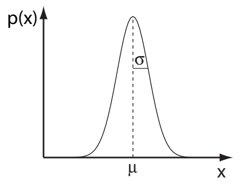
Equally distributed random numbers in the interval $a \leq x \leq b$. Pseudo-random variables with this distribution are often generated by routines in many programming languages.



Probability density function:
$p(x) = \frac{1}{b-a}$
mean: $(a+b)/2$
variance: $(b-a)^2/12$

4.3.4 Normal (Gaussian) distribution

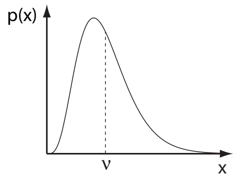
Limit of the binomial distribution for a large number of trials. Depends on two parameters, the mean μ and the standard deviation σ. The importance of the normal distribution stems from the central limit theorem outlined below.



Probability density function:
$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$
mean: μ
variance: σ^2

4.3.5 Chi-square distribution

The sum of the squares of normally distributed random numbers is chi-square distributed and depends on a parameter ν that is equal to the mean. Γ is the gamma function included in MATLAB as `gamma`.



Probability density function:
$p(x) = \frac{x^{(\nu-2)/2}e^{-x/2}}{2^{\nu/2}\Gamma(\nu/2)}$
mean: ν
variance: 2ν

4.3.6 Multivariate Gaussian distribution

We will later consider density functions of a several random variables, $x_1, ..., x_n$. Such density functions are functions in higher dimensions. An important example is the multivariate Gaussian (or Normal) distribution given by

$$p(x_1, ..., x_n) = p(\mathbf{x}) = \frac{1}{(\sqrt(2\pi))^n \sqrt(\det(\mathbf{\Sigma})} \exp(-\frac{1}{2}(\mathbf{x}-\mu)^T\mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)). \tag{4.12}$$

This is a straight forward generalization of the one-dimensional Gaussian distribution mentioned before where the mean is now a vector, μ and the variance generalizes to a covariance matrix, $\mathbf{\Sigma} = [\mathrm{Cov}[X_i, X_j]]_{i=1,2,...,k;j=1,2,...,k}$ which must be symmetric and positive semi-definit. An example with mean $\mu = (1\ 2)^T$ and covariance $\Sigma = (1\ 0.5; 0.5\ 1$ is shown in Fig,4.1.

4.4 Cumulative probability (density) function and the Gaussian error function

The probability of having a value x for the random variable X in the range of $x_1 \leq x \leq x_2$ is given by

$$P(x_1 \leq X \leq x_2) = \int_{x_1}^{x_2} p(x)\mathrm{d}x. \tag{4.13}$$

Note that we have shortened the notation by replacing the notation $P_X(x_1 \leq X \leq x_2)$ by $P(x_1 \leq X \leq x_2)$ to simplify the following expressions. In the main text we often need to calculate the probability that a normally (or Gaussian) distributed variable has values between $x_1 = 0$ and $x_2 = y$. The probability of eqn 4.13 then becomes a function of y. This defines the **Gaussian error function**

$$\frac{1}{\sqrt{2\pi}\sigma}\int_0^y \mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}}\mathrm{d}x = \frac{1}{2}\mathrm{erf}(\frac{y-\mu}{\sqrt{2}\sigma}). \tag{4.14}$$

This Gaussian error function (erf) for normally distributed variables (Gaussian distribution with mean $\mu = 0$ and variance $\sigma = 1$) is commonly tabulated in books on statistics. Programming libraries also frequently include routines that return the values

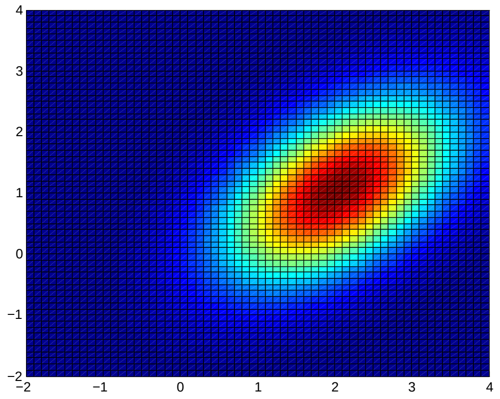


Fig. 4.1 Multivariate Gaussian with mean $\mu = (1\ 2)^T$ and covariance $\Sigma = (1\ 0.5; 0.5\ 1)$.

for specific arguments. In MATLAB this is implemented by the routine `erf`, and values for the inverse of the error function are returned by the routine `erfinv`.

Another special case of eqn 4.13 is when x_1 in the equation is equal to the lowest possible value of the random variable (usually $-\infty$). The integral in eqn 4.13 then corresponds to the probability that a random variable has a value smaller than a certain value, say y. This function of y is called the **cumulative density function** (cdf),[5]

$$P^{\text{cum}}(x < y) = \int_{-\infty}^{y} p(x)\mathrm{d}x, \tag{4.15}$$

[5]Note that this is a probability function, not a density function.

which we will utilize further below.

4.5 Functions of random variables and the central limit theorem

A function of a random variable X,

$$Y = f(X), \tag{4.16}$$

is also a random variable, Y, and we often need to know what the pdf of this new random variable is. Calculating with functions of random variables is a bit different to regular functions and some care has be given in such situations. Let us illustrate how to do this with an example. Say we have an equally distributed random variable X as commonly approximated with pseudo-random number generators on a computer. The probability density function of this variable is given by

$$p_X(x) = \begin{cases} 1 & \text{for } 0 \leq x \leq 1, \\ 0 & \text{otherwise.} \end{cases} \tag{4.17}$$

We are seeking the probability density function $p_Y(y)$ of the random variable

$$Y = \mathrm{e}^{-X^2}. \tag{4.18}$$

The random number Y is **not** Gaussian distributed as we might think naively. To calculate the probability density function we can employ the cumulative density function eqn 4.15 by noting that

$$P(Y \leq y) = P(\mathrm{e}^{-X^2} \leq y) = P(X \geq \sqrt{-\ln y}). \tag{4.19}$$

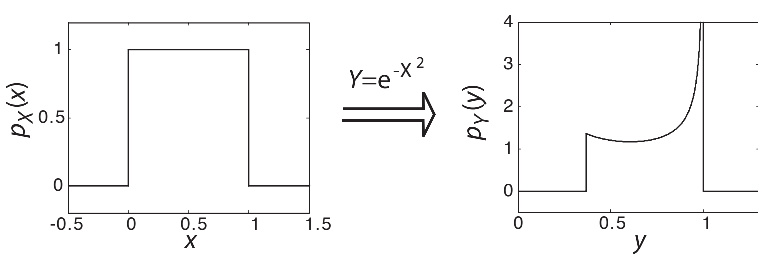
Thus, the cumulative probability function of Y can be calculated from the cumulative probability function of X,

$$P(X \geq \sqrt{-\ln y}) = \begin{cases} \int_{\sqrt{-\ln y}}^{1} p_X(x)\mathrm{d}y = 1 - \sqrt{-\ln y} & \text{for } \mathrm{e}^{-1} \leq y \leq 1, \\ 0 & \text{otherwise.} \end{cases} \tag{4.20}$$

The probability density function of Y is the the derivative of this function,

$$p_Y(y) = \begin{cases} 1 - \sqrt{-\ln y} & \text{for } e^{-1} \leq y \leq 1, \\ 0 & \text{otherwise.} \end{cases} \tag{4.21}$$

The probability density functions of X and Y are shown below.



A special function of random variables, which is of particular interest it can approximate many processes in nature, is the sum of many random variables. For example, such a sum occurs if we calculate averages from measured quantities, that is,

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i, \tag{4.22}$$

and we are interested in the probability density function of such random variables. This function depends, of course, on the specific density function of the random variables X_i. However, there is an important observation summarized in the **central limit theorem**. This theorem states that the average (normalized sum) of n random variables that are drawn from any distribution with mean μ and variance σ is approximately normally distributed with mean μ and variance σ/n for a sufficiently large sample size n. The approximation is, in practice, often very good also for small sample sizes. For example, the normalized sum of only seven uniformly distributed pseudo-random numbers is often used as a pseudo-random number for a normal distribution.

4.6 Measuring the difference between distributions

An important practical consideration is how to measure the similarity of difference between two density functions, say the density function p and the density function q. Note that such a measure is a matter of definition, similar to distance measures of real numbers or functions. However, a proper distance measure, d, should be zero if the items to be compared, a and b, are the same, itÕs value should be positive otherwise, and a distance measure should be symmetrical, meaning that $d(a, b) = d(b, a)$. The following popular measure of similarity between two density functions is not symmetric and is hence not called a distance. It is called **Kulbach–Leibler divergence** and is given by

$$d^{\mathrm{KL}}(p, q) = \int p(x)\log(\frac{p(x)}{q(x)})\mathrm{d}x \tag{4.23}$$

$$= \int p(x)\log(p(x))\mathrm{d}x - \int p(x)\log(q(x))\mathrm{d}x \tag{4.24}$$

This measure is zero if $p = q$ since $log(1) = 0$. This measure is related to the information gain or relative entropy in information theory.

4.7 Density functions of multiple random variables

So far, we have discussed mainly probability (density) functions of single random variables. In many applications, we consider multiple random variables and their interactions, and some basic rules will be very useful. We start by illustrating these with two random variables and mention some generalizations at the end.

We have seen that probability theory is quite handy to model data, and probability theory also considers multiple random variables. The total knowledge about the co-occurance of specific values for two random variables X and Y is captured by the

$$\textbf{joined distribution:} \quad p(x,y) = p(X=x, Y=y). \tag{4.25}$$

This is a two dimensional functions. The two dimensions refers here to the number of variables, although a plot of this function would be a three dimensional plot. An example is shown in Fig.4.2. All the information we can have about a stochastic system is encapsulated in the joined pdf. The slice of this function, given the value of one variable, say y, is the

$$\textbf{conditional distribution}: \quad p(x|y) = p(X=x|Y=y). \tag{4.26}$$

A conditional pdf is also illustrated in Fig.4.2 If we sum over all realizations of y we get the

$$\textbf{marginal distribution}: \quad p(x) = \int p(x,y)dy. \tag{4.27}$$

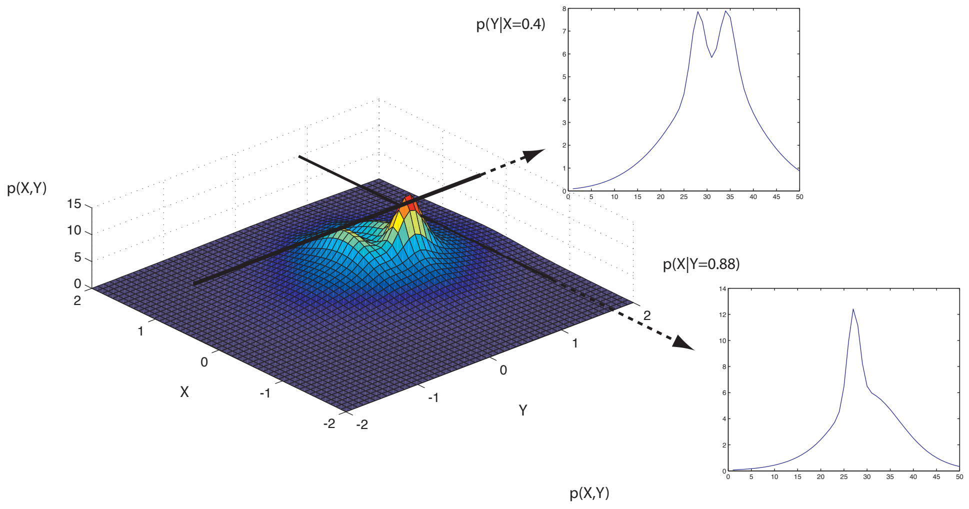


Fig. 4.2 Example of a two-dimensional probability density function (pdf) and some examples of conditional pdfs.

If we know some functional form of the density function or have a parameterized hypothesis of this function, than we can use common statistical methods, such as maximum likelihood estimation, to estimate the parameters as in the one dimensional cases. If we do not have a parameterized hypothesis we need to use other methods, such as treating the problem as discrete and building histograms, to describe the density function of the system. Note that parameter-free estimation is more challenging with increasing dimensions. Considering a simple histogram method to estimate the joined density function where we discretize the space along every dimension into n bins.

This leads to n^2 bins for a two-dimensional histogram, and n^d for a d-dimensional problem. This exponential scaling is a major challenge in practice since we need also considerable data in each bin to sufficiently estimate the probability of each bin.

As mentioned before, if we know the joined distribution of some random variables we can make the most predictions of these variables. However, in practice we have often to estimate these functions, and we can often only estimate conditional density functions. A very useful rule to know is therefore how a joined distribution can be decompose into the product of a conditional and a marginal distribution,

$$p(x, y) = p(x|y)p(y) = p(y|x)p(x), \tag{4.28}$$

which is sometimes called the **chain rule**. Note the two different ways in which we can decompose the joined distribution. This is easily generalized to n random variables by

$$\begin{aligned} p(x_1, x_2, ..., x_n) &= p(x_n|x_1, ...x_{n-1})p(x_1, ..., x_{n-1}) && (4.29)\\ &= p(x_n|x_1, ..., x_{n-1}) * ... * p(x_2|x_1) * p(x_1) && (4.30)\\ &= \Pi_{i=1}^{n} p(x_i|x_{i-1}, ...x_1) && (4.31) \end{aligned}$$

but note that there are also different decompositions possible. We will learn more about this and useful graphical representations in Chapter **??**.

If we divide this chain rule eq. 4.28 by $p(x)$, which is possible as long as $p(x) > 0$, we get the identity

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)}, \tag{4.32}$$

which is called **Bayes theorem**. This theorem is important because it tells us how to combine a **prior** knowledge, such as the expected distribution over a random variable such as the state of a system, $p(x)$, with some evidence called the likelihood function $p(y|x)$, for example by measuring some sensors reading y when controlling the state, to get the **posterior** distribution, $p(y|x)$ from which the new estimation of state can be derived. The marginal distribution $p(y)$, which does not depend on the state X, is the proper normalization so that the left-hand side is again a probability.

Estimations of processes are greatly simplified when random variables are independent. A random variable X is independent of Y if

$$p(x|y) = p(x). \tag{4.33}$$

Using the chain rule eq.4.28, we can write this also as

$$p(x, y) = p(x)p(y), \tag{4.34}$$

that is, the joined distribution of two independent random variables is the product of their marginal distributions. Similar, we can also define conditional independence. For example, two random variables X and Y are conditionally independent of random variable Z if

$$p(x, y|z) = p(x|z)p(y|z). \tag{4.35}$$

Note that total independence does not imply conditionally independence and visa versa, although this might hold true for some specific examples.

Exercises

1. Measure the response function of the light sensor and the ultrasonic sensor with regards to distance. How do the measurements vary in repeated measurments. Discuss briefly your experimental setup and results.
2. Write a Matlab program that produces and bimodal distributed (pseudo-)random variable. Plot a normalized histogram together with an indication of the mean, the median, and the variance. Plot the cumulative distribution for this random variable.
3. (From Thrun, Burgard and Fox, Probabilistic Robotics) A robot uses a sensor that can measure ranges from $0m$ to $3m$. For simplicity, assume that the actual ranges are distributed uniformly in this interval. Unfortunately, the sensors can be faulty. When the sensor is faulty it constantly outputs a range below $1m$, regardless of the actual range in the sensor's measurement cone. We know that the prior probability for a sensor to be faulty is $p = 0.01$.
   Suppose the robot queries its sensors N times, and every single time the measurement value is below $1m$. What is the posterior probability of a sensor fault, for $N = 1, 2, ..., 10$. Formulate the corresponding probabilistic model.

Part II

Supervised learning

5 Regression, classification and maximum likelihood

This chapter start the discussion of an important subset of learning problems, that of supervised learning with labeled data. We start by considering regression with noisy data and formalize our learning objectives. This chapter includes some discussion of fundamental and important algorithms to solve the learning problem as minimization procedure.

5.1 Regression of noisy data

An important type of learning is **supervised learning**. In supervised learning, examples of input-output relations are given, and the goal is to learn the underlying mapping so that accurate predictions can be made for previously unseen data. For example, let us consider health records of 30 employees who were regular members of a company's health club[6]. We want to know the relation between the weight of the persons and their time in a one-mile run. These data are partially shown in a table in Fig.5.1 on plotted on the right of this figure with the Matlab commands
`load healthData; plot(x,y,'*')`. The full data set is provided in file
`healthData.mat`.

Weight (in pounds)	Time of one-mile run (in seconds)
217	481
141	292
152	338
153	357
180	396
.	
.	
.	
245	469
141	252
177	338

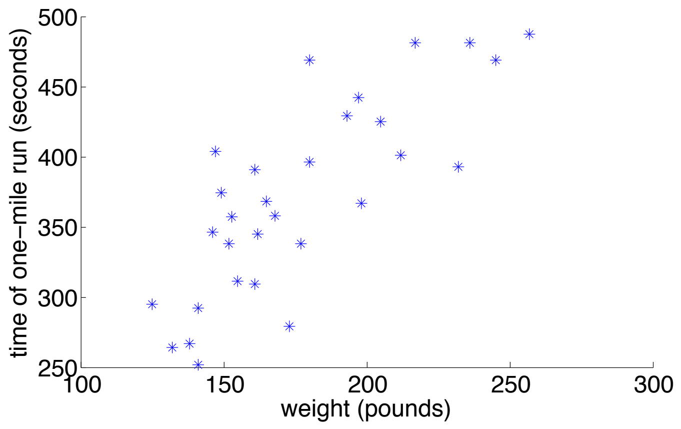


Fig. 5.1 Health data.

[6]Source: Chatterjee, S. and Hadi, A.S. (1988). Sensitivity Analysis in Linear Regression. John Wiley & Sons: New York.

Looking at the plot reveals that there seems to be a systematic relation between the weights and running times, with a trend that heavier persons tend to be slower in the runs, although this is not true for all individuals. Moreover, the trend seems linear. This hypothesis can be quantified as a parameterized function,

$$h(x;\theta) = \theta_0 + \theta_1 x. \tag{5.1}$$

This notation means that the hypothesis h is a function of the quantity x, and the hypothesis includes all possible straight lines, where each line can have a different offset θ_0 (intercept with the y-axis), and slope θ_1. We typically collect parameters in a **parameter vector** θ. We only considered on input variable x above, but we can easily generalize this to higher dimensional problems where more input **attributes** are given. For example, there might be the amount of exercise each week that might impact the results of running times. If we make the hypothesis that this additional variable has also a linear influence on the running time, independently of the other attribute, we can write the hypothesis as

$$h(\mathbf{x};\theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_2. \tag{5.2}$$

A useful trick to enable a compact notation in higher dimensions with n attributes is to introduce $x_0 = 1$. We can then write the linear equations as

$$h(\mathbf{x};\theta) = \theta_0 x_0 + ... + \theta_n x_n = \sum_i \theta_i x_i = \theta^T \mathbf{x}. \tag{5.3}$$

The vector θ^T is the transpose of the vector θ.

In the following we consider m training examples, the pairs of values

$$\{(x^{(i)}, y^{(i)}); i = 1...m\}, \tag{5.4}$$

which is also called a **training set**. We use here again an index counter i, though this is different than the index over the attributes used before. To distinguish this better, we write the index of training examples as superscript and use brackets around it to distinguish it from a potentiation operation.

We can use the training examples to come-up with some reasonable values for the parameters. For example, a common technique is called **least-mean-square (LMS) linear regression** to determine those parameters. In this technique we chose values for θ that will minimize the sum of the **mean square error** (MSE) from the line to each data point. In general we consider a **cost function** to be minimized, which is in this case the square function

$$E(\theta) = \frac{1}{2}(y - h(x;\theta))^2 \tag{5.5}$$

$$\approx \frac{1}{2m}\sum_i (y^{(i)} - h(x^{(i)};\theta))^2. \tag{5.6}$$

Note that the objective function is a function of the parameters. Also, there is a small subtlety in the above equation since we wrote the general form of the objective function in line 5.5 and considered its approximation with the data, considered to be independent, in line 5.6. With this objective function, we reduced the learning problem to a search problem of finding the parameter values that minimize this objective function,

$$\theta = \arg\min_{\theta} E(\theta) \tag{5.7}$$

We will demonstrate practical solutions to this search problem with three important methods. The first method is an analytical one. You might remember from basic mathematics classes that finding a minimum of a function $f(x)$ is characterized by $\frac{\mathrm{d}f}{\mathrm{d}x} = 0$, and $\frac{\mathrm{d}^2 f}{\mathrm{d}x^2} > 0$. Here we have a vector function since the cost function depends an several parameters. The derivative then becomes a gradient

$$\nabla_\theta E(\theta) = \begin{pmatrix} \frac{\partial E}{\partial \theta_0} \\ . \\ . \\ . \\ \frac{\partial E}{\partial \theta_n} \end{pmatrix}. \tag{5.8}$$

It is useful to collect the training data in a large matrix for the x values, and a vector for the y values,

$$X = \begin{pmatrix} \mathbf{x}^{(1)T} \\ . \\ . \\ . \\ \mathbf{x}^{(m)T} \end{pmatrix} \qquad Y = \begin{pmatrix} y^{(1)T} \\ . \\ . \\ . \\ y^{(m)T} \end{pmatrix}. \tag{5.9}$$

We can then write the cost function in as

$$E(\theta) = \frac{1}{2m}(Y - X\theta)^T(Y - X\theta), \tag{5.10}$$

The parameters for which the gradient is zero is then given by the **normal equation**

$$\theta = (X^T X)^{-1} X^T Y. \tag{5.11}$$

We have still to make sure these parameters are the minimum and and a maximum value, but this can easily be done and is also obvious when plotting the result.

A second method we want to consider is random search. This is a very simple algorithm, but worthwhile considering to compare them to the other solutions. In this algorithm, a new random values for the parameters are tried, and these new parameters replace the old ones if the new values result in a smaller error value (see Matlab code in Tab.5.2).

5.2 Gradient Descent

The last method discussed here for finding a minimum for LMS regression is **Gradient Descent**. This method will often be used in the following and it will thus be reviewed here in more detail.

Table 5.1 Program randomSearch.m

```
%% Linear regression with random search
clear; clf; hold on;

load healthData;
E=1000000;
for trial=1:100
    thetaNew=[100*rand()+50, 3*rand()];
    Enew=0.5*sum((y-(thetaNew(1)+thetaNew(2)*x)).^2);
    if Enew<E; E=Enew; theta=thetaNew; end
end

plot(x,y,'*')
plot(120:260,theta(1)+theta(2)*(120:260))
xlabel('weight (pounds)')
ylabel('time of one-mile run (seconds)')
```

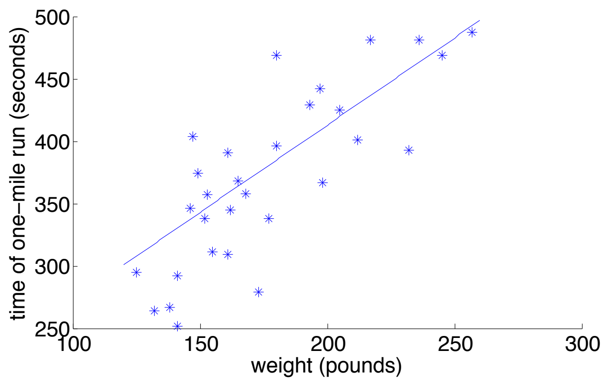


Fig. 5.2 Health data with linear least-mean-square (LMS) regression from random search.

Gradient Descent starts at some initial value for the parameters and improves our objective (lowers the cost) iteratively by making changes to the parameters along the negative gradient of the cost function,

$$\theta \leftarrow \theta - \alpha \nabla_\theta E(\theta). \tag{5.12}$$

The constant α is called a **learning rate**. The principle idea behind this method is illustrated for a general cost function with one parameter in Fig.5.3.

The gradient is simple the slope (local derivative) for a function with one variable, but with functions in higher dimensions (more variables), the gradient is the local slope along the direction of the steepest descent. For large gradients, this method takes large steps, whereas the effective step-width becomes smaller near a minimum. The Gradient Descent works often well for local optimization, but it can get stuck in local

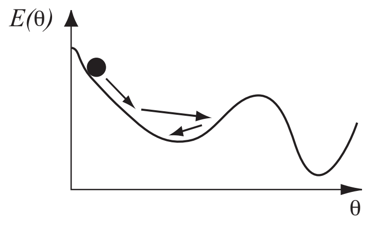


Fig. 5.3 Illustration of error minimization with a gradient descent method on a one-dimensional error surface $E(\theta)$.

minima. In case of the MSE with a linear regression function, the update rule for the two parameters θ_0 and θ_1 can easily be calculated:

$$h(x^{(i)};\theta) = \theta_0 + \theta_1 x^{(i)} \tag{5.13}$$

$$\theta_0 \leftarrow \theta_0 + \alpha \frac{\partial E(\theta)}{\partial \theta_0} \tag{5.14}$$

$$\theta_1 \leftarrow \theta_1 + \alpha \frac{\partial E(\theta)}{\partial \theta_1} \tag{5.15}$$

$$E(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (y^{(i)} - h(x^{(i)};\theta))^2 \tag{5.16}$$

$$\frac{\partial E(\theta)}{\partial \theta_0} = \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - \theta_0 - \theta_1 x^{(i)})(-1) \tag{5.17}$$

$$\frac{\partial E(\theta)}{\partial \theta_1} = \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - \theta_0 - \theta_1 x^{(i)})(-x^{(i)}), \tag{5.18}$$

which lead to the final rule:

$$\theta_0 \leftarrow \theta_0 + \frac{\alpha}{m} \sum_{i=1}^{m} (y_i - \theta_0 - \theta_1 x_i) \tag{5.19}$$

$$\theta_1 \leftarrow \theta_1 + \frac{\alpha}{m} \sum_{i=1}^{m} (y_i - \theta_0 - \theta_1 x_i) x_i. \tag{5.20}$$

Note that the learning rate α has to be chosen small enough for the algorithm to converge. An example is show in Fig.5.4, where the dashed line shows the initial hypothesis, and the solid line the solution after 100 updates.

In the algorithm above we calculate the average gradient over all examples before updating the parameters. This is called a **batch algorithm** or **synchronous update** since the whole batch of training data is used for each updating step and the update is only made after seeing all training data. This might be problematic in some applications as the training examples have to be stored somewhere and have to be recalled continuously. A much more applicable methods, also thought to be more biological realistic, is to use each training example when it comes in and disregards it right after.

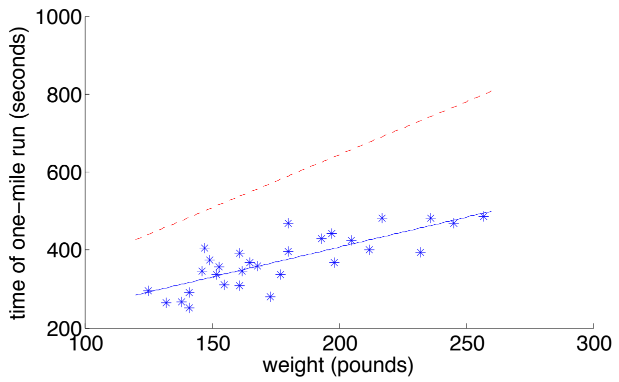
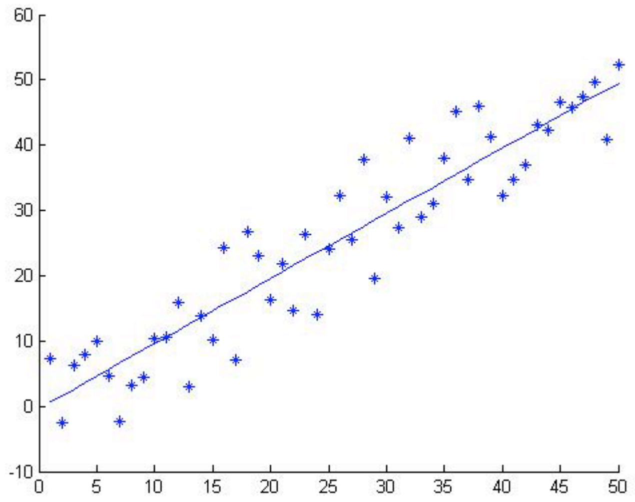


Fig. 5.4 Health data with linear least-mean-square (LMS) regression with gradient descent. The dashed line shows the initial hypothesis, and the solid line the solution after 100 updates.

In this way we do not have to store all the data. Such algorithms are called **online algorithms** or **asynchronous update**. Specifically, in the example above, we calculate the change for each training examples and update the parameters for this training example before moving to the next example. While we might have to run through the short list of training examples in this specific example, it can still be considered online since we need only one training example at each training step and a list could be supplied to us externally. There are also variations of this algorithms depending on the order we use the training examples (e.g. random or sequential), although this should not be crucial in for the examples discussed here.

5.2.1 LinearRegressionExampleCode

```
%% Linear regression with gradient descent
clear all; clc; hold on;

load SampleRegressionData; m=50; alpha=0.001;
theta1=rand*100*((rand<0.5)*2-1);
theta2=rand*100*((rand<0.5)*2-1);

for trial=1:50000
    sum1 = sum(y - theta1 - theta2 *x);
    sum2 = sum((y - theta1 - theta2 *x).* x);
    theta1 = theta1 + (2*alpha/m) * sum1;
    theta2 = theta2 + (2*alpha/m) * sum2;
    sum1 = 0; sum2 = 0;
end

plot(x, y, '*');
plot(x, x*theta2+theta1);
hold off;
```



A final remark: We have used in this section the popular cost function MSE to calculate the regression in the above example, but it is interesting to think about this choice a bit more. If we change this function, then we would value outliers in a different way. The MSE is a quadratic function and does therefore weight heavily large deviations from the regression curve. An alternative approach would be to say that these might be outliers to which we should not give so much weight. In this case we might want that the increase of the error value decrease with large distances, for example by using the logarithm function $log(|y - h|)$ as cost (error) function. So, the MSE is certainly not the only choice. We will see below that there is a natural choice for cost functions if we take the stochastic nature of the data into account.

5.3 Calibrating state estimator with linear regression

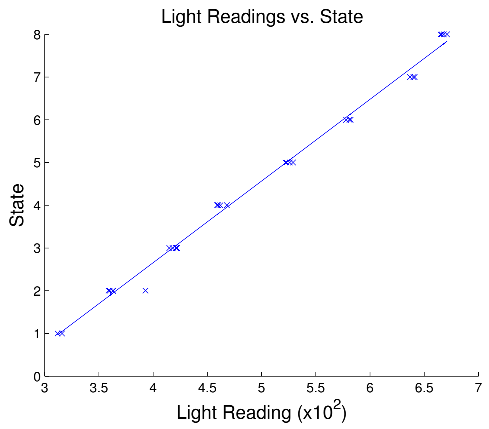
One of the basic requirements for a robot or agent is to estimate the state it is in such as its position in space. In this tutorial we will use the color sensor and a floor with different shades of gray to help the robot to determine its position on a surface. Specifically, we use a sheet of gray bars of various gradients that is given in file `StateSheet.pdf`. The NXT can read this model as each gray bar will give a different light reading when the its light sensor is held over it. However, we need to pair each light reading with a number which represents the state value. Of course, the reading of the light sensors will fluctuate somewhat, we would like to repeat the measurements and find a function that translates each light reading to a state value.

To set up the tribot for this experiment, do the following:

1. Mount a switch sensors on the NXT.
2. Mount light sensor on front of the NXT, facing downward.
3. Because the state space used in this tutorial only involves driving forward and backward, it is a good idea to use additional lego pieces to lock the back wheel

so it stays straight. If it is left to freely rotate, the NXT will start to turn and travel off the state sheet which will interfere with results.

4. Tape the state sheet to a table or other flat surface.
5. Setup an NXT motor object that will drive the robot foward in a straight line. Power should be between 20-30, TachoLimit around 20; motors should brake once it reaches the TachoLimit.
6. Create two matricies, one for light readings (from the NXT's light sensor), and the other for states. They must be the same size.
7. Have the NXT move forward, take a light reading from its sensor, and store it in the light reading matrix. The user should then be prompted to input which state the NXT is in. Store this value in the state matrix. Repeat this until you have a few readings from every state on the state sheet.
8. Using one of the methods discussed above, use linear regression to find a linear relationship between the light reading matrix and its associated state.



9. Test the data. Have the NXT take in a light sensor reading, and using the regressed line, interpolate the state and a round function to round the value to the nearest whole number). Does the regressed value give the correct state?

The state sheet will be used in chapter **??** when experimenting with reinforcement learning.

5.4 Probabilistic models and maximum likelihood

So far, we have only modelled the mean trend of the data, and we should investigate more the fluctuations around this mean trend. Fig.5.5 is a plot of the histogram of the differences between the actual data and the hypothesis regression line. This look a bit Gaussian, which is likely to be a common finding though not necessarily the only possible. With this additional conjecture, we should revise our hypothesis. More precisely, we acknowledge that the data are noisy and that we can only give a probability of finding certain values. Specifically, we assume here that the data follow a certain trend $h(\mathbf{x};\theta$ with **additive noise**, η,

$$p(y|x;\theta) = h(\mathbf{x};\theta) + \eta, \tag{5.21}$$

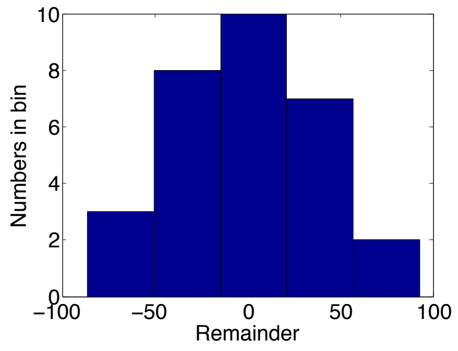


Fig. 5.5 Histogram of the difference between the data points and the fitted hypothesis, $(y_i - \theta_1 - \theta_2 x_i)$.

where the random variable η is Gaussian distributed in the example above,

$$p(\eta) = \aleph(\mu, \sigma) \tag{5.22}$$

We can then also write the probabilistic hypothesis in the example as a Gaussian model of the data distributed with a mean that depends on the variable x,

$$p(y|x;\theta) = \aleph(\mu = h(x), \sigma) \tag{5.23}$$
$$= \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{-(y - \theta^T \mathbf{x})^2}{2\sigma^2}\right) \tag{5.24}$$

This functions specifies the probability of values for y, given an input x and the parameters θ.

Specifying a model with a density function is an important step in modern modelling and machine learning. In this type of thinking, we treat data from the outset as fundamentally stochastic, that is, data can be different even in situations that we deem identical. This randomness may come **irreducible indeterminacy**, but might also represent **epistemological limitations** such as the lack of knowledge of hidden processes or limitations in observing states directly. The language of probability theory has helped to make large progress the machine learning area.

While we have made a parameterized hypothesis underlying the nature of data, we need to estimate values for the parameters to make real predictions. We therefore consider again the examples for the input-output pairs, our training set $\{(x^{(i)}, y^{(i)}); i = 1...m\}$. The important principle that we will now follow is to choose the parameters so that the examples we have are most likely. This is called **maximum likelihood estimation**. To formalize this principle, we need to think about how to combine probabilities for several observations. If the observations are independent, then the joined probability of several observations is the product of the individual probabilities,

$$p(y_1, y_2, ..., y_m | x_1, x_2, ..., x_m; \theta) = \Pi_i^m p(y_i | x_i; \theta). \tag{5.25}$$

Note that y_i are still random variables in the above formula. We now use our training examples as specific observations for each of these random variables, and introduce the **Likelihood function**

$$L(\theta) = \Pi_i^m p(\theta; y^{(i)}, x^{(i)}). \tag{5.26}$$

The p on the right hand side is now not a density function, but it is a regular function (with the same form as our parameterized hypothesis) of the parameters θ for the given values $y^{(i)}$ and $x^{(i)}$. Instead of evaluating this large product, it is common to use the logarithm of the likelihood function, so that we can use the sum over the training examples,

$$l(\theta) = \log L(\theta) = \sum_i^m \log(p(\theta; y^{(i)}, x^{(i)})). \tag{5.27}$$

Since the log function strictly monotonly rising, the maximum of L is also the maximum of l. The maximum (log-)likelihood can thus be calculated from the examples as

$$\theta^{\text{MLE}} = arg \max_\theta l(\theta). \tag{5.28}$$

We might be able to calculate this analytically or use one of the search algorithms to find a minimum from this function.

Let us apply this to the linear regression discussed above. The log-likelihood function for this example is

$$\begin{aligned} l(\theta) &= \log \Pi_{i=1}^m \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{-(y^{(i)} - \theta^T \mathbf{x}^{(\mathbf{i})})^2}{2\sigma^2}\right) & (5.29)\\ &= \sum_{i=1}^m \left(\log \frac{1}{\sqrt{2\pi}\sigma} - \frac{-(y^{(i)} - \theta^T \mathbf{x}^{(\mathbf{i})})^2}{2\sigma^2}\right) & (5.30)\\ &= -\frac{m}{2} \log 2\pi\sigma - \sum_{i=1}^m \frac{-(y^{(i)} - \theta^T \mathbf{x}^{(i)})^2}{2\sigma^2}. & (5.31) \end{aligned}$$

Thus, the log was chosen so that we can use the sum in the estimate instead of dealing with big numbers based on the product of the examples.

Since the first term in the expression 5.31, $-\frac{m}{2} \log 2\pi\sigma$, is independent of θ, and since we considered here a model with a constant, σ^2, for the variance of the data, maximizing the log-likelihood function is equivalent to minimizing a quadratic error term

$$E = \frac{1}{2}(y - h((x; (\theta))^2 \Longleftrightarrow p(y|\mathbf{x}; \theta) = \frac{1}{\sqrt{2\pi}} \exp(-\frac{((y - h((x; \theta)^2}{2}) \tag{5.32}$$

This was the cost function chosen in the first section. We have now a deeper understanding of the choice of a cost function. With a probabilistic model our principle should be maximum likelihood estimation, which determines the cost function in the previous schemes. While this was equivalent in the case studied here, it would be different if we consider also the variance as a free parameter, which leads to χ^2 fitting procedures in the literature.

We have discussed Gaussian distributed data in most of this section, but one can similarly find correspondences error functions for other distributions. For example, a **polynomial error function** correspond more generally to a density model of the form

$$E = \frac{1}{p}||y - h((x; (\theta)||^p \Longleftrightarrow p(y|\mathbf{x}; \theta) = \frac{1}{2\Gamma(1/p)} \exp(-||y - h((x; \theta||^p). \quad (5.33)$$

Later we will specifically discuss and use the **ϵ-insensitive error function**, where errors less than a constant ϵ do not contribute to the error measure, only errors above this value,

$$E = ||y - h((x; (\theta)||_\epsilon \Longleftrightarrow p(y|\mathbf{x}; \theta) = \frac{p}{2(1-\epsilon)} \exp(-||y - h((x; \theta||_\epsilon). \quad (5.34)$$

Since we already acknowledged that we do expect that data are noisy, it is somewhat logical to not count some deviations form the expectation as errors. It also turns out that this error function is much more robust than other erro functions.

5.5 Maximum a posteriori estimates

In the maximum likelihood estimation we assumed that we have no prior knowledge of the parameters θ. However, we sometimes might know which values of the parameters are impossible or less likely. This prior knowledge can be summarized in the prior distribution $p(\theta)$, and the next question is how to combine this prior knowledge in the maximum likelihood scheme. Combining prior knowledge with some evidence is described by Bayes' theorem. Thus, let us consider again that we have some observations (x, y) from specific realizations of the parameters, which is given by $(p(x, y|\theta)$, and the prior about the possible values of the parameters, given by $p(\theta)$. The prior is in this situation sometimes called the **regularizer**, restricting possible values in a specific domain. We want to know the distribution of parameters given the observation, $p(\theta|x, y)$, which can be calculated from Bayes's theorem,

$$p(\theta|x, y) = \frac{p(x, y|\theta)p(\theta)}{\int_{\theta \in \Theta} p(x, y|\theta)p(\theta)\mathrm{d}\theta}, \quad (5.35)$$

where Θ is the domain of the possible parameter values. We can now use this expression to estimate the most likely values for the parameters. For this we should notice that the denominator, which is called the **partition function**, does not depend on the parameters θ. The most likely values for the parameters can thus be calculated without this term and is given by the **maximum a posteriori (MAP)** estimate,

$$\theta^{\mathrm{MAP}} = arg \max_\theta p(x, y|\theta)p(\theta). \quad (5.36)$$

This is, in a Bayesian sense, the most likely value for the parameters, where, of course, we now treat the probability function as a function of the parameters (e.g., a likelihood function).

A final caution: ML and MAP estimates give us a **point estimate**, a single answer of the most likely values of the parameters. This is often useful as a first guess and is commonly used to **make decisions** about which actions to take. However, it is possible that other sets of parameters values might have only a little smaller likelihood value, and should therefore also be considered. Thus, one limit of the estimation methods discussed here is that they do not take distribution of answers into account, which is more common in more advanced Bayesian methods.

5.6 Non-linear regression and the bias-variance tradeoff

So far, we have always discussed linear regression which assume that there is a linear relation between the variables. This is of course not the only possible relations between data. In Fig.5.6A we plotted the number of transistors of microprocessors against the year the processors were introduced. The plot also includes a linear fit, suggesting that we should assume some other functions.

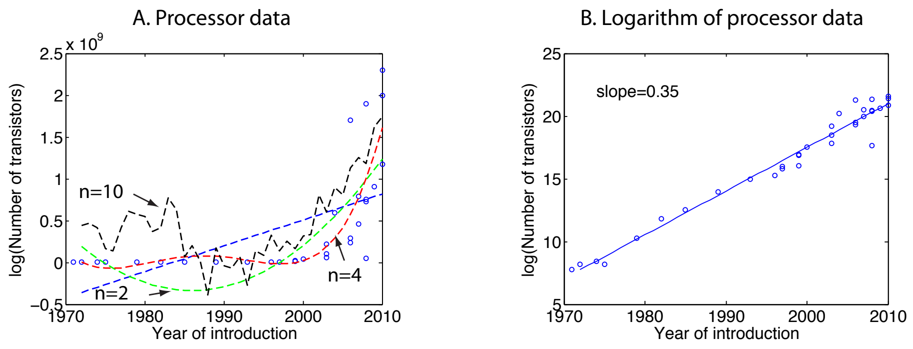


Fig. 5.6 Data from showing the number of transistors in microprocessors plotted the year that the processor was introduced. (A) Data and some linear and polynomial fits of the data. (B) Logarithm of the data and linear fit of these data.

Finding the right function is one of the most difficult tasks, and there is not a simple algorithm that can give us the answer. This task is therefore an important area where experience, a good understanding of the problem domain, and a good understanding of scientific methods are required. This section does therefore try to give some recommendations when generalizing regression to the non-linear domain. These comments are important to understand for applying machine learning techniques in general.

It is often a good idea to visualize data an various ways since the human mind is often quite advanced in 'seeing' trends and patterns. Domain-knowledge thereby very valuable as specialists in the area from which the data are collected can give important advice of or they might have specific hypothesis that can be investigated. It is also good to know common mechanisms that might influence processes. For example, the rate of change in basic growth processes is often proportional to the size of the system itself. Such an situation lead to exponential growth. (Think about why this is the case). Such situations can be revealed by plotting the functions on a logarithmic scale or the logarithm of the function as shown in Fig.5.6B. A linear fit of the logarithmic values is also shown, confirming that the average growth of the number of transistors in microprocessors is exponential.

But how about more general functions. For example, we can consider a polynomial of order n, that can be written as

$$y = \theta_0 x^0 + \theta_1 x^1 + \theta_2 x^2 + ... + \theta_n x^n \tag{5.37}$$

We can again use LMS regression to determine the parameters from the data by minizing the LMS error function between the hypothesis and the data. This is implemented in Matlab as function `polyfit(x,y,n0)`. The LMS regression of the transistor data to a polynomials for orders $n = 2, 4, 10$ are shown in Fig.**??**A as dashed lines.

A major question when fitting data with fairly general non-linear functions is the order that we should consider. The polynomial of order $n = 4$ seem to somewhat fit the data. However, notice there are systematic deviations between the curve and the data points. For example, all the data between years 1980 and 1995 are below the fitted curve, while earlier data are all above the fitted curve. Such a systematic **bias** is typical when the order of the model is too low. However when we increase the order, then we usually get large fluctuations, or **variance**, in the curves. This fact is also called **overfitting** the data since we have typically too many parameters compared to the number of data points so that our model starts describing individual data points with their fluctuations that we earlier assumed to be due to some noise in the system. This difficulty to find the right balance between these two effects is also called the **bias-variance tradeoff**.

The bias-variance tradeoff is quite important in practical applications of machine learning because the complexity of the underlying problem is often not know. It then becomes quite important to study the performance of the learned solutions in some detail. For this it is useful to split the data set into **training set**, which is used to estimate the parameters of the model, and a **validation set** that can be used to study the performance on data that have not been used in the training procedure, that is, how the machine performs in **generalizes**. A schematic figure showing the bias-variance tradeoff is shown in Fig.5.7. The plot shows the error rate as evaluated by the training data (dashed line) and validation curve (solid line) when considering models with different complexities. When the model complexity is lower than the true complexity of the problem, then it is common to have a large error both in the training set and in the evaluation due to some systematic bias. In the case when the complexity of the model is larger than the generative model of the data, then it is common to have a small error on the training data but a large error on the generalization data since the predictions are becoming too much focused on the individual examples. Thus varying the complexity of the data, and performing experiments such as training the system on different number of data sets or for different training parameters or iterations can reveal some of the problems of the models.

Using some of the data to validate the learning model is essential for many machine learning methods. A important question is then how many data we should keep to validate versus train the model. If we use too many data for validation, than we might have too less data for accurate learning in the first place. On the other end, if we have to few data for validation than this might not be very representative. In practice we are often using some **cross-validation** techniques to minimize the trade-off. That is, we use the majority of the data for training, but we repeat the selection of the validation data several times to make sure that the validation was not just a result of outliers. The repeated division of the data into a training set and validation set can be done in different ways. For example, in **random subsampling** we just use random subsample for each set and repeat the procedure with other random samples. More common is k**-fold cross-validation**. In this technique we divide the data set into k-subsamples

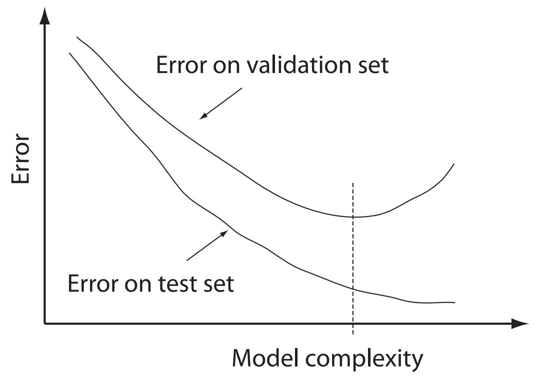


Fig. 5.7 Illustration of bias-variance tradeoff.

samples and use $k-1$ subsamples for training and one subsample for validation. In the next round we use another subsample for validating the training. A common choice for the number of subsamples is $k = 10$. By combining the results for the different runs we can often reduce the variance of our prediction while utilizing most data for learning.

We can sometimes even help the learning process further. In many learning examples it turns out that some data are easy to learn while others are much harder. In some techniques called **boosting**, data which are hard to learn are over-sampled in the learning set so that the learning machine has more opportunities to learn these examples. A popular implementation of such an algorithm is **AdaBoost** (adaptive Boosting).

5.7 Classification and logistic regression

An important special case of learning problems is **classification** in which features are mapped to a finite number of possible categories. We discuss in this section **binary classification**, which is the case of two target classes, and we view the problem here as a special case of non-linear regression in which the target function (y-values) has only two possible values such as 0 and 1.

Let us first consider a random number which takes the value of 1 with probability ϕ and the value 0 with probability $1 - \phi$ (the probability of being either of the two choices has to be 1.) Such a random variable is called Bernoulli distributed. Tossing a coin is a good example of a process that generates a Bernolli random variable and we can use maximum likelihood estimation to estimate the parameter ϕ from such trials. That is, let us consider m tosses in which h heads have been found. The log-likelihood of having h heads ($y = 1$) and $1 - h$ tails ($y = 0$) is

$$l(\phi) = \log(\phi^h(1-\phi)^{m-h}) \tag{5.38}$$

$$= h\log(\phi) + (m-h)\log(1-\phi). \tag{5.39}$$

To find the maximum with respect to ϕ we set the derivative of l to zero,

$$\frac{\mathrm{d}l}{\mathrm{d}\phi} = \frac{h}{\phi} - \frac{m-h}{1-\phi} \tag{5.40}$$

$$= \frac{h}{\phi} - \frac{m-h}{1-\phi} \tag{5.41}$$
$$= 0 \tag{5.42}$$
$$\rightarrow \quad \phi = \frac{h}{m} \tag{5.43}$$

As you might have expected, the maximum likelihood estimate of the parameter ϕ is the fraction of heads in m trials.

Now let us discuss the case when the probability of observing a head or tail, the parameter ϕ, depends on an attribute x, as usual in a stochastic (noisy) way. An example is illustrated in Fig.5.8 with 100 examples plotted with star symbols. The data suggest that it is far more likely that the class is $y = 0$ for small values of x and that the class is $y = 1$ for large values of x, and the probabilities are more similar inbetween. We put forward the hypothesis that the transition between the low and high probability region is smooth and qualify this hypothesis as parameterized density function known as a **logistic** (sigmoidal) function

$$p(y=1) = \frac{1}{1+\exp(-\theta^T \mathbf{x})}. \tag{5.44}$$

As before, we can then treat this density function as function of the parameters θ for the given data values (likelihood function), and use maximum likelihood estimation to estimate values for the parameters so that the data are most likely. The density function with sigmoidal offset $\theta_0 = 2$ and slope $\theta_1 = 4$ is plotted as solid line in Fig.5.8.

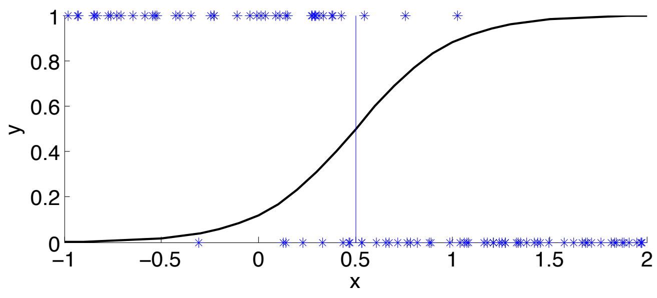


Fig. 5.8 Binary random numbers (stars) drawn from the density $p(y=1) = \frac{1}{1+\exp(-\theta_1 x - \theta_0)}$ (solid line).

How can we use the knowledge (estimate) of the density function to do classification? The obvious choice is to predict the class with the higher probability, given the input attribute. This **bayesian decision point**, x^t, or **dividing hyperplane** in higher dimensions, is give by

$$p(y=1|x^t) = p(y=0|x^t) = 0.5 \rightarrow x^t \theta^T \mathbf{x^t} = 0. \tag{5.45}$$

We have here considered binary classification with linear decision boundaries as logistic regression, and we can also generalize this method to problems with non-linear decision boundaries by considering hypothesis with different functional forms of the decision boundary. However, coming up with specific functions for boundaries is often difficult in practice, and we will discuss much more practical methods for binary classification later in this course.

5.8 Exercise

1. Compare the analytic solution to the numerical solutions of random search and gradient descent search in the linear regression of the health data in section 5.1.
2. Is a linear fit of the logarithm of a function equal to an exponential fit of the original function? Discuss.
3. Explain briefly how a binary classification method can be used to support multiclass classification of more than two classes.

6 Generative models

In the previous chapter we have introduced the idea that understanding the world should be based on a model of the world in a probabilistic sense. That is, building knowledge about the world really means estimating a large density function about the world. So far we have used such stochastic model mainly for a recognition model that take feature values $\mathbf{x}$ and make a prediction of an output y. Given the stochastic nature of the systems we want to model, the models where formulated as parameterized functions that represent the conditional probability $p(y|\mathbf{x};\theta)$. Of course, learning such models is a big task. Indeed, we had to assume that we know already the principle form of the distribution, and we used only simple model with low-dimensional feature vectors. The learning tasks of humans to be able to function in the real world seems much more daunting, and even training a robot in more restricted environment seems still beyond our current ability. While the previous models illustrate the principle problem in supervised learning, much of the rest of this course discusses more practical methods.

At the end of the last chapter we discussed a classification task where the aim of the model was to discriminate between classes based on the feature values. Such models are called **discriminative models** because they try to discriminate between possible outcomes based on the input values. Building a discriminative model directly from example data can be a daunting task as we have to learn how each item is distinguished from every other possible item. A different strategy, which seems much more resembling human learning, is to learn first about the nature of specific classes and then use this knowledge when faced with a classification task. For example, we might first learn about chairs, and independently about tables, and when we are shown pictures with different furnitures we can draw on our knowledge to classify them. Thus, in this chapter we start discussing **generative models** of individual classes, given by $p(\mathbf{x}|y;\theta)$.

Generative models can be useful in its own right, and are also important to guide learning as discussed later, but for now we are mainly interested in using these models for classification. Thus, we need to ask how we can combine the knowledge about the different classes to do classification. Of course, the answer is provided by Bayes' theorem. In order to make a discriminative model from the generative models, we need to the **class priors know**, e.g. what the relative frequencies of the classes is, and can then calculate the probability that an item with features $\mathbf{x}$ belong to a class y as

$$p(y|\mathbf{x};\theta) = \frac{p(\mathbf{x}|y;\theta)p(y)}{p(x)}. \tag{6.1}$$

We can use this directly in the case of classification. The Bayesian decision criterion of predicting the class with the largest posterior probability is then:

$$\arg\max_y p(y|\mathbf{x};\theta) = \arg\max_y \frac{p(\mathbf{x}|y;\theta)p(y)}{p(x)} \tag{6.2}$$

$$= \arg\max_y p(\mathbf{x}|y;\theta)p(y), \tag{6.3}$$

where we have used the fact that the denominator does not depend on y and can hence be ignores. In the case of binary classification, this reads:

$$\arg\max_y p(y|\mathbf{x};\theta) = \arg\max_y (p(\mathbf{x}|y=0;\theta)p(y=0) + p(\mathbf{x}|y=1;\theta)p(y=1). \tag{6.4}$$

While using generative models for classification seem to be much more elaborate, we will see later that there are several arguments which make generative models attractive for machine learning, and we will arue that generative models are do more closely resemble human brain processing principles.

6.1 Discriminant analysis

We will now discuss some common examples of using generative models in classification. The methods in this section go back to a paper by R. Fisher in 1936. In the following examples we consider that there are k classes, and we first assume that each class has members which are Gaussian distribution over the n feature value. An example for $n = 2$ is shown in Fig.**??**A.

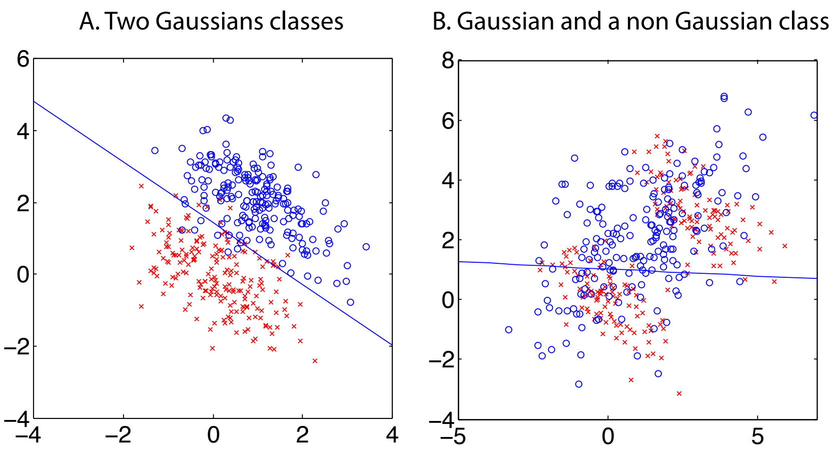


Fig. 6.1 Linear Discriminant analysis on a two class problem with different class distributions.

Each of the classes have a certain class prior

$$p(y = k) = \phi_k \tag{6.5}$$

, and each class itself is multivariate Gaussian distributed, generally with different means, μ_k and variances, Σ_k,

$$p(\mathbf{x}|y = k) = \frac{1}{\sqrt{2\pi}^n \sqrt{|\Sigma_0|}} e^{-\frac{1}{2}(\mathbf{x}-\mu_k)^T \Sigma_k^{-1}(\mathbf{x}-\mu_k)} \tag{6.6}$$

(6.7)

Since we have supervised data with examples for each class, we can use maximum likelihood estimation to estimate the most likely values for the parameters $\theta = (\phi_k, \mu_k, \Sigma_k)$. For the class priors, this is simply the relative frequency of the training data,

$$\phi_k = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}(y^{(i)} = k) \tag{6.8}$$

where the function $\mathbb{1}(y^{(i)} = k) = 1$ if the ith example belongs to class k, and $\mathbb{1}(y^{(i)} = k) = 0$ otherwise. The estimates of the means and variances within each class are given by

$$\mu_k = \frac{\sum_{i=1}^{m} \mathbb{1}(y^{(i)} = k)\mathbf{x}^{(i)}}{\sum_{i=1}^{m} \mathbb{1}(y^{(i)} = k)} \tag{6.9}$$

$$\Sigma_k = \frac{\sum_{i=1}^{m} \mathbb{1}(y^{(i)} = k)(x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T}{\sum_{i=1}^{m} \mathbb{1}(y^{(i)} = k)}. \tag{6.10}$$

With these estimates, we can calculate the optimal (in a Bayesian sense) decision rule, $G(x;\theta)$, as a function of $\mathbf{x}$ with parameters θ, namely

$$\begin{aligned}
G(x) &= arg\max_k p(y = k|\mathbf{x}) &(6.11)\\
&= arg\max_k [p(\mathbf{x}|y = k; \theta)p(y = k)] &(6.12)\\
&= arg\max_k [log(p(\mathbf{x}|y = k; \theta)p(y = k))] &(6.13)\\
&= arg\max_k [-log(\sqrt{2\pi}^n \sqrt{|\Sigma_0|}) - \frac{1}{2}(\mathbf{x} - \mu_k)^T \Sigma_k^{-1}(\mathbf{x} - \mu_k) + log(\phi_k)] &(6.14)\\
&= arg\max_k [-\frac{1}{2}\mathbf{x}^T \Sigma_k^{-1}\mathbf{x} - \frac{1}{2}\mu_k^T \Sigma_k^{-1}\mu_k + \mathbf{x}^T \Sigma_k^{-1}\mu_k + log(\phi_k)], &(6.15)
\end{aligned}$$

since the first term in equation 6.14 does not depend on k and we can multiply out the other terms. With the maximum likelihood estimates of the parameters, we have all we need to make this decision.

In order to calculate the decision boundary between classes l and k, we make the common additional assumption that the covariance matrices of the classes are the same,

$$\Sigma_k =: \Sigma. \tag{6.16}$$

The decision boundary is then

$$log(\frac{\phi_k}{\phi_l}) - \frac{1}{2}(\mu_k - \mu_l)^T \Sigma^{-1}(\mu_k - \mu_l) - \mathbf{x}\Sigma^{-1}(\mu_k - \mu_l) = 0. \tag{6.17}$$

The first two terms do not depend on x and can be summarized as constant $\mathbf{a}$. We can also introduce the vector

$$\mathbf{w} = -\Sigma^{-1}(\mu_k - \mu_l). \tag{6.18}$$

With these simplifying notations is it easy to see that this decition boundary is a linear,

$$\mathbf{a} + \mathbf{w}\mathbf{x} = 0, \tag{6.19}$$

and this method with the Gaussian class distributions with equal variances is called **Linear Discriminant Analysis (LDA)**. The vector $\mathbf{w}$ is perpendicular to the decision surface. Examples are shown in Figure **??**. If we do not make the assumption of equal variances of the classes, than we have a quadratic equation for the decision boundary, and the method is then called **Quadratic Discriminant Analysis (GDA)**. With the assumptions of LDA, we can calculate the contrastive model directly using Bayes rule.

$$p(y = k|\mathbf{x}; \theta) = \frac{\phi_k \frac{1}{\sqrt{2\pi}^n \sqrt{|\Sigma|}} e^{-\frac{1}{2}(\mathbf{x}-\mu_k)^T \Sigma_k^{-1}(\mathbf{x}-\mu_k)}}{\phi_k \frac{1}{\sqrt{2\pi}^n \sqrt{|\Sigma|}} e^{-\frac{1}{2}(\mathbf{x}-\mu_k)^T \Sigma_k^{-1}(\mathbf{x}-\mu_k)} + \phi_l \frac{1}{\sqrt{2\pi}^n \sqrt{|\Sigma|}} e^{-\frac{1}{2}(\mathbf{x}-\mu_l)^T \Sigma_l^{-1}(\mathbf{x}-\mu_l)}} \tag{6.20}$$

$$= \frac{1}{1 + \frac{\phi_l}{\phi_k} exp^{-\theta^T x}}, \tag{6.21}$$

where θ is an appropriate function of the parameters μ_k, μ_l, and Σ. Thus, the contrastive model is equivalent to logistic regression discussed in the previous chapter, although we use parametrisations and the two methods will therefore usual give different results on specific data sets. So which method should be used? In LDA we made the assumption that each class is Gaussian distributed. If this is the case, then LDA is the best method we can use. Discriminant analysis is also popular since it often works well even when the classes are not strictly gaussian. However, as can be seen in Figure **??**B, it can also produce quite bad results if the data are multimodal distributed. Logistic regression is somewhat more general since it does not make the assumption that the class distributions are Gaussian. However, so far we have mainly looked at linear models and logistic regression would have also problems with the data shown in Figure **??**B.

Finally, we should nte that Fisher's original method was slightly more general than the examples discussed here since he did not assume Gaussian distributions. Instead considered within-class variances compared to between-class variances, something which resembles a signal-to-noise ratio. In **Fisher discriminant analysis (FDA)**, the separating hyperplane is defined as

$$\mathbf{w} = -(\Sigma_k + \Sigma_l)^{-1}(\mu_k - \mu_l). \tag{6.22}$$

which is the same as in LDA in the case of equal covariance matrices.

6.2 Naive Bayes

7 Graphical models

7.1 Causal models

In the regression example that looked at health data relating the weight of subjects to running times, we considered the weight to be a random variable to capture the uncertainties in the data. There we treated the running time as the dependent variable. Since this is also a measured quantity, it should also be treated as random variable. In general we should anyhow consider more complex models with many more factors described as random variables, and we are most interested in describe the relations of these variables in a model.

We have seen that probability theory is quite handy to model data, and probability theory also considers multiple random variables. The total knowledge about the co-occurance of specific values for two random variables X and Y is captured by the **joined probability function** $p(X, Y)$. This is a symmetric function,

$$p(X, Y) = p(Y, X), \tag{7.1}$$

There is an interesting limitation of joined density functions of multiple variables which only describe the co-occurance of specific values of the random variables. However, we want to use our knowledge about the world, captured by a model, to reason about possible events. For this we want to add knowledge or hypotheses about **causal relations**. For example, a fire alarm should be triggered by a fire, although there is some small chance that the alarm will not sound when the unit is defect. However, it is (hopefully) unlikely that the sound of a fire alarm will trigger a fire. It is useful to illustrate such casual relations with graphs such as

$$\begin{tikzcd} \textsf{F} \arrow[r] & \textsf{A} \end{tikzcd}$$

In such **graphical models**, the nodes represent random variables, and the links between them represent causal relations with conditional probabilities, $p(A|F)$. Since we use arrows on the links we are discussing here **directed graphs**, and we are also restricting our discussions here to graphs that have no loops, so called **acyclic graphs**. **Directed acyclic graphs** are also called **DAG**s.

Graphical causal models have been advanced largely by Judea Pearl, and the following example is taken from his book[7]. The model is shown in Figure 7.1. Each of the five nodes stands for a random binary variable (Burglary B={yes,no}, Earthquake E={yes,no}, Alarm A={yes,no}, JohnCalls J={yes,no}, MaryCalls M={yes,no}) The figure also include **conditional probability tables (CPTs)** that specify the conditional probabilities represented by the links between the nodes.

[7]Judea Pearl, ‘Causality: Models, Reasoning and Inference’, Cambridge University Press 2000, 2009’.

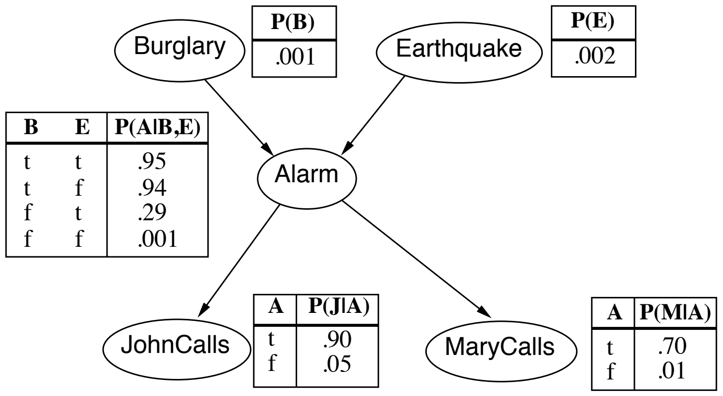


Fig. 7.1 Example of causal model a two-dimensional probability density function (pdf) and some examples of marginal pdfs.

The joined distribution of the five variables can be factories in various ways following the chain rule mentioned before (equations 4.29), for example as

$$p(B,E,A,J,M) = P(B|E,A,J,M)P(E|A,J,M)P(A|J,M)P(J|M)P(M) \tag{7.2}$$

However, the the causal model represents a specific factorization of the joined probability functions, namely

$$p(B,E,A,J,M) = P(B)P(E)P(A|B,E)P(J|A)P(M|A), \tag{7.3}$$

which is much easier to handle. For example, if we do not know the conditional probability functions, we need to run many more experiments to estimate the various conditions ($2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 31$) instead of the reduced conditions in the causal model ($1 + 1 + 2^2 + 2 + 2 = 10$). It is also easy to use the casual model to do inference (drawing conclusions), for specific questions. For example, say we want to know the probability that there was no earthquake or burglary when the alarm rings and both John and Mary call. This is given by

$$\begin{aligned}
&P(B=f,E=f,A=t,J=t,M=t) = \\
&\quad = P(B=f)P(E=f,)P(A=t|B=f,E=f)P(J=t|A=t)P(M=t|A=t) \\
&\quad = 0.998 * 0.999 * 0.001 * 0.7 * 0.9 \\
&\quad = 0.00062
\end{aligned}$$

Although we have a casual model where parents variables influence the outcome of child variables, we can also use a child evidence to infer some possible values of parent variables. For example, let us calculate the probability that the alarm rings given that John calls, $P(A=t|J=t)$. For this we should first calculate the probability that the alarm rings as we need this later. This is given by

$$\begin{aligned}
P(A=t) = &P(A=t|B=t,E=t)P(B=t)P(E=t) + ... \\
&P(A=t|B=t,E=f)P(B=t)P(E=f) + ...
\end{aligned}$$

$$\begin{aligned}
&P(A=t|B=f,E=t)P(B=f)P(E=t)+...\\
&P(A=t|B=f,E=f)P(B=f)P(E=f)\\
=\;&0.95*0.001*0.002+0.94*0.001*0.998+...\\
&0.29*0.999*0.002+0.001*0.999*0.998\\
=\;&0.0025
\end{aligned}$$

We can then use Bayes' rule to calculate the required probability,

$$\begin{aligned}
P(A=t|J=t) &= \frac{P(J=t|A=t)P(A=t)}{P(J=t|A=t)P(A=t)+P(J=t|A=f)P(A=f)}\\
&= \frac{0.90.0025}{0.90.0025+0.050.9975}\\
&= 0.0434
\end{aligned}$$

We can similarly apply the rules of probability theory to calculate other quantities, but these calculations can get cumbersome with larger graphs. It is therefore useful to use numerical tools to perform such inference. A Matlab toolbox for Bayesian networks is introduced in the next section.

While inference is an important application of causal models, inferring causality from data is another area where causal models revolutionize scientific investigations. Many traditional methods evaluate co-occurrences of events to determine dependencies, such as a correlation analysis. However, such a correlation analysis is usually not a good indication of causality. Consider the example above. When the alarm rings it is likely that John and Mary call, but the event that John calls is mutually independent of the event that Mary calls. Yet, when John calls it is also statistically more likely to observe the event that Mary calls. Sometimes we might just be interested in knowing about the likelihood of co-occurrence, for which a correlation analysis can be a good start, but if we are interested in describing the causes of the observations, then we need another approach. Some algorithms have been proposed for **structural learning**, such as an algorithm called **inferred causation (IC)**, which deduces the most likely causal structure behind given data is.

7.2 Bayes Net toolbox

An Matlab implementation of various algorithms for inference, parameters estimation and inferred causation is provided by Kevin Murphy in the Bayes Net toolbox[8]. We will demonstrate some of its features on the burglary/earthquake example above.

The first step is to create a graph structure for the DAG We have five nodes. The nodes are given numbers, but we also use variables with capital letter names to refer to them. The DAG is then a matrix with entries 1 where directed links exist.

```
N=5;% number of nodes
B=1; E=2; A=3; J=4; M=5;
dag = zeros(N,N);
```

[8]The toolbox can be downloaded at http://code.google.com/p/bnt.

```
dag(B,A)=1;
dag(E,A)=1;
dag(A,[J M])=1;
```

The nodes represent discrete random variables with two possible state. We only discuss here discrete random variables, although the toolbox contains methods for continuous random variables. For the discrete case we have to specify the number of possible states of each variable, and we can then create the corresponding Bayesian network,

```
% Make bayesian network
node_sizes=[2 2 2 2 2];  %binary nodes
bnet=mk_bnet(dag,node_sizes); %make bayesian net
```

The next step is to provide the numbers for the conditional probability distributions, which are the conditional probability tables for discrete variables. For this we provide the numbers in a vector according to the following convention. Say we specify the probabilities for node 3, which is conditionally dependent on nodes 1 and 2. We then provide the probabilities in the following order:

Node 1	Node 2	P(Node 3=X)
F	F	F
T	F	F
F	T	F
T	T	F
F	F	T
T	F	T
F	T	T
T	T	T

For our specific examples, the CPT are thus specified as

```
bnet.CPD{B} = tabular_CPD(bnet,B,[0.999 0.001]);
bnet.CPD{E} = tabular_CPD(bnet,E,[0.998 0.002]);
bnet.CPD{A} = tabular_CPD(bnet,A,[0.999 0.06 0.71 0.05 0.001 0.94 0.29 0.95]);
bnet.CPD{J} = tabular_CPD(bnet,J,[0.95 0.10 0.05 0.90]);
bnet.CPD{M} = tabular_CPD(bnet,M,[0.99 0.30 0.01 0.70]);
```

We are now ready to calculate some inference. For this we need to specify a specific inference engine. There are several algorithms implemented, a variety of exact algorithm as well as approximate procedures in case the complexity of the problem is too large. Here we use the basic exact inference engine, the **junction tree algorithm**, which is based on a message passing system.

```
engine=jtree_inf_engine(bnet);
```

While this is an exact inference engine, there are other engines, such as approximate engines, that might be employed for large graphs when other methods fail.

As an example of an inference we recalculate the example above, that of calculate the probability that the alarm rings given that John calls, $P(A = t|J = t)$. For this we have to enter some evidence, namely that $J = t$, into a cell array and add this to the inference engine,

```
evidence=cell(1,N);
evidence{J}=2;
```

```
[engine,loglik]=enter_evidence(engine,evidence)
```

We can then calculate the marginal distribution for a variable, given the evidence as,

```
marg=marginal_nodes(engine,A)
p=marg.T(2)
```

It is now very easy to calculate other probabilities.

The Bayesnet toolbox also includes routines to handle some continuous models such as models with Gaussian nodes. In addition, there are routines to do parameter estimation, including point estimates, such maximum likelihood estimation, and also full bayesian priors. Finally, he toolbox includes routines to inferred causation through structural learning.

Exercise:

In the above example, shown in Figure 7.1, calculate the probability that there was a burglary, given that John and Marry called.

7.3 Temporal Bayesian networks: Markov Chains and Bayes filters

In many situations we are interested in how things change over time. For this we need to build time-dependent causal models, also called **dynamic bayesian models (DBNs)**. We will discuss here a specific family of such models in which the states at one time, t, only depend on the random variables at the previous time, $t-1$. This condition is called a **Markov condition**, and the corresponding models are called **markov chains**. A very common situation is captured in the example shown in Figure 7.2. In this example we have three time-dependent random variables called $u(t)$, $x(t)$, and $z(t)$. We used different grey shades for the nodes to indicate if they are observed or not. Only $u(t)$ and $z(t)$ are observed, whereas $x(t)$ is an un-observed, **hidden** or **latent** variable. Such Markov chains are called **Hidden Markov Models (HMMs)**.

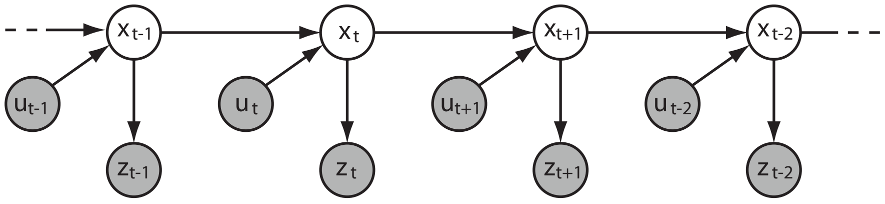


Fig. 7.2 A hidden markov model (HMM) with time dependent state variable x, motor control u and sensor readings z.

The HMM shown in figure 7.2 capture the basic operation of a mobile robot in which the variable $u(t)$ represents the motor command at time t and the variable $z(t)$

represents sensor readings at time t. The motor command is the cause that the robot goes into a new state $x(t)$. The problem illustrated in the figure is that the new state of the robot, such as its location or posture, can not be observed directly. In the following we will specifically discuss **robot localization**, where we must infer the possible state from the motor command and sensor reading. We have previously used a state sheet to 'measure' the state (location) of the tribot with the light sensor (see section 5.3). The model above allows us a much more sophisticated guess of the location by taking not only the current reading into account, but by combining this information with our previous guess of the location and the knowledge of the motor command. More specifically, we want to calculate the **believe** of the state, which is a probability (density) function over the state space

$$bel(x_t) = p(x_t|bel(x_0), u_{1:t}, z_{1:t}), \tag{7.4}$$

give an initial believe and the history of motor command and sensor reading. This believe can be calculated recursively from the previous believes and the new information, which is a form of **believe propagation**. It is often convenient to break this process down into calculating a new **prediction** from the previous believe and the motor command, and then to add the knowledge provided by the new sensor reading. To predict of probability of a new state from the previous believes and the current motor command, we need to marginalize over the previous states, that is, we need to multiply the previous believe with the probability of the new state, given the previous state and motor command, and to sum (integrate) over it,

$$pred(x_t) = \frac{1}{N_x} \sum_{x_{t-1}} p(x_t|u_t, x_{t-1}) bel(x_{t-1}), \tag{7.5}$$

where N_x is the number of states. This prediction can be combined with the sensor reading to give a new believe,

$$bel(x_t) = \frac{p(z_t|x_t)pred(x_t)}{\sum_{x_t} p(z_t|x_t)pred(x_t)}, \tag{7.6}$$

where we included the normalization over all possible states to get a number representing probabilities. This is called **Bayes filtering**. With a Bayes filter we can calculate the new believe from the previous believe, the current motor command, and the latest sensor reading. This is the best we can do with the available knowledge to estimate the positions of a system. The application of Bayes filtering to a robot localization is also called **Markov localization** and is illustrated in Figure 7.3. The limitation in practice is often the that we have to sum (integrate) over all the possible states, which might be a very large sum if it has to be done explicitly. In some cases we can do this analytically, as shown in the next sections.

7.3.1 The Kalman filter

We have, so far, only assumed that our process complies with the Markov condition in that the new state only depends on the previous state and current motor command.

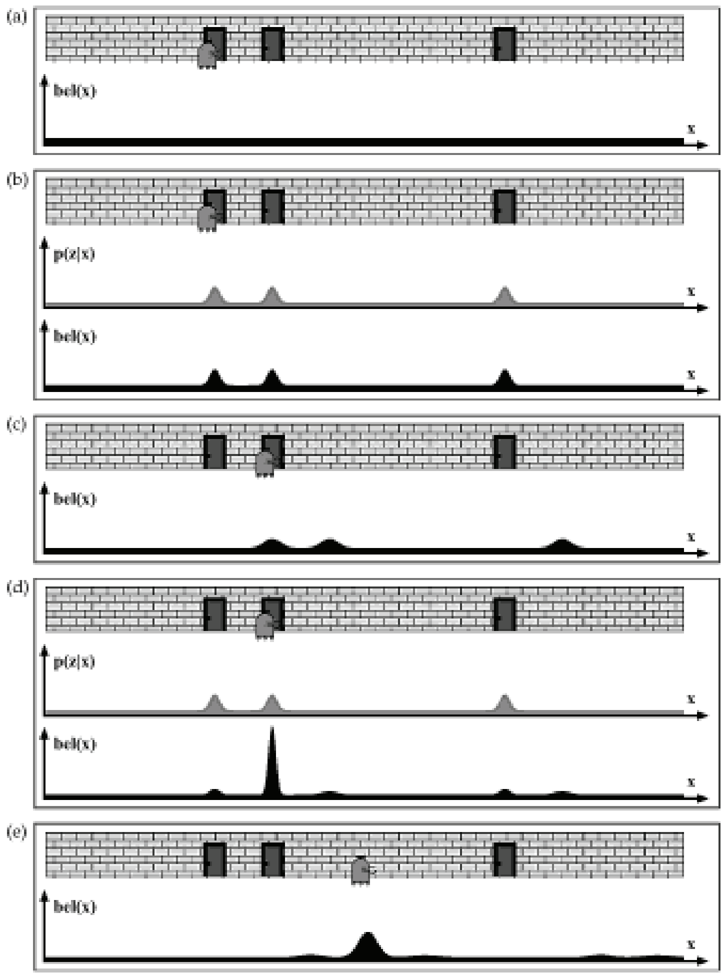


Fig. 7.3 A illustration of markov localization [from *Probabilistic model*, Thrun, Burgard, Fox, MIT press 2006].

In many cases we might have a good estimate of the state at some point with some variance. We now assume that our believes are Gaussian distributed,

$$p(\mathbf{x}_{t-1}) = \frac{1}{(\sqrt(2\pi))^n \sqrt(\det(\mathbf{\Sigma}_{t-1})} \exp(-\frac{1}{2}(\mathbf{x}_{t-1} - \mu_{t-1})^T \mathbf{\Sigma}_{t-1}^{-1}(\mathbf{x}_{t-1} - \mu_{t-1})). \tag{7.7}$$

We also consider first the case where the transition probability $p(\mathbf{x}_t|\mathbf{u}_t, \mathbf{x}_{t-1})$ is linear in the previous state and the motor command, up to Gaussian noise ϵ_t,

$$\bar{\mathbf{x}}_t = \mathbf{A}_t\mathbf{x}_{t-1} + \mathbf{B}_t\mathbf{u}_t + \epsilon_t; \;\; \epsilon_t \sim N(0, \mathbf{Q}_t), \tag{7.8}$$

where $\mathbf{A}_t$ and $\mathbf{B}_t$ are time dependent matrices. The gaussian noise ϵ_t has thereby mean zero and covariance $\mathbf{Q}_t$. This setting is very convenient since the posterior after moving a Gaussian uncertainty in such a linear fashion is again a Gaussian. The believe after incorporating the movement is hence a Gaussian probability over states with parameters

$$\bar{\mu}_t = \mathbf{A}_t \mu_{t-1} + \mathbf{B}_t \mathbf{u}_t \tag{7.9}$$
$$\bar{\mathbf{\Sigma}}_t = \mathbf{A}_t \mathbf{\Sigma}_{t-1} \mathbf{A}_t^T + \mathbf{Q}_t. \tag{7.10}$$

We now need to take the measurement probability into account, $P(\mathbf{z}_t|\bar{\mathbf{x}}_t)$, which we also assume to be linear in its argument up to a Gaussian noise δ_t,

$$\mathbf{z}_t = \mathbf{C}_t \bar{\mathbf{x}}_t + \delta_t; \quad \delta_t \sim N(0, \mathbf{R}_t). \tag{7.11}$$

With this measurement update, we can calculate our new state estimate, parameterized by μ_t and $Sigma_t$, as

$$\bar{\mathbf{K}}_t = \bar{\mathbf{\Sigma}}_t \mathbf{C}_t^T (\mathbf{C}_t \bar{\mathbf{\Sigma}}_t \mathbf{C}_t^T + \mathbf{R}_t)^{-1} \tag{7.12}$$
$$\mu_t = \bar{\mu}_t + \mathbf{K}_t(\mathbf{z}_t - \mathbf{C}_t \bar{\mu}_t) \tag{7.13}$$
$$\mathbf{\Sigma}_t = (\mathbf{I} - \mathbf{K}_t \mathbf{C}_t) \bar{\mathbf{\Sigma}}_t, \tag{7.14}$$

where **I** is the identity matrix.

The basic Kalman filter makes several simplifying assumptions such as Gaussian believes, Gaussian noise, and linear relations. There are many generalizations of this method. For example, it is possible to extend the method to non-linear transformations while still demanding that the posteriors are Gaussian. While this introduces some errors and is this only an approximate method, this **extended Kalman filter (EKF)** is often very useful in practical applications. Also, there are several non-parametric methods such as **particle filters**.

Exercises:

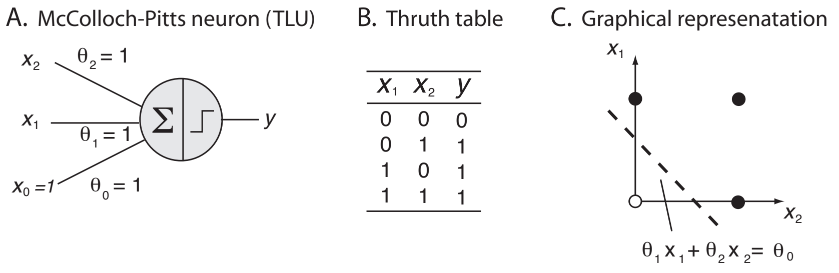
1. A mobile robot is instructed to travel 3 meters along a line in each time step. The true distance traveled is gaussian distributed around the intended position with standard deviation of 2 meters; the positions sensors are also unreliable with Gaussian that has a standard deviation of 4 meters; What is the average absolute difference between the true position and the estimated position when using only the sensor information, only the information inherent in the motor command, and both sources of information?
2. The lego robot is traveling along a line that has dark stripes at various positions. These distance between the stripes stripes are doubling for each consecutive stipe ($d(i) = 2 * d(i-1)$). Estimate the position of the Lego tribot when traveling straight ahead if the robot is positioned at a random initial position.

8 General learning machines

The previous models, through the formulation of specific hypothesis functions, have been designed specifically for each applications. However, much more practical would be to have more general machines that can learn without making very specific functional assumptions. But how can we do this? The general idea is to provide a very general functions with many parameters that will be adjusted through learning. Of course, the real problem is then to not over-fit the model by using appropriate restrictions and also to make the learning efficient so that it can be used to large problem size. This chapter starts with a brief historical introduction to general learning machines and neural networks. We then discuss support vector machines and more rigorous learning theories.

8.1 The Perceptron

There was always a strong interest of AI researchers in **real intelligence**, that is, to understand the human mind. For example, both Alan Turing and John von Neumann worked more directly on biological systems in their last years before their early passing, and human behaviour and the brain have always been of interest to AI researchers. Of course, we want to understand how the brain is working in its own right, but it is also a great example of a quite general and successful learning machine.


A. McColloch-Pitts neuron (TLU)
x_2
$\theta_2 = 1$
x_1
$\theta_1 = 1$
Σ
y
$x_0 = 1$
$\theta_0 = 1$
B. Thruth table
C. Graphical represenatation
x_1
x_2
$\theta_1 x_1 + \theta_2 x_2 = \theta_0$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

Fig. 8.1 Representation of the boolean OR function with a McCulloch-Pitts neuron (TLU).

A seminal paper, which has greatly influenced the development of early learning machines, is the 1943 paper by Warren McCulloch and Walter Pitts. In this paper, they proposed a simple model of a neuron, called the **threshold logical unit**, often called now the **McCulloch–Pitts neuron**. Such a unit is shown in Fig. 8.1A with three input channels, although it could have an arbitrary number of input channels. Input values are labeled by x with a subscript for each channel. Each channel has also a **weight**

parameter, θ_i. The McCulloch–Pitts neuron operates in the following way. Each input value is multiplied with the corresponding weight value, and these weighted values are then summed. If the weighted summed input is larger than a certain threshold value, $-\theta_0$, then the output is set to one, and zero otherwise, that is,

$$h(\mathbf{x};\theta) = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} \theta_i x_i = \theta^T \mathbf{x} > 0 \\ 0 \qquad\quad \text{otherwise} \end{cases} . \tag{8.1}$$

Such an operation resembles, to some extend, a neuron in that a neuron is also summing synaptic inputs and fires (has a spike in its membrane potential) when the membrane potential reaches a certain level that opens special voltage-gated ion channels. McCulloch and Pitts introduced this unit as a simple neuron model, and they argued that such a unit can perform computational tasks resembling boolean logic. This is demonstrated in Fig. 8.1. The symbol h is used in these lecture notes since the output of the perceptron is the **hypothesis** of the perceptron, given the parameters θ.

The next major developments in this area were done by Frank Rosenblatt and his engineering colleague Charles Wightman (Fig. 8.2), using such elements to build a machine that Rosenblatt called the **perceptron**. As can be seen in the figures, they worked on a machine that can perform letter recognition, and that the machine consisted of a lot of cables, forming a network of simple, neuron-like elements.

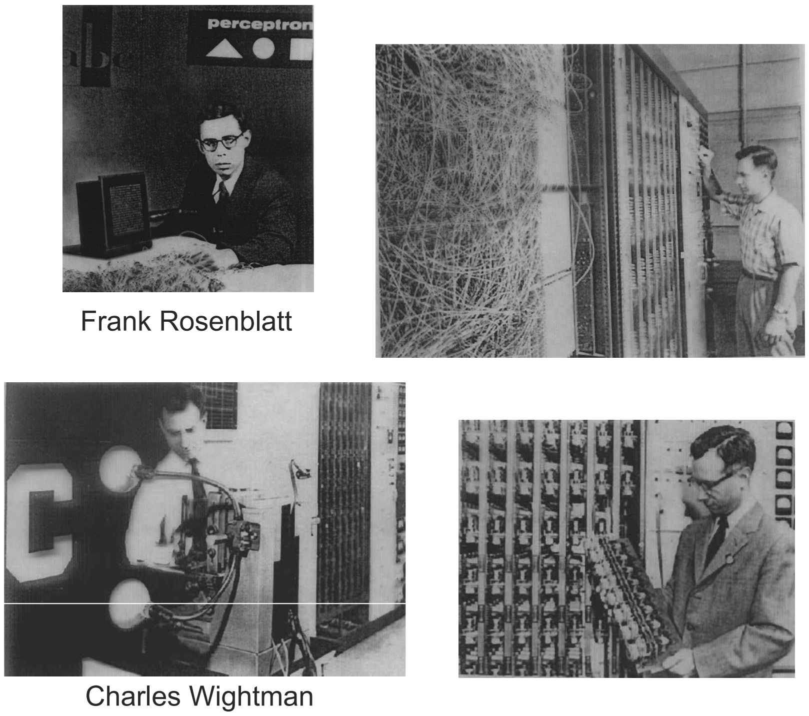


Fig. 8.2 Neural Network computers in the late 1950s.

The most important challenge for the team was to find a way how to adjust the parameters of the model, the connection weights θ_i, so that the perceptron would perform a task correctly. The procedure was to provide to the system a number of examples, let's say m input data, $\mathbf{x}^{(i)}$ and the corresponding desired outputs, $y^{(i)}$. The procedure to update the parameters of the systems based on these training examples is called a **learning rule**, and this form of learning with explicit examples is called **supervised learning**. The **perceptron learning rule** is given by

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(\mathbf{x_i}) \right) x_j^{(i)}. \tag{8.2}$$

This learning rule is also known, or related to, the Widrow-Hoff learning rule, the Adaline rule, and the delta rule.[9] It is often called the delta rule because the difference between the desired and actual output (difference between actual (training) data and hypothesis) to guide the learning. When multiplying out the difference with the inputs we have end up with the product of the activity between the inputs and output values for each synaptic channel. Such a learning rule is also called Hebbian after the famous NovaScotian Donald Hebb.

There was a lot of excitement during the 1960s in the AI and psychology community about such learning system that resemble some brain functions. However, Minsky and Peppert showed in 1968 that such perceptrons can not represent all possible boolean functions (sometimes called the XOR problem). While it was known at this time that this problem could be overcome by a layered structure of such perceptrons (called **multilayer perceptrons**), a learning algorithms was not widely known at this time. This killed the field, and the AI community concentrated on rule-based systems in the following years. The generalization of a delta rule, known as error-backpropagation, was finally introduced by Rumelhart, Hinton and Williams in 1992 (although Paul Werbos, and also Sunichi Amari, used it before), and resulted in the explosion of the field of **Neural Networks**. This area has now become known as machine learning, which has clarified a lot of the abilities and challenges of neural networks. We therefore follow in the next sections a more contemporary path.

The area of Neural Networks has been active since the 1950s. A large potion of this area is concerned with supervised learning, which we will discuss further in the next sections. This area is now mainly absorbed into the field of machine learning. There is also an active area of modelling brain functions known as **computational neuroscience**. This area is subject of CSCI6508/NESC4177 taught in the winter term. While these fields have developed into some different directions and have some distinct goals, there is now some exciting convergence when it comes to unsupervised learning and complex modelling. This will be subject of later discussions in this course.

8.2 Support Vector Machines (SVM)

8.2.1 Linear classifiers with large margins

In this section we briefly outline the basic idea behind Support Vector Machines (SVM) that are currently thought to be the best general purpose supervised classifier algorithm.

[9]These learning rules are nearly identical, but are sometimes used in slightly different contexts

SVMs, and the underlying statistical learning theory, has been worked out by Vladimir Vapnik since the early 1960, but some breakthroughs were also made in the late 1990 with some collaborators like Corinna Cortes, Chris Burges, Alex Smola, and Bernhard Schölkopf to name but a few, and SVM have since become very popular and hard to beat. While we outline some of the underlying formulas, we do not derive all the steps but will try to give some intuition. A more thorough treatment can be found in the references on the web page. The here we just want to provide the big picture, but need to show some formulas to highlight some of the discussion.

The basic SVMs are concerned with binary classification. Figure 8.3 shows an example of two classes, depicted by different symbols, in a two dimensional attribute space. We distinguish here attributes from features as follows. Attributes are the raw measurements, where as features can be made up by combining attributes. For example, the attributes x_1 and x_2 could be combines in a feature vector $(x_1, x_1x_2, x_2, x_1^2, x_2^2)^T$. This will become important later, but it is important to introduce the notation here. Our training set consists again of m data with attribute values $\mathbf{x}^{(i)}$ and labels $y^{(i)}$. We chose here the labels of the two classes as $y \in \{-1, 1\}$, as this will nicely simplify some equations.

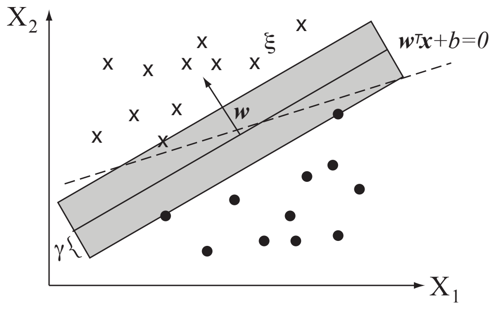


Fig. 8.3 Illustration of linear support vector classification.

The two classes in the figure 8.3 can be separated by a line, which can be parameterized by

$$w_1x_1 + w_2x_2 - b = \mathbf{w}^T\mathbf{x} - b = 0. \tag{8.3}$$

While the first equation shows the lines equation with its components in two dimensions, the next expression is the same in any dimension. Of course, in three dimension we would talk about a plane. In general, we will talk about a **hyperplane** in any dimensions. The particular hyperplane is the dividing or separating hyperplane between the two classes. We also introduce what the **margin** γ, which is the perpendicular distance between the dividing hyperplane and the closest point.

The main point to realize now is that the dividing hyperplane that maximizes the the margin, the so called **maximum margin classifier**, is the best solution we can find. Why is that? We should assume that the training data, shown in the figure, are some unbiased examples of the true underlying density function describing the distribution of points within each class. It is then likely that new data points, which we want to classify, are close to the already existing data points. Thus, if we make the separating hyperplane as far as possible from each point, than it is most likely to not make wrong classification. Or, with other words, a separating hyperplane like the one shown as dashed line in the figure, is likely to generalize much worse than the maximum margin hyperplane. So the maximum margin hyperplane is the best generalizer for binary classification for the training data.

What is if we can not divide the data with a hyperplane and we have to consider non-linear separators. Don't we then run into the same problems as outlined before, specifically the bias-variance tradeoff? Yes, indeed, this will still be the challenge, and our aim is really to work on this problem. But before going there it is useful to formalize the linear separable case in some detail as the representation of the optimization problem will be a key in applying some tricks later.

Learning a linear maximum margin classifier on labeled data means finding the parameters (w) and b that maximizes the margin. For this we could computer the distances of each point from the hyperplane, which is simply a geometric exercise,

$$\gamma^{(i)} = y^{(i)} \left((\frac{\mathbf{w}}{||\mathbf{w}||})^T \mathbf{x}^{(i)} + \frac{b}{||\mathbf{w}||} \right). \tag{8.4}$$

The vector $\mathbf{w}/||\mathbf{w}||$ is the normal vector of the hyperplane, a vector of unit length perpendicular to the hyperplane. We overall margin we want to maximize is the distance to the closest point,

$$\gamma = \min_i \gamma^{(i)}. \tag{8.5}$$

By looking at equation 8.4 we see that maximizing γ is equivalent to minimizing $||\mathbf{w}||$, or, equivalently, of minimizing

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2. \tag{8.6}$$

More precisely, we want to maximize this margin under the constraint that no training data lies within the margin,

$$\mathbf{w}^T\mathbf{x} + b \geq 1 \quad \text{for} \quad y^{(i)} = 1 \tag{8.7}$$

$$\mathbf{w}^T\mathbf{x} + b \leq -1 \quad \text{for} \quad y^{(i)} = -1, \tag{8.8}$$

which can nicely be combines with our choice of class representation as

$$y^{(i)}(\mathbf{w}^T\mathbf{x} + b) \leq 1. \tag{8.9}$$

Thus we have a quadratic minimization problem with linear inequalities as constraint. Taking a constrain into account can be done with a **Lagrange formalism**. For this we simply add the constraints to the main objective function with parameters α_i called Lagrange multipliers,

$$Ł^P(\mathbf{w}, b, \alpha_i) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{m} \alpha_i [y^{(i)}(\mathbf{w}^T\mathbf{x} + b) - 1]. \tag{8.10}$$

The Lagrange multipliers determine how well the constrain are observed. In the case of $\alpha_i = 0$, the constrains do not matter. In order conserve the constrains, we should thus make these values as big as we can. Finding the maximum margin classifier is given by

$$p^* = \min_{\mathbf{w},b} \max_{\alpha_i} Ł^P(\mathbf{w}, b, \alpha_i) \leq p^* = \max_{\alpha_i} \min_{\mathbf{w},b} Ł^D(\mathbf{w}, b, \alpha_i) = d^*. \tag{8.11}$$

In this formula we also added the formula when interchanging the min and max operations, and the reason for this is the following. It is straight forward to solve the optimization problem on the left hand side, but we can also solve the related problem on the right hand side which turns out to be essential when generalizing the method to nonlinear cases below. Moreover, the equality hold when the optimization function and the constraints are convex[10]. So, if we minimize Ł by looking for solutions of the derivatives $\frac{\partial Ł}{\partial \mathbf{w}}$ and $\frac{\partial Ł}{\partial b}$, we get

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i y^{(i)} \mathbf{x}^{(i)} \tag{8.12}$$

$$0 = \sum_{i=1}^{m} \alpha_i y^{(i)} \tag{8.13}$$

Substituting this into the optimization problem we get

$$\max_{\alpha_i} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} y^{(i)} y^{(y)} \alpha_i \alpha_j \mathbf{x}^{(i)T} \mathbf{x}^{(j)}, \tag{8.14}$$

subject to the constrains

$$\alpha_i \geq 0 \tag{8.15}$$

$$\sum_{i=1}^{m} \alpha_i y^{(i)} = 0. \tag{8.16}$$

From this optimization problem it turns out that the α_i's of only a few examples, those ones that are lying on the margin, are the only ones with have $\alpha_i \neq 0$. The corresponding training examples are called **support vectors**. The actual optimization can be done with several algorithms. In particular, John Platt developed the sequential minimal optimization (SMO) algorithm that is very efficient for this optimization problem. Please note that the optimization problem is convex and can thus be solved very efficiently without the danger of getting stuck in local minima.

Once we found the support vectors with corresponding α_i's, we can calculate (w) from equation 8.12 and b from a similar equation. Then, if we are given a new input vector to be classified, this can then be calculated with the hyperplane equation 8.3 as

$$y = \begin{cases} 1 & \text{if} \quad \sum_{i=1}^{m} \alpha_i y^{(i)} \mathbf{x}^{(i)T} \mathbf{x} > 0 \\ -1 & \text{otherwise} \end{cases} \tag{8.17}$$

Since this is only a sum over the support vectors, classification becomes very efficient after training.

[10]Under these assumptions there are other conditions that hold, called the **Karush-Kuhn-Tucker** conditions, that are useful in providing proof in the convergence of these the methods outlined here.

8.2.2 Soft margin classifier

So far we only discussed the linear separable case. But how about the case when there are overlapping classes? It is possible to extend the optimization problem by allowing some data points to be in the margin while penalizing these points somewhat. We include therefore some **slag variables** ξ_i that reduce the effective margin for each data point, but we add to the optimization a penalty term that penalizes if the sum of these slag variables are large,

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C\sum_i \xi_i, \tag{8.18}$$

subject to the constrains

$$y^{(i)}(\mathbf{w}^T\mathbf{x} + b) \geq 1 - \xi_i \tag{8.19}$$

$$\xi_i \geq 0 \tag{8.20}$$

The constant C is a free parameter in this algorithm. Making this constant large means allowing less points to be in the margin. This parameter must be tuned and it is advisable to at least try to vary this parameter to verify that the results do not dramatically depend on a initial choice.

8.2.3 Nonlinear Support Vector Machines

We have treated the case of overlapping classes while assuming that the best we can do is still a linear separation. But what if the underlying problem is separable, f only with a more complex function. We will now look into the non-linear generalization of the SVM.

When discussing regression we started with the linear case and then discussed non-linear extensions such as regressing with polynomial functions. For example, a linear function in two dimensions (two attribute values) is given by

$$y = w_0 + w_1x_1 + w_2x_2, \tag{8.21}$$

and an example of a non-linear function, that of an polynomial of 3rd order, is given by

$$y = w_0 + w_1x_1 + w_2x_2 + w_3x_1x_2 + w_4x_1^2 + w_5x_2^2. \tag{8.22}$$

The first case is a linear regression of a feature vector

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}. \tag{8.23}$$

We can also view the second equation as that of linear regression on a feature vector

$$\mathbf{x} \rightarrow \phi(\mathbf{x}) = \begin{pmatrix} x_1 \\ x_2 \\ x_1x_2 \\ x_1^2 \\ x_2^2 \end{pmatrix}, \tag{8.24}$$

which can be seen as a mapping $\phi(\mathbf{x})$ of the original attribute vector. We call this mapping a **feature map**. Thus, we can use the above maximum margin classification method in non-linear cases if we replace all occurrences of the attribute vector $\mathbf{x}$ with the mapped feature vector $\phi(\mathbf{x})$. There are only two problems remaining. One is that we have again the problem of overfitting as we might use too many feature dimensions and corresponding free parameters w_i. The second is also that with an increased number of dimensions, the evaluation of the equations becomes more computational intensive. However, there is a great trick to alleviate the later problem in the case when the methods only rely on dot products, like in the case of the formulation in the dual problem. In this, the function to be minimized, equation 8.14 with the feature maps, only depends on the dot products $\phi(\mathbf{x}^{(i)})^T\phi(\mathbf{x}^{(j)})$. Also, when predicting the class for a new input vector $\mathbf{x}$ from equation 8.12 when adding the feature maps, we only need the resulting values for the dot products $\phi(\mathbf{x}^{(i)})^T\phi(\mathbf{x})$ which can sometimes be represented as function called **Kernel function**,

$$K(\mathbf{x},\mathbf{z}) = \phi(\mathbf{x})^T\phi(\mathbf{z}). \tag{8.25}$$

Instead of actually specifying a feature map, which is often a guess to start, we could actually specify a Kernel function. For example, let us consider a quadratic feature map

$$K(\mathbf{x},\mathbf{z}) = (\mathbf{x}^T\mathbf{z} + c)^2. \tag{8.26}$$

We can then try to write this in the form of equation 8.25 to find the corresponding feature map. That is

$$\begin{align}
K(\mathbf{x},\mathbf{z}) &= (\mathbf{x}^T\mathbf{z})^2 + 2c\mathbf{x}^T\mathbf{z} + c^2 \tag{8.27}\\
&= (\sum_i x_i z_i)^2 + 2c\sum_i x_i z_i + c^2 \tag{8.28}\\
&= \sum_j\sum_i (x_i x_j)(z_i z_j) + \sum_i (\sqrt{(2c)}x_i)(\sqrt{(2c)}z_i) + cc \tag{8.29}\\
&= \phi(\mathbf{x})^T\phi(\mathbf{z}), \tag{8.30}
\end{align}$$

with

$$\phi(\mathbf{x}) = \begin{pmatrix} x_1x_1 \\ x_1x_2 \\ \dots \\ x_nx_1 \\ x_nx_2 \\ \dots \\ \sqrt{2c}x_1 \\ \sqrt{2c}x_2 \\ \dots \\ c \end{pmatrix}, \tag{8.31}$$

The dimension of this feature vector is $O(n^2)$ for n original attributes. Thus, evaluating the dot product in the mapped feature space is much more time consuming then calculating the Kernel function which is just the square of the dot product of the original attribute vector. The dimensionality Kernels with higher polynomials is quickly rising, making the benefit of the Kernel method even more impressive.

While we have derived the corresponding feature map for a specific Kernel function, this task is not always easy and not all functions are valid Kernel functions. We have also to be careful that the Kernel functions still lead to convex optimization problems. In practice, only a small number of Kernel functions is used. Besides the polynomial Kernel mention before, one of the most popular is the Gaussian Kernel,

$$K(\mathbf{x}, \mathbf{z}) = \exp -\frac{||\mathbf{x} - \mathbf{z}||^2}{2\gamma^2}, \tag{8.32}$$

which corresponds to an infinitely large feature map.

As mentioned above, a large feature space corresponds to a complex model that is likely to be prone to overfitting. We must therefore finally look into this problem. The key insight here is that we are already minimizing the sum of the components of the parameters, or more precisely the square of the norm $||\mathbf{w}||^2$. This term can be viewed as **regularization** which favours a smooth decision hyperplane. Moreover, we have discussed two extremes in classifying complicated data, one was to use Kernel functions to create high-dimensional non-linear mappings and hence have a high-dimensional separating hyperplane, the other method was to consider a low-dimensional separating hyperplane and interpret the data as overlapping. The last method includes a parameter C that can be used to tune the number of data points that we allow to be within the margin. Thus, we can combine these two approaches to classify non-linear data with overlaps where the soft margins will in addition allow us to favour more smooth dividing hyperplanes.

8.2.4 Regularization and parameter tuning

In practice we have to consider several free parameters when applying support vector machines. First, we have to decide which Kernel function to use. Most packages have a number of choices implemented. We will use for the following discussion the Gaussian Kernel function with width parameter γ. Setting a small value for γ and allowing for a large number of support vectors (small C), corresponds to a complex model. In contrast, larger width values and regularization constant C will increase the stiffness of the model and lower the complexity. In practice we have to tune these parameters to get good results. To do this we need to use some form of validation set, as discussed in section 5.6, and k-times cross validation is often implemented in the software packages. An example of the SVM performance (accuracy) on some examples (Iris Data set from the UCI repository; From Broadman and Trappenberg, 2006) is shown in figure 8.4 for several values of γ and C. It is often typical that there is a large area where the SVM works well and has only little variations in terms of performance. This robustness has helped to make SVMs practical methods that often outperform other methods. However, there is often also an abrupt onset of the region where the SVM fails, and some parameter tuning is hence required. While just trying a few settings might be sufficient, some more systematic methods such as grid search or simulated annealing also work well.

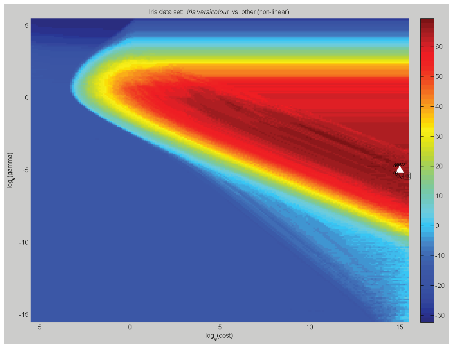


Fig. 8.4 Illustration of SVM accuracy for different values of paraneters C abd γ.

8.2.5 Statistical learning theory and VC dimension

SVMs are good and practical classification algorithms for several reasons, including the advantage of being convex optimization problem that than can be solved with quadratic programming, have the advantage of being able to utilize the Kernel trick, have a compact representation of the decision hyperplane with support vectors, and turn out to be fairly robust with respect to the hyper parameters. However, in order to be good learners, they need to moderate the variance-bias tradeoff dicussed in section 5.6. A great theoretical contributions of Vapnik and colleagues was the embedding of supervised learning into statistical learning theory and to derive some bounds that make statements on the average ability to learn form data. We outline here briefly the ideas and state some of the results. We discuss this issue here in the context of binary classification, although similar observations can be made in the case of multiclass classification and regression.

We start again by stating our objective, which is to find a hypothesis which minimized the generalization error. To state this a bit more differentiated and to use the nomenclature common in these discussions, we call the error function here the **risk function** R. In particular, the **expected risk** for a binary classification problem is the probability of misclassification,

$$R(h) = P(h(x) \neq y) \tag{8.33}$$

Of course, we generally do not know this density function, though we need to approximate this with our validation data. We assume thereby again that the samples are iid (independent and identical distributed) data, and can then estimate what is called the **empirical risk**,

$$\hat{R}(h) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}(h(\mathbf{x}^{(i)};\theta) = y^{(i)}). \tag{8.34}$$

We use here again m as the number of examples, but note that this is here the number of examples in the validations set, which is the number of all training data minus the ones used for training. Also, we will discuss this empirical risk further, but note that it is better to use the regularized version that incorporates a smoothness constrain such as

$$\hat{R}_{rmreg}(h) = \frac{1}{m}\sum_i \mathbb{1}(h(\mathbf{x}^{(i)};\theta) = y^{(i)}) - \lambda||\mathbf{w}||^2 \tag{8.35}$$

in the case of SVM, where λ is a regularization constant. Thus, wherever $\hat{R}(h)$ is used in the following, we can replace this with $\hat{R}_{rmreg}(h)$. **Empirical risk minimization** is the process of finding the hypothesis $\hat{h}$ that minimizes the empirical risk,

$$\hat{h} = arg\min_h \hat{R}(h). \tag{8.36}$$

The empirical risk is the MLE of the mean of a Bernoulli-distributed random variable with true mean $R(h)$. Thus, the empirical risk is itself a random variable for each possible hypothesis h. Let us first assume that we have k possible hypothesis h_i. We now draw on a theorem by Hoeffding called the **Hoeffding inequality** that provides and upper bound for the sum of random numbers to its mean,

$$P(|R(h_i) - \hat{R}(h_i)| > \gamma) \leq 2\exp(-2\gamma^2 m). \tag{8.37}$$

This formula states that there is a certain probability that we make an error larger than γ for each hypothesis of the empirical risk compared to the expected risk, although the good news is that this probability is bounded and that the bound itself becomes exponentially smaller with the number of validation examples. This is already an interesting results, but we now want to know the probability that some, out of all possible hypothesis, are less than γ. Using the fact that the probability of the union of several events is always less or equal to the sum of the probabilities, one can show that with probability $1 - \delta$ the error of a hypothesis is bounded by

$$|R(h) - \hat{R}(h)| \leq \sqrt{\frac{1}{2m}\log\frac{2k}{\delta}}. \tag{8.38}$$

This is a great results since it shows how the error of using an estimate the risk, the empirical risk that we can evaluate from the validation data, is getting smaller with training examples and with the number of possible hypothesis.

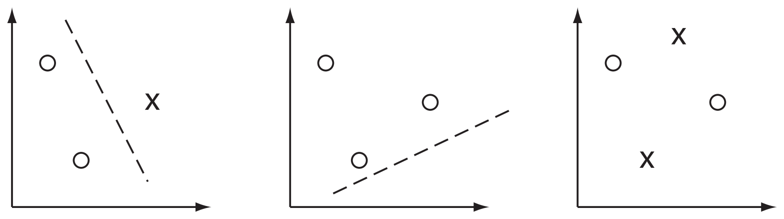


Fig. 8.5 Illustration of VC dimensions for the class of linear functions in two dimensions.

While the error scales only with the log of the number of possible hypothesis, the values goes still to infinite when the number of possible hypothesis goes to infinite, which much more resembles the situation when we have parameterized hypothesis. However, Vapnik was able to show the following generalization in the infinite case, which is that given a hypothesis space with **Vapnic-Chervonencis** dimension VC($\{$h$\}$), then, with probability $1 - \delta$, the error of the empirical risk compared to the expected risk (true generalization error) is

$$|R(h) - \hat{R}(h)| \leq O\left(\sqrt{\frac{VC}{m}\log\frac{m}{VC} + \frac{1}{m}\log\frac{1}{\delta}}\right). \tag{8.39}$$

The VC dimensions is thereby a measure of how many points can be divided by a member of the hypothesis set for all possible label combinations of the point. For example, consider three arbitrary points in two dimensions as shown in figure 8.5, and let us consider the hypothesis class of all possible lines in two dimensions. I can always divide the three points under any class membership condition, of which two examples are also shown in the figure. In contrast, it is possible to easily find examples with four points that can not be divided by a line in two dimensions. The VC dimension of lines in two dimensions is hence $VC = 3$.[11]

8.3 SV-Regression and implementation

8.3.1 Support Vector Regression

While we have mainly discussed classification in the last few sections, it is time to consider the more general case of regression and to connect these methods to the general principle of maximum likelihood estimation outlined in the previous chapter. It is again easy to illustrate the method for the linear case before generalizing it to the non-linear case similar to the strategy followed for SVMs.

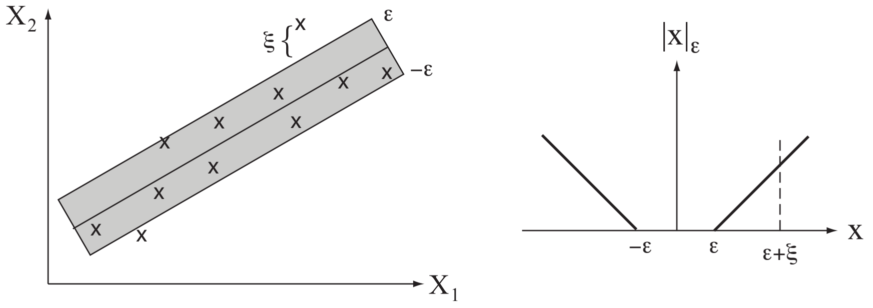


Fig. 8.6 Illustration of support vector regression and the ϵ-insensitive cost function.

We have already mentioned in section 5.4 the ϵ-insensitive error function which does not count deviations of data from the hypothesis that are less than ϵ form the hypothesis, This is illustrated in figure 8.6. The corresponding optimization problem is

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C\sum_i(\xi_i + \xi^*), \tag{8.40}$$

[11] Three points of different classes can not be separated by a single line, but these are singular points that are not effective in the definition of VC dimension.

subject to the constrains

$$y^{(i)} - \mathbf{w}^T\mathbf{x} - b \leq \xi_i \tag{8.41}$$

$$y^{(i)} - \mathbf{w}^T\mathbf{x} - b \geq \xi_i^* \tag{8.42}$$

$$\xi_i, \xi_i^* \geq 0 \tag{8.43}$$

The dual formulations does again only depend on scalar products between the training examples, and the regression line can be also be expressed by a scalar product between the support vectors and the prediction vector,

$$h(\mathbf{x}; \alpha_i, \alpha_i^*) = \sum_{i=1}^{m} (\alpha_i - \alpha_i^*)\mathbf{x}_i^T\mathbf{x}. \tag{8.44}$$

This, we can again use Kernels to generalize the method to non-linear cases.

8.3.2 Implementation

There are several SVM implementations available, and SVMs are finally becoming a standard component of data mining tools. We will use the implementation called LIBSVM which was written by Chih-Chung Chang and Chih-Jen Lin and has interfaces to many computer languages including Matlab. There are basically two functions that you need to use, namely `model=svmtrain(y,x,options)` and `svmpredict(y,x,model,options)`. The vectors `x` and `y` are the training data or the data to be tested. `svmtrain` uses k-fold cross validation to train and evaluate the SVM and returns the trained machine in the structure `model`. The function `svmpredict` uses this model to evaluate the new data points. Below is a list of options that shows the implemented SVM variants. We have mainly discussed C-SVC for the basic soft support vector classification, and epsilonSVR for support vector regression.

```
-s svm_type : set type of SVM (default 0)
0 -- C-SVC
1 -- nu-SVC
2 -- one-class SVM
3 -- epsilon-SVR
4 -- nu-SVR
-t kernel_type : set type of kernel function (default 2)
0 -- linear: u'*v
1 -- polynomial: (gamma*u'*v + coef0)^degree
2 -- radial basis function: exp(-gamma*|u-v|^2)
3 -- sigmoid: tanh(gamma*u'*v + coef0)
-d degree : set degree in kernel function (default 3)
-g gamma : set gamma in kernel function (default 1/num_features)
-r coef0 : set coef0 in kernel function (default 0)
-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 1)
-n nu : set the parameter nu of nu-SVC, one-class SVM, and nu-SVR (default 0.5)
-p epsilon : set the epsilon in loss function of epsilon-SVR (default 0.1)
-m cachesize : set cache memory size in MB (default 100)
```

```
-e epsilon : set tolerance of termination criterion (default 0.001)
-h shrinking: whether to use the shrinking heuristics, 0 or 1 (default 1)
-b probability_estimates: whether to train a SVC or SVR model for probability estimates, 0 o
-wi weight: set the parameter C of class i to weight*C, for C-SVC (default 1)

The k in the -g option means the number of attributes in the input data.
```

8.4 Supervised Line-following

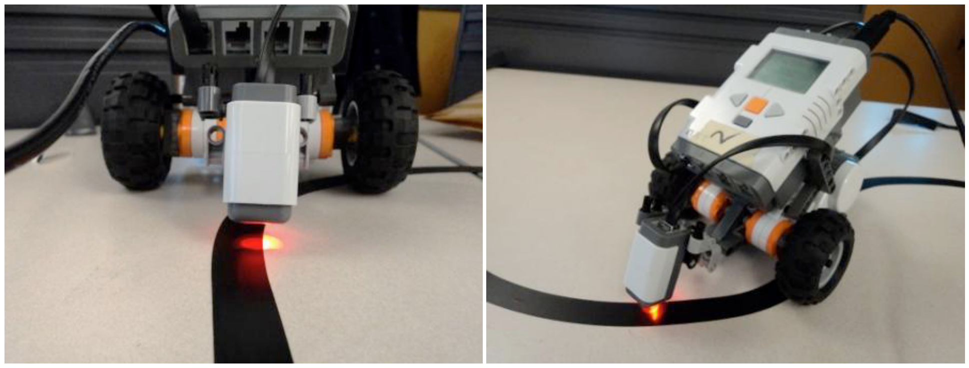
8.4.1 Objective

The objective of this experiment is to investigate Supervised Learning through teaching a robot how to follow a line using a Support Vector Machine (SVM).

8.4.2 Setup

- Mount light sensor on front of NXT, plugged into Port 1
- Use a piece of dark tape (i.e. electrical tape) to mark a track on a flat surface. Make sure the tape and the surface are coloured differently enough that the light sensor returns reasonably different values between the two surfaces.

- This program requires a MATLAB extension that can use Support Vector Machines. Download:



8.4.3 Program

Data collection requires the user to manually move the wheels of the NXT. When the training begins, start the NXT so the light sensor's beam is either on the tape or the surface. Zig zag the NXT so the beam travels on and off the tape by moving either the right or the left wheel, one at a time. Record the positions of the left and right wheels, as well as the light sensor's reading during frequent intervals. It is important to make sure the wheel not in motion stays as stationary as possible to obtain the optimal training set of data.

After data collection, find the difference between the right and the left wheel positions for each time sample taken, and use the SVM to create a model between these differences and the light sensor readings. For instance:

```
model = svmtrain(delta,lightReading,'-s 8 -g 0 -b 1 -e 0.1 -q');
```

To implement the model, place the NXT on the line and use SVMPredict to input a light sensor reading and drive the robot left or right depending on the returned value of the SVMPredict.

```
    lightVal=GetLight(SENSOR_1);
    if svmpredict(0,lightVal,model,'0')>0
        left.Stop();
        right.SendToNXT();
    else
        right.Stop();
        left.SendToNXT();
    end
```

Part III

Unsupervised learning

9 Unsupervised learning

In the previous learning problems we had training examples with feature vectors $\mathbf{x}$ and labels $\mathbf{y}$. In this chapter we discuss unsupervised learning problems in which no labels are given. The luck of training labeled examples restricts the type of learning that can be done, but unsupervised has important applications and even can be an important part in aiding supervised learning. Unsupervised does not mean that the learning is not guided at all; the learning follows specific principles that are used to organize the system based on the characteristics provided by the data. We will discuss several examples in this chapter.

9.1 K-means clustering

The first example is **data clustering**. In this problem domain we are given unlabelled data described by feature and asked to put them into k categories. In the first example of such clustering we categories the data by proximity to a mean value. That is, we assume a model that specifies a mean feature value of the data and classifies the data based on the proximity to the mean value. Of course, we do not know this mean value for each class. The idea of the following algorithm is that we start with a guess for this mean value and label the data accordingly. We then use the labeled data from this hypothesis to improve the model by calculating a new mean value, and repeat these steps until convergence is reached. Such an algorithm usually converges quickly to a stable solution. More formally, given a training set of data points $\{x^{(1)}, x^{(2)}, ..., x^{(m)}\}$ and a hypothesis of the number of clusters, k, the k-means clustering algorithm is shown in Figure 9.1.

```
1. Initialize the means $\mu_1, ...\mu_k$ randomly.
2. Repeat until convergence: {
        **Model prediction:**
            For each data point $i$, classify data to class with closest mean
                $c^{(i)} = arg\min_j ||x^{(i)} - \mu_j||$
        **Model refinement:**
            Calculate new means for each class
                $\mu_j = \frac{1 \ 1(c^{(i)}=j)x^{(i)}}{1 \ 1(c^{(i)}=j)}$
} convergence
```

Fig. 9.1 k-means clustering algorithm

An example is shown in Figure **??**. The corresponding program is shown is

```
%% Demo of k-mean clustering on Gaussian data
```

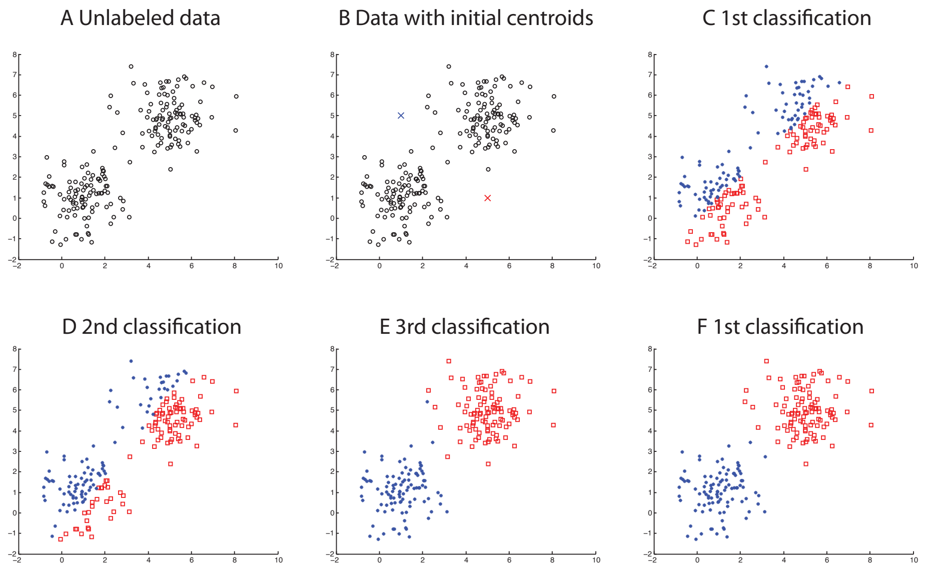


Fig. 9.2 Example of k-means clustering with two clusters.

```
%  Thomas Trappenberg, March 09
clear; clf; hold on;

%% training data generation; 2 classes, each gaussian with mean (1,1) and (2,2) and diagonal
n0=100; %number of points in class 0
n1=100; %number of points in class 1

x=[1+randn(n0,1), 1+randn(n0,1); ...
   5+randn(n1,1), 5+randn(n1,1)];

% plotting points
plot(x(:,1),x(:,2),'ko');

%two centers
mu1=[5 1]; mu2=[1 5];

while(true)
waitforbuttonpress;


plot(mu1(1),mu1(2),'rx','MarkerSize',12)
plot(mu2(1),mu2(2),'bx','MarkerSize',12)

for i=1:n0+n1;
    d1=(x(i,1)-mu1(1))^2+(x(i,2)-mu1(2))^2;
    d2=(x(i,1)-mu2(1))^2+(x(i,2)-mu2(2))^2;
    y(i)=(d1<d2)*1;
end

waitforbuttonpress;

x1=x(y>0.5,:);
x2=x(y<0.5,:);

clf; hold on;


plot(x1(:,1),x1(:,2),'rs');
plot(x2(:,1),x2(:,2),'b*');


mu1=mean(x1);
mu2=mean(x2);

end
```

9.2 Mixture of Gaussian and the EM algorithm

We have previously discussed generative models where we assumed specific models for the in-cass distributions. In particular, we have discussed linear discriminant analysis where we had labelled data and assumed that each class is Gaussian distributed. Here we assume that we have k Gaussian classes, where each class is chosen randomly from a multinominal distribution,

$$z^{(i)} \propto \text{multinomial}(\Phi_j) \tag{9.1}$$

$$x^{(i)}|z^{(i)} \propto N(\mu_j, \Sigma_j) \tag{9.2}$$

This is called a **Gaussian Mixture Model**. The corresponding log-likelihood function is

$$l(\Phi, \mu, \sigma) = \sum_{i=1}^{m} \log \sum_{z^{(i)}=1}^{k} p(x^{(i)}|z^{(i)}; \mu, \Sigma)p(z^{(i)}; \Phi). \tag{9.3}$$

Since we consider here unsupervised learning in which we are given data without labels, the random variables $z^{(i)}$ are latent variables. This makes the problem hard. If we would be give the class membership, than the log-likelihood would be

$$l(\Phi, \mu, \sigma) = \sum_{i=1}^{m} \log p(x^{(i)}; z^{(i)}, \mu, \Sigma), \tag{9.4}$$

which we could use to calculate the maximum likelihood estimates of the parameter (see equations 6.8-6.10),

$$\phi_k = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}(z^{(i)} = j) \tag{9.5}$$

$$\mu_k = \frac{\sum_{i=1}^{m} \mathbb{1}(z^{(i)} = j)\mathbf{x}^{(i)}}{\sum_{i=1}^{m} \mathbb{1}(z^{(i)} = j)} \tag{9.6}$$

$$\Sigma_k = \frac{\sum_{i=1}^{m} \mathbb{1}(z^{(i)} = j)(x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{m} \mathbb{1}(y^{(i)} = k)}. \tag{9.7}$$

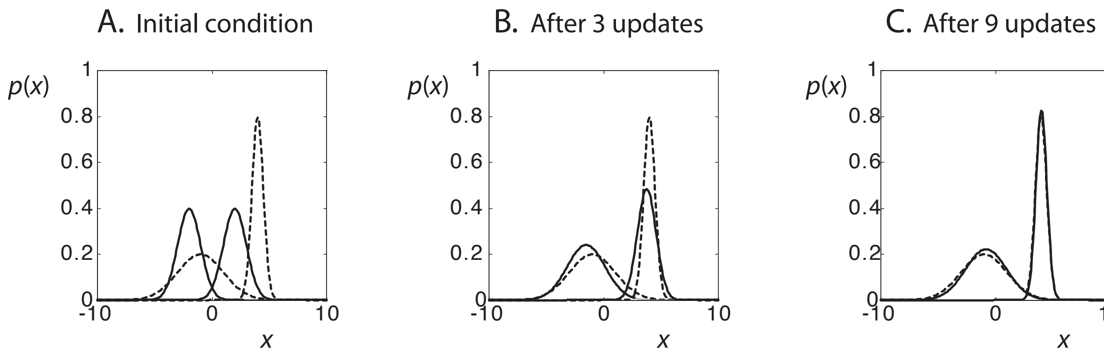
While we do not know the class labels, we can follow a similar strategy to the k-means clustering algorithm and just propose some labels and use them to estimate the parameters. We can then use the new estimate of the distributions to find better labels for the data, and repeat this procedure until a stable configuration is reached. In general, this strategy is called the **EM algorithm** for expectation-maximization. The algorithm is outlined in Fig.9.3. In this version we do not hard classify the data into one or another class, but we take a more soft classification approach that considers the probability estimate of a data point belonging to each class.

```
1. Initialize parameters $\phi, \mu, \Sigma$ randomly.
2. Repeat until convergence: {
        E step:
            For each data point $i$ and class $j$ (soft-)classify data as
                $w_j^{(i)} = p(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma)$
        M step:
            Update the parameters according to
                $\phi_j = \frac{1}{m}\sum_{i=1}^{m} w_j^{(i)}$
                $\mu_j = \frac{\sum_{i=1}^{m} w_j^{(i)} x^{(i)}}{\sum_{i=1}^{m} w_j^{(i)}}$
                $\Sigma_k = \frac{\sum_{i=1}^{m} w_j^{(i)}(x^{(i)}-\mu_j)(x^{(i)}-\mu_j)^T}{\sum_{i=1}^{m} \mathbb{1} w_j^{(i)}}.$
} convergence
```

Fig. 9.3 EM algorithm

An example is shown in Fig. 9.2. In this simple world, data are generated with equal likelihood from two Gaussian distributions, one with mean $\mu_1 = -1$ and standard deviation $\sigma_1 = 2$, the other with mean $\mu_2 = 4$ and standard deviation $\sigma_2 = 0.5$. These two distributions are illustrated in Fig. 9.2A with dashed lines. Let us assume that we know that the world consists only of data from two Gaussian distributions with equal likelihood, but that we do not know the specific realizations (parameters) of these distributions. The pre-knowledge of two Gaussian distributions encodes a specific **hypothesis** which makes up this **heuristic model**. In this simple example, we have chosen the heuristics to match the actual data-generating system (world), that is, we have explicitly used some knowledge of the world.

Learning the parameters of the two Gaussians would be easy if we had access to the information about which data point was produced by which Gaussian, that is, which cause produced the specific examples. Unfortunately, we can only observe the data without a teacher label that could supervise the learning. We choose therefore a



Example of the expectation maximization (EM) algorithm for a world model with two Gaussian distributions. The Gaussian distributions of the world data (input data) are shown with dashed lines. (A) The generative model, shown with solid lines, is initialized with arbitrary parameters. In the EM algorithm, the unlabelled input data are labelled with a recognition model, which is, in this example, the inverse of the generative model. These labelled data are then used for parameter estimation of the generative model. The results of learning are shown in (B) after three iterations, and in (C) after nine iterations .

self-supervised strategy, which repeats the following two steps until convergence:

E-step: We make assumptions of training labels (or the probability that the data were produced by a specific cause) from the current model (expectation step); and

M-step: use this hypothesis to update the parameters of the model to maximize the observations (maximization step).

Since we do not know appropriate parameters yet, we just chose some arbitrary values as the starting point. In the example shown in Fig. 9.2A we used $\mu_1 = 2$, $\mu_2 = -2$, $\sigma_1 = \sigma_2 = 1$. These distributions are shown with solid lines. Comparing the generated data with the environmental data corresponds to hypothesis testing.

The results are not yet very satisfactory, but we can use the generative model to express our **expectation** of the data. Specifically, we can assign each data point to the class which produces the larger probability within the current world model. Thus, we are using our specific hypothesis here as a **recognition model**. In the example we can use Bayes' rule to invert the generative model into a recognition model as detailed in the simulation section below. If this inversion is not possible, then we can introduce a separate recognition model, Q, to approximate the inverse of the generative model. Such a recognition model can be learned with similar methods and interleaved with the generative model.

Of course, the recognition with the recognition model early in learning is not expected to be exact, but estimation of new parameters from the recognized data in the M-step to maximize the expectation can be expected to be better than the model with the initial arbitrary values. The new model can then be compared to the data again and, when necessary, be used to generate new expectations from which the model is refined. This procedure is known as the **expectation maximization** (EM) algorithm. The distributions after three and nine such iterations, where we have chosen new data

Table 9.1 Program ExpectationMaximization.m

```
 1   %% 1d example EM algorithm
 2    clear; hold on; x0=-10:0.1:10;
 3    var1=1; var2=1; mu1=-2; mu2=2;
 4    normal= @(x,mu,var) exp(-(x-mu).^2/(2*var))/sqrt(2*pi*var);
 5    while 1
 6   %%plot distribution
 7       clf; hold on;
 8       plot(x0, normal(x0,-1,4),'k:');
 9       plot(x0, normal(x0,4,.25),'k:');
10       plot(x0, normal(x0,mu1,var1),'r');
11       plot(x0, normal(x0,mu2,var2),'b');
12       waitforbuttonpress;
13   %% data
14       x=[2*randn(50,1)-1;0.5*randn(50,1)+4;];
15   %% recogintion
16       c=normal(x,mu1,var1)>normal(x,mu2,var2);
17   %% maximization
18       mu1=sum(x(c>0.5))/sum(c);
19       var1=sum((x(c>0.5)-mu1).^2)/sum(c);
20       mu2=sum(x(c<0.5))/(100-sum(c));
21       var2=sum((x(c<0.5)-mu2).^2)/(100-sum(c));
22    end
```

points in each iteration, are shown in Figs 9.2B and C.

Simulation

The program used to produce Fig. 9.2 is shown in Table 9.1. The vector x_0, defined in Line 2, is used to plot the distributions later in the program. The arbitrary random initial conditions of the distribution parameters are set in Line 3. Line 4 defines an **inline function** of a properly normalized Gaussian since this function is used several times in the program. An inline function is an alternative to writing a separate function file. It defines the name of the functions, followed by a list of parameters and an expression, as shown in Line 4. The rest of the program consist of an infinite loop produced with the statement `while 1`, which is always true. The program has thus to be interrupted by closing the figure window or with the interruption command `Ctrl C`. In Lines 7–12, we produce plots of the real-world models (dotted lines) and the model distributions (plotted with a red and a blue curve when running the program). The command `waitforbuttonpress` is used in Line 12 so that we can see the results after each iteration.

In Line 14 we produce new random data in each iteration. Recognition of this data is done in Line 16 by inverting the generative model using Bayes' formula,

$$P(c|\mathbf{x};G) = \frac{P(\mathbf{x}|c;G)P(c;G)}{P(\mathbf{x};G)}. \tag{9.8}$$

In this specific example, we know that the data are equally distributed from each Gaussian so that the **prior distribution over causes**, $P(c; G)$ is $1/2$ for each cause. Also, the **marginal distribution of data** is equally distributed, so that we can ignore this normalizing factor. The recognition model in Line 16 uses the Bayesian decision criterion, in which the data point is assigned to the cause with a larger **recognition distribution**, $P(c|\mathbf{x}; G)$. Using the labels of the data generated by the recognition model, we can then use the data to obtain new estimates of the parameters for each Gaussian in Lines 17–21.

Note that when testing the system for a long time, it can happen that one of the distributions is dominating the recognition model so that only data from one distribution are generated. The model of one Gaussian would then be **explaining away** data from the other cause. More practical solutions must take such factors into account.

9.3 The Boltzmann machine

9.3.1 General one-layer module

Our last model that uses unsupervised learning is again a general learning machine invented by Geoffrey Hinton and Terrance Sejnowski in the mid 1980 called **Boltzmann machine**. This machine is a general form of a recurrent neural network with **visible nodes** that receive input or provide output, and **hidden notes** that are not connected to the outside world directly. Such a stochastic dynamic network, a recurrent system with hidden nodes, together with the adjustable connections, provide the system with enough degrees of freedom to approximate any dynamical system. While this has been recognized for a long time, finding practical training rules for such systems have been a major challenge for which there was only recently major progress. These machines use unsupervised learning to learn hierarchical representations based on the statistics of the world. Such representations are key to more advanced applications of machine learning and to human abilities.

The basic building block is a one-layer network with one visible layer and one hidden layer. An example of such a network is shown in Fig. 9.4. The nodes represent

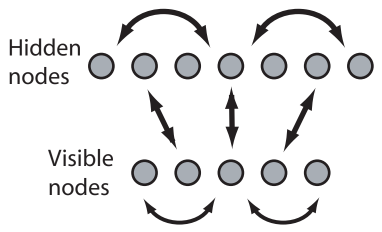


Fig. 9.4 A Boltzmann machine with one visible and one hidden layer.

random variable similar to the Bayesian networks discussed before. We will specifically consider binary nodes that mimic neuronal states which are either firing or not. The connections between the have weights w_{ij} which specify how much they influence the on-state of connected nodes. Such systems can be described by an energy function. The energy between two nodes that are symmetrically connected with strength w_{ij} is

$$H^{nm} = -\frac{1}{2}\sum_{ij} w_{ij} s_i^n s_j^m. \tag{9.9}$$

The state variables, s, have superscripts n or m which can have values (v) or (h) to indicate visible and hidden nodes. We consider again the probabilistic update rule,

$$p(s_i^n = +1) = \frac{1}{1 + \exp(-\beta \sum_j w_{ij} s_j^n)}, \tag{9.10}$$

with inverse temperature, β, which is called the Glauber dynamics in physics and describes the competitive interaction between minimizing the energy and the randomizing thermal force. The probability distribution for such a stochastic system is called the Boltzmann–Gibbs distribution. Following this distribution, the distribution of visible states, in thermal equilibrium, is given by

$$p(\mathbf{s}^v; \mathbf{w}) = \frac{1}{Z}\sum_{m \in h} \exp(-\beta H^{vm}), \tag{9.11}$$

where we summed over all hidden states. In other words, this function describes the distribution of visible states of a Boltzmann machine with specific parameters, $\mathbf{w}$, representing the weights of the recurrent network. The normalization term, $Z = \sum_{n,m} \exp(-\beta H^{nm})$, is called the **partition function**, which provides the correct normalization so that the sum of the probabilities of all states sums to one. These stochastic networks with symmetrical connections have been termed **Boltzmann machines** by **Ackley**, **Hinton** and **Sejnowski**.

Let us consider the case where we have chosen enough hidden nodes so that the system can, given the right weight values, implement a generative model of a given world. Thus, by choosing the right weight values, we want this dynamical system to approximate the probability function, $p(\mathbf{s}^v)$, of the sensory states (states of visible nodes) caused by the environment. To derive a learning rule, we need to define an objective function. In this case, we want to minimize the difference between two density functions. A common measure for the difference between two probabilistic distributions is the Kulbach–Leibler divergence (see Appendix 4.6),

$$\mathrm{KL}(p(\mathbf{s}^v), p(\mathbf{s}^v; \mathbf{w})) = \sum_{\mathbf{s}}^{v} p(\mathbf{s}^v) \log \frac{p(\mathbf{s}^v)}{p(\mathbf{s}^v; \mathbf{w})} \tag{9.12}$$

$$= \sum_{\mathbf{s}}^{v} p(\mathbf{s}^v) \log p(\mathbf{s}^v) - \sum_{\mathbf{s}}^{v} p(\mathbf{s}^v) \log p(\mathbf{s}^v; \mathbf{w}). \tag{9.13}$$

To minimize this divergence with a gradient method, we need to calculate the derivative of this 'distance measure' with respect to the weights. The first term in the difference in eqn 9.13 is the entropy (see Appendix **??**) of sensory states, which does not depend on the weights of the Boltzmann machine. Minimizing the Kulbach–Leibler divergence is therefore equivalent to maximizing the average log-likelihood function,

$$l(\mathbf{w}) = \sum_{\mathbf{s}}^{v} p(\mathbf{s}^v) \log p(\mathbf{s}^v; \mathbf{w}) = \langle \log p(\mathbf{s}^v; \mathbf{w}) \rangle. \tag{9.14}$$

In other words, we treat the probability distribution produced by the Boltzmann machine as a function of the parameters, w_ij, and choose the parameters which maximize the likelihood of the training data (the actual world states). Therefore, the averages of the model are evaluated over actual visible states generated by the environment. The log-likelihood of the model increases the better the model approximates the world. A standard method of maximizing this function is gradient ascent, for which we need to calculate the derivative of $l(\mathbf{w})$ with respect to the weights. We omit the detailed derivation here, but we note that the resulting learning rule can be written in the form

$$\Delta w_{ij} = \eta \frac{\partial l}{\partial w_{ij}} = \eta \frac{\beta}{2} \left(\langle s_i s_j \rangle_{\text{clamped}} - \langle s_i s_j \rangle_{\text{free}} \right). \tag{9.15}$$

The meaning of the terms on the right-hand side is as follows. The term labelled 'clamped' is the thermal average of the correlation between two nodes when the states of the visible nodes are fixed. The termed labelled 'free' is the thermal average when the recurrent system is running freely. The Boltzmann machine can thus be trained, in principle, to represent any arbitrary density functions, given that the network has a sufficient number of hidden nodes.

This result is encouraging as it gives as an exact algorithm to train general recurrent networks to approximate arbitrary density functions. The learning rule looks interesting since the clamped phase could be associated with a sensory driven agent during an awake state, whereas the freely running state could be associated with a sleep phase. Unfortunately, it turns out that this learning rule is too demanding in practice. The reason for this is that the averages, indicated by the angular brackets in eqn 9.15, have to be evaluated at thermal equilibrium. Thus, after applying each sensory state, the system has to run for a long time to minimize the initial transient response of the system. The same has to be done for the freely running phase. Even when the system reaches equilibrium, it has to be sampled for a long time to allow sufficient accuracy of the averages so that the difference of the two terms is meaningful. Further, the applicability of the gradient method can be questioned since such methods are even problematic in recurrent systems without hidden states since small changes of system parameters (weights) can trigger large changes in the dynamics of the dynamical systems. These problems prevented, until recently, more practical progress in this area. Recently, Hinton and colleagues developed more practical, and biologically more plausible, systems which are described next.

9.3.2 The restricted Boltzmann machine and contrastive Hebbian learning

Training of the Boltzmann machine with the above rule is challenging because the states of the nodes are always changing. Even with the visible states clamped, the

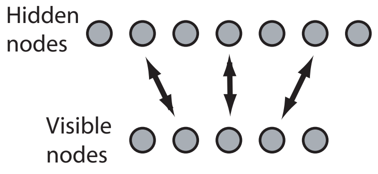


Fig. 9.5 Restricted Boltzmann machine in which recurrences within each later are removed.

states of the hidden nodes are continuously changing for two reasons. First, the update rule is probabilistic, which means that even with constant activity of the visible nodes, hidden nodes receive variable input. Second, the recurrent connections between hidden nodes can change the states of the hidden nodes rapidly and generate rich dynamics in the system. We certainly want to keep the probabilistic update rule since we need to generate different responses of the system in response to sensory data. However, we can simplify the system by eliminating recurrent connections within each layer, although connections between the layers are still bidirectional. While the simplification of omitting collateral connections is potentially severe, much of the abilities of general recurrent networks with hidden nodes can be recovered through the use of many layers which bring back indirect recurrencies. A **restricted Boltzmann machine** (RBM) is shown in Fig. 9.5.

When applying the learning rule of eqn 9.15 to one layer of an RBM, we can expect faster convergence of the rule due to the restricted dynamics in the hidden layer. We can also write the learning rule in a slightly different form by using the following procedure. A sensory input state is applied to the input layer, which triggers some probabilistic recognition in the hidden layer. The states of the visible and hidden nodes can then be used to update the expectation value of the correlation between these nodes, $\langle s_i^v s_j^h \rangle^0$, at the initial time step. The pattern in the hidden layer can then be used to approximately reconstruct the pattern of visible nodes. This **alternating Gibbs sampling** is illustrated in Fig. 9.6 for a connection between one visible node and one hidden node, although this learning can be done in parallel for all connections. The learning rule can then be written in form,

$$\Delta w_{ij} \propto \langle s_i^v s_j^h \rangle^0 - \langle s_i^v s_j^h \rangle^\infty. \tag{9.16}$$

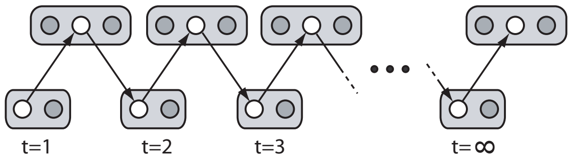


Fig. 9.6 Alternating Gibbs sampling.

Alternating Gibbs sampling becomes equivalent to the Boltzmann machine learning rule (eqn 9.15) when repeating this procedure for an infinite number of time steps, at which point it produces pure fantasies. However, this procedure still requires averaging over long sequences of simulated network activities, and sufficient evaluations of thermal averages can still take a long time. Also, the learning rule of eqn 9.16 does not seem to correspond to biological learning. While developmental learning also takes some time, it does not seems reasonable that the brain produces and evaluates long sequences of responses to individual sensory stimulations. Instead, it seems more reasonable to allow some finite number of alternations between hidden responses and the reconstruction of sensory states. While this does not formally correspond to the mathematically derived gradient leaning rule, it is an important step in solving the learning problem for practical problems, which is a form of **contrastive divergence** introduced by Geoffrey Hinton. It is heuristically clear that such a restricted training procedure can work. In each step we create only a rough approximation of ideal average fantasies, but the system learns the environment from many examples, so that it continuously improves its expectations. While it might be reasonable to use initially longer sequences, as infants might do, Hinton and colleagues showed that learning with only a few reconstructions is able to self-organize the system. The self-organization, which is based on input from the environment, is able to form internal representations that can be used to generate reasonable sensory expectations and which can also be used to recognize learned and novel sensory patterns.

The basic Bolzmann machine with a visible and hidden layer can easily be combined into hierarchical networks by using the activities of hidden nodes in one layer as inputs to the next layer. Hinton and colleagues have demonstrated the power of restricted Boltzmann machines for a number of examples. For example, they applied layered RBMs as auto-encoders where restricted alternating Gibbs sampling was used as pre-training to find appropriate initial internal representations that could be fine-tuned with backpropagation techniques to yield results surpassing support vector machines. However, for our discussions of brain functions it is not even necessary to yield perfect solutions in a machine learning sense, and machines can indeed outperform humans in some classification tasks solved by machine learning methods. For us, it is more important to understand how the brain works.

Simulation 1: Hinton

To illustrate the function of an anticipating brain model, we briefly outline a demonstration by the Hinton group. The online demonstration can be run in a browser from `http://www.cs.toronto.edu/~hinton/adi`, and a stand alone version of this demonstration is available at this book's resource page. MATLAB source code for restricted Boltzmann machines are available at Hinton's home page. An image of the demonstration program is shown in Fig. 9.7. The model consists of a combination of restricted Boltzmann machines and a Helmholz machine. The input layer is called the **model retina** in the figure, and the system also contains a **recognition-readout-and-stimulation** layer. The model retina is used to apply images of handwritten characters to the system. The recognition-readout-and-stimulation layer is a brain imaging and stimulation device from and to the uppermost RBM layer. This device is trained by providing labels as inputs to the RBM for the purpose of 'reading the mind' of the system and to give it high-level instructions. This device learns to recognize patterns in the uppermost layer and map them to their meaning, as supplied during supervised learning of this device. This is somewhat analogous to **brain–computer interfaces** developed with different brain-imaging devices such as EEG, fMRI, or implanted electrodes. The advantage of the simulated device is that it can read the activity of every neuron in the upper RBM layer. The device can also be used with the learned connections in the opposite direction to stimulate the upper RBM layer with typical patterns for certain image categories.

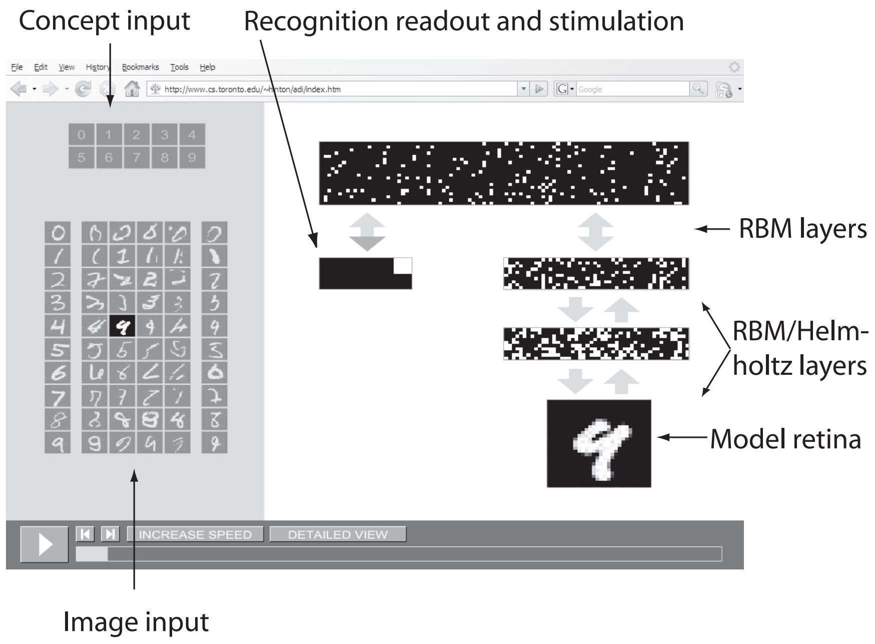


Fig. 9.7 Simulation of restricted Boltzmann machine by Geoffrey Hinton and colleagues, available at `www.cs.toronto.edu/∼hinton/adi`.

The model for this demonstration was trained on images of handwritten numbers from a large database. Some example images can be seen on the left-hand side. All layers of this model were first treated as RBMs with symmetrical weights. Specifically, these were trained by applying images of handwritten characters to the model retina and using three steps of alternating Gibbs sampling for training the different layers. The evolving representations in each layer are thus purely unsupervised. After this basic training, the model was allowed, for fine-tuning purposes, to develop different weight values for the recognition and generative model as in Helmholtz machines with a wake–sleep training algorithm as mentioned above.

The simulations provided by Hinton demonstrate the ability of the system after training. The system can be tested in two ways, either by supplying a handwritten image and asking for recognition, or by asking the system to produce images of a certain letter. These two modes can be initiated by selecting either an image or by selecting a letter category on the left-hand side. In the example shown in Fig. 9.7, we selected an example of an image of the number 4. When running the simulation, this image triggers response patterns in the layers. These patterns change in every time step, due to the probabilistic nature of the updating rule. The recognition read-out of the uppermost layer does, therefore, also fluctuate. In the shown example, the response of the system is 4, but the letter 9 is also frequently reported. This makes sense, as this image does also look somewhat like the letter 9. A histogram of responses can be constructed when counting the responses over time, which, when properly normalized, corresponds to an estimate of the probability over high-level concepts generated by this sensory state. Thus, this mode tests the recognition ability of the model.

The stimulation device connected to the upper RBM layer allows us to instruct the system to 'visualize' specific letters, which corresponds to testing the generative ability of the model. For example, if we ask the system to visualize a letter 4 by evoking corresponding patterns in the upper layer, the system responds with varying images on the model retina. There is not a single right answer, and the answers of the system change with time. In this way, the system produces examples of possible images of letter 4, proportional to some likelihood that these images are encountered in the sensory world on which the system was trained. The probabilistic nature of the system much better resembles human abilities to produce a variety of responses, in contrast to the neural networks that have been popular in the 1980s, so called multilayer perceptrons, which were only able to produce single answers for each input.

Simulation 2: Simplified one layer model

A simplified version of the RBM trained on some letters are included in folder `RBM example` on the web resource page. The overage reconstruction error and some examples of reconstructions after training are shown in Fig.9.8.

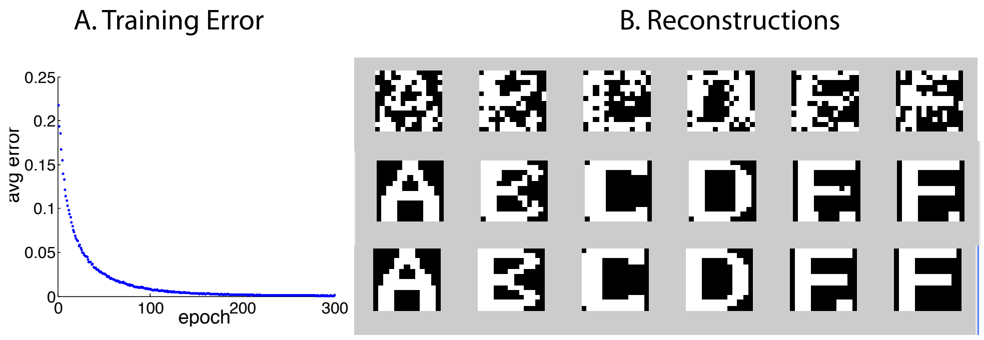


Fig. 9.8 (A) Reconstruction error during training of alphabet letters the letters, and (B) reconstructions after learning.

Part IV

Reinforcement learning

10 Reinforcement learning

This chapter is an introduction to reinforcement learning, specifically in terms of the Markov decision process (MDP) and temporal difference (TD) learning.

10.1 Learning from reward and the credit assignment problem

We discussed in previous chapters supervised learning and unsupervised learning. These types of learning are based on detailed examples (feature vectors of training instances), and, in case of supervised learning, also on a teacher signal that tells the machine exactly how to respond to this input. Reinforcement learning is about learning appropriate actions from reward feedback. The reward feedback does not tell the agent directly which action to take. Rather, it indicates which states are desirable (rewarded) or not (punished). The agent has to discover the right sequence of actions to take to optimize the reward over time.

Reward learning introduces several challenges. For example, in typical circumstances reward is only received after a long sequence of actions. The problem is then how to assign the credit for the reward to specific actions. This is the **temporal credit assignment problem**. To illustrate this, let us think about a car that crashed into a wall. It is likely that the driver used the breaks before the car crashed into the wall, though the breaks could not prevent the accident. However, from this we should not conclude that breaking is not good and lead to crashes.

Another challenge in reinforcement learning is the balance between **exploitation** and **exploration**. That is, we might find a way to receive some small food reward if we repeat certain actions, but if we only repeat these specific actions, we might never discover a bigger reward following different actions. Some escape from self-reinforcement is important.

The idea of reinforcement learning is to use the reward feedback to build up a **value function** that reflect the expected future payoff. We can use such a value function to make decisions of which action to take. This is called a **policy**. This will be formalized in this chapter. We start thereby with simple processes where the transitions to new states depend only on the current state. A process which such a characteristics is called a **Markov process**. In addition to the Markov property, we also assume at first that the agent has full knowledge its environment. Finally, it is again important that we acknowledge and uncertainties and possible errors. For example, we can take error in motor commands into account by considering state transition as probabilistic.

10.2 The Markov Decision Process

A Markov Decision Process (MDP) is characterized with a set of 5 quantities $(S, A, P_{sa}(s'), R(s), \xi)$. More specifically,

- S is a set of states
- A is a set of actions. We also use a function called **policy** that maps specific states to specific actions, $\pi(s)$.
- $P_{sa}(s') = P_{\pi(s)}(s')$ is a **transition probability** for ending up in state s' when taking action a from state s. This transition probability only depend on the previous state, which is called the Markov condition; hence the name of the process.
- $R(s)$ is a **reward function**, which provides feedback from the environment. R is a numeric value with positive values providing reward and negative values relating to punishment.
- ξ are specific parameters for some of the different kinds of RL settings. This will be the **discount factor** γ in our first examples.

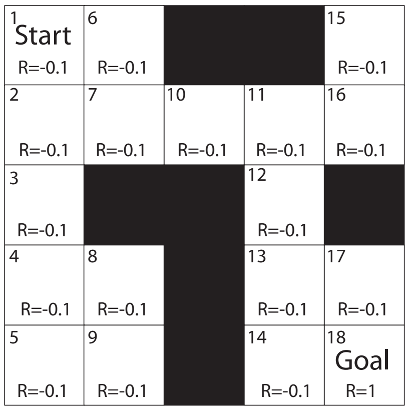


Fig. 10.1 A maze where each state is rewarded with a value R.

For example, let us consider an agent that should learn to navigate through a maze. An example is shown in Figure 10.1. The states of the maze are the possible discrete positions (x and y coordinates or simply given unique numbers) of location in the maze. In the shown examples the state space has 18 states. The possible actions of the agent is to move one step forward, either to the north, east, south and west, e.g. $A = \{N, E, S, W\}$. However, even though the agents gives these commands to its actuators, stochastic circumstances – such as faulty hardware or environmental conditions (e.g. someone 'kicking' the agent) – make the agent end up in different states with certain probabilities specified by $P_{sa}(s')$. More precisely, if the agent is in state s and is instructed to take action a, it will end up in state s' with probability $P_{sa}(s')$. We assume for now that the transition probability is given explicitly, although in many practical circumstances it might need to be estimated from examples (e.g. supervised learning). Finally, the agent is given immediate reward $R(s)$ when it is in state s. For example, to agent should be given a large reward when finding the exit to the maze ($R(18) = 1$ in the example of Figure 10.1). In practice it is also common and useful to give some small negative reward to the other states. This could, for example, stand for the battery power the Lego robot uses when it is not in the final state, whereas it gets recharged at the exit of the maze.

The goal of reinforcement learning is to learn which action to take in each state in order to optimize the expected **total payoff**. The total payoff is the sum of all future reward, such as

$$R(s_1) + R(s_2) + R(s_3) + R(s_4) + ... \tag{10.1}$$

when going through states $s_1, s_2, s_3, ...$ One problem with this definition is that it is an infinite quantity as it runs over infinitely many states into the future. One possible solution of this problem is to restrict the sum by considering only a **finite reward horizon**, for example by only consider reward given within a certain time period or a finite number of steps such as

$$R(s_1) + R(s_2) + R(s_3) + R(s_4). \tag{10.2}$$

Another way to solve the infinite payoff problem is to consider reward that is discounted when it is given at later times. In particular, if we consider the discount factor $0 < \gamma < 1$ for each step, we have a total payoff of

$$R(s_1) + \gamma R(s_2) + \gamma^2 R(s_3) + \gamma^3 R(s_4) + ... \tag{10.3}$$

Of course, we don't know this quantity and our task is hence to estimate this quantity from our knowledge of the environment. More specifically, we want to estimate the expected value of the total payoff when starting at state s_1 and taking actions according to a policy $\pi(s)$. The policy is a function which tells the agent at each state s which action a to take from this particular state. Thus, the expected total payoff depends on the starting state and the specific policy function. When writing the expected value as a function $E(...)$, the expected discounted total payoff from state s_1 when following policy π can be expressed more formally as

$$V^\pi(s) = E(R(s_1) + \gamma R(s_2) + \gamma^2 R(s_3) + \gamma^3 R(s_4) + ...|s_1 = s, \pi). \tag{10.4}$$

This is called the **state-value-function**. Our main goal in reinforcement learning is to find or estimate this value function. Once we have a good estimate of this function, we can use this function to initiate actions that will yield good payoff.

Note that we can use path planing, for example with the A^* search algorithm, to find a path to the goal, especially if we have complete knowledge of the system. However, the task here is different in that the agent must discover itself the task of completing the maze. Indeed, the agent might not even be aware of this as the main task for the agent is simply to optimize future reward. What is the benefit of this approach? The great thing about reinforcement learning is that it is very general and can readily be applied to many task. Also, being a learning system, we can even change the task at any point by changing the reward feedback, and there should be no need to change anything in the program of the agent. Indeed, when training animals, this is usually the main way that we can communicate with the animals in learning situations since we can not verbally communicate the task goal.

10.3 The Bellman equation

It is possible to estimate this value function from a self-consistent equation as introduced by Richard Bellman in the mid 1950s. This method is also known as **dynamic programming**. To derive the Bellman equation, we start with the state-value-function

$$V^{\pi}(s) = E(R(s_1) + \gamma R(s_2) + \gamma^2 R(s_3) + \gamma^3 R(s_4) + ...|s_1 = s, \pi). \qquad (10.5)$$

Since the expected value of the immediate reward is equal to the immediate reward, $E(R(s_1)) = R(s)$, the above equation is equivalent to

$$V^{\pi}(s) = R(s) + \gamma E(R(s_2) + \gamma R(s_3) + \gamma^2 R(s_4) + ...|s_1 = s, \pi). \qquad (10.6)$$

The expected value in the last equation looks very similar to the state-value-function itself, but it is starting at state s_2 instated of state s_1. The states s_2 are the states that can be reached from state s_1 by one step. Thus we can incorporate this into the equation by

$$V^{\pi}(s) = R(s) + \gamma \sum_{s'} P_{\pi(s)}(s') E(R(s_2) + \gamma R(s_3) + \gamma^2 R(s_4) + ...|s_2 = s', \pi). \qquad (10.7)$$

In the last expression, the expectation value is the state-value-function of state s'. Thus, if we substitute the corresponding expression of equation 10.5 into the above formula, we get the **Bellman equation**

$$V^{\pi}(s) = R(s) + \gamma \sum_{s'} P_{\pi(s)}(s') V^{\pi}(s'). \qquad (10.8)$$

In an environment with N states, the Bellman equation is a set of N linear equations, one for each state on the left hand side, with N unknowns, the value for state. We can thus use well known methods from linear algebra to solve for $V^{\pi}(s)$. We can also use the Bellman equation directly and calculate state-value-function iteratively for each policy, starting with a guess of $V^{\pi}(s)$ and calculating from this estimate a better approximation until this process converges. This will be demonstrated in more detail in the examples below. The equation above depends on a specific policy. We are mainly interested in finding the policy that gives us the **optimal payoff**,

$$V^{*}(s) = \max_{\pi} V^{\pi}(s). \qquad (10.9)$$

The **optimal policy** is the policy that maximizes the expected reward. The main problem of finding this is the shear number of possible policies that we need to consider in the maximum operation above, which is equal to the number of actions to the power of the number of states. This explosion of the problem size with the number of states was termed **curse of dimensionality** by Richard Bellman and is possibly one of the main challenges in reinforcement learning. Most of the problems discussed here have small number of states and small number of possible actions, so that this is not a major concern here. We will later discuss briefly methods to overcome this problem.

Instead of using the above Bellman equation for the value function and then calculating the optimal value functions, we can also derive a version of Bellman's equation for the optimal value function itself[12]. This is given by

$$V^*(s) = R(s) + \max_a \gamma \sum_{s'} P_{as}(s')V^*(s'). \tag{10.10}$$

Note that this version includes a max function over all possible actions. Finally, given a good approximation of the optimal value functions, we can calculate the **optimal policy**

$$\pi^*(s) = arg \max_a \sum_{s'} P_{as}(s')V^*(s'), \tag{10.11}$$

which should be used by an agent to achieve good performance.

10.4 Different schemes of solving MDPs

To demonstrate different schemes for solving MDPs, we will follow a simple example. In this example we have a chain of N consecutive states $s = 1, 2, ..., N$. An agent has two possible actions, go to lower state numbers ($a = -1$), or go to higher state numbers ($a = +1$). The last state in the chain, state number N, is rewarded, $R(N) = 1$, whereas going to the first state in the chain is punished $R(1) = -1$. The reward of the intermediate states is set to a small negative value, $R(i) = -0.1, 1 < i < N$.

10.4.1 Value iteration

In the following two schemes we assume full knowledge of the environment so that we can easily iterate over all possible states. In the first method, called **value iteration**, we use directly Bellman's equation for the optimal value function, equation 10.10, to calculate increasingly better estimations of the optimal state-value-function by iterations from initially arbitrary values.

```
Choose initial estimate of optimal value function
Repeat until change in values is sufficiently small {
    For each state {
        Calculate the maximum expected value of neigh-  \
          bouring states for each possible action.       |        $V^*$
        Use maximal value of this list to update estimate |  equation 10.10
          of optimal value function.                     /
    } each state
} convergence
Calculate optimal value function from equation 10.11
```

Fig. 10.2 Value Iteration with asynchronous update.

[12]Convince yourself that this is also true

The basic algorithm is outlines in Figure 10.2. This algorithm takes an initial guess of the optimal value function, typically random or all zeros. The then initialize a loop over several episodes until the change of the value function is sufficiently small. For example, we could calculate the sum of value functions in each iteration t, and then terminate the procedure if the absolute difference of consecutive iterations is sufficiently small, that is if $|\sum_s V_t^*(s) - \sum_s V_{t-1}^*(s)|$ <threshold. In each of those iterations, we iterate over all states and update the estimated optimal value functions according to equation 10.10. Thus, the exploration of the state is very simple in this state; the agent just goes repeatedly to every possible state in the system. While this works well in the examples with small state spaces, this will be a major problem when the state space increases.

The state iteration can thereby be done in various ways. For example, in the **sequential asynchronous updating schema** we update each state in sequence and repeat this procedure over several iterations. Small variations of this schema are concerned with how the algorithm iterates over states. For example, instead of iterating sequentially over the states, we could also use a random oder. We could also first calculate the maximum value of neighbours for all states before updating the value function for all states with an **synchronous updating schema**. Since it can be shown that theses procedure will converge to the optimal solution, all these schemas should work similarly well though might differ slightly for particular examples.

Once we found the optimal value functions, we can then calculate the corresponding optimal policy according to equations 10.11. That is, the optimal policy is taking the action that maximizes the expected payoff from the corresponding state.

Exercise:

Implement the value iteration for the chain problem and plot the learning curve (how the error changes over time), the optimal value function, and the optimal policy. Change parameters such as N, γ, and the number of iterations and discuss the results.

10.4.2 Policy iteration

Another strategy to solve the MDP problem is to use the basic Bellman equation for a specific policy, equation 10.8, instead of the Bellman equation for the optimal policy, and than improve the policies from the latest value function. The basic algorithm is outlined in Figure 10.3. In addition to an initial guess of the value function, we have now also to initialize the policy, which could be randomly chosen from the set of possible actions at each state. For this value functions we could then calculate the corresponding state-value-function according to equation 10.8. This step is the evaluating the specific policy. The next step is to take this value functions and calculate the corresponding best set of actions to take accordingly, which corresponds to the next candidate policy. This policy is again simply to take the action from each state that would maximize the corresponding future payoff. These two steps, the policy evaluation and the policy improvement are repeated the policy won't change any more.

This method has some advantages over value iterations. In value iteration we have to try out all possible actions when evaluating the value function, and this can be time consuming when there are many possible actions. In policy iteration, we choose a

```
Choose initial policy and value function
Repeat until policy is stable {
    1. Policy evaluation
    Repeat until change in values is sufficiently small {
        For each state {
            Calculate the value of neighbouring states when taking     }  $V^\pi$
              action according to current policy.                      }  equation 10.8
            Update estimate of optimal value function.                 }
        } each state
    } convergence
    2. Policy improvement
    new policy according to equation 10.11, assuming $V^* \approx$ current $V^\pi$
} policy
```

Fig. 10.3 Policy iteration with asynchronous update.

specific policy, although we have then to iterate over consecutive policies. In practice it turns out that policy iteration often converges fairly rapidly so that it becomes a practical method.

As an example we want to apply policy iteration to the chain problem. We can directly implement the above algorithm for evaluating each policy iteratively using Bellmans equation 10.8, which is very similar to the previous implementation.

Exercise:

Solve the chain problem with the policy iteration using the basic Bellman functions iteratively, and compare this method to the value iteration.

Finally, in the case of complete knowledge of the environment, we can replace the iterative policy evaluation with an explicite solution of the N linear equations as mentioned above. For this calculation it is useful to write the basic Bellman equation 10.8 in matrix notation,

$$\mathbf{M}\mathbf{v}^\pi = \mathbf{R} \tag{10.12}$$

where $\mathbf{M}$ is the matrix of coefficients that are multiplied by the column vector of state values $\mathbf{v}^\pi$ and rewards are represented by the column vector $\mathbf{R}$. For the chain example, where the agent can move either left ($\pi(s) = -1$) or right ($\pi(s) = +1$), the typical equation in this linear system is derived from the Bellman equation as follows:

$$V^\pi(s) = R(s) + \gamma \sum_{s' \in S} P_{\pi(s)s}(s') V^\pi(s') \tag{10.13}$$

$$-R(s) = -V^\pi(s) + \gamma \left[P_{\pi(s)} V^\pi(s + \pi(s)) + (1 - P_{\pi(s)}) V^\pi(s - \pi(s)) \right] \tag{10.14}$$

$$V^{\pi}(s) - \gamma P_{\pi(s)} V^{\pi}(s + \pi(s)) - \gamma(1 - P_{\pi(s)}) V^{\pi}(s - \pi(s)) = R(s) \tag{10.15}$$

This last equation gives coefficients for three states of V^{π}: the current state, s, and the two states on either side. Implicitly, the coefficients for all other states are zero. The matrix form, depicting a state where the policy is to move right, would be

$$\begin{pmatrix} m_{11} & \cdots & & & & & \cdots & m_{1N} \\ \vdots & & & & & & & \vdots \\ 0 & \cdots & 0 & -\gamma(1-P_{\pi(s)}) & 1 & -\gamma P_{\pi(s)} & 0 \cdots & 0 \\ \vdots & & & & & & & \vdots \\ m_{N1} & \cdots & & & & & \cdots & m_{NN} \end{pmatrix} \begin{pmatrix} v_1^{\pi} \\ v_2^{\pi} \\ \vdots \\ v_N^{\pi} \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_N \end{pmatrix} \tag{10.16}$$

Note that ones line the diagonal of the matrix, signifying the $V^{\pi}(s)$ term for each equation except for the first and last state with the reflecting boundary conditions. For the opposite policy (going left at s), the two coefficients on either side of the 1 would be swapped. Thus, the missing coefficients can easily be filled in, given the policy for each state. The linear system can be solved by rearranging equation 10.12,

$$\mathbf{v}^{\pi} = \mathbf{M}^{-1}\mathbf{r}^{\mathbf{T}} \tag{10.17}$$

To implement this solution in Matlab, we have set the following parameters

```
N=8; gamma=0.9; Psa=0.8;
```

whereN is the number of states, gamma is the discounting factor, and Psa is the probability of reaching state s when initiating state a. We need then to initialize the value function and initial policy,

```
Vpi=zeros(1,N); %initial value function
policy=2*floor(rand(1,N)*2)-1; %random initial policy
```

We then need to set up a loop which includes the policy evaluation and policy improvement. The policy evaluation can be done by implementing the Matrix and then using the Matlab function `inv()` to solve the linear eqaution system,

```
%solving Bellman equation
M=diag(ones(1,N));
M(1,1)=1-gamma*((1-policy(1))/2*Psa+(1+policy(1))/2*(1-Psa));
M(1,2)=-gamma*((1-policy(1))/2*(1-Psa)+(1+policy(1))/2*Psa);
for i=2:N-1;
    M(i,i-1)=-gamma*((1-policy(i))/2*Psa+(1+policy(i))/2*(1-Psa));
    M(i,i+1)=-gamma*((1-policy(i))/2*(1-Psa)+(1+policy(i))/2*Psa);
end
M(N,N-1)=-gamma*((1-policy(N))/2*Psa+(1+policy(N))/2*(1-Psa));
M(N,N)=1-gamma*((1-policy(N))/2*(1-Psa)+(1+policy(N))/2*Psa)
Vpi=inv(M)*r';
```

This value function can then be used to find the optimal policy for these specific values, for example like

```
[tmp1,tmp2]=max([Psa*Vstar(1)+(1-Psa)*Vstar(2),...
        (1-Psa)*Vstar(1)+Psa*Vstar(2)]);
    policy(1)=(tmp2-1)*2-1;
    for x=2:N-1
        [tmp1,tmp2]=max([Psa*Vstar(x-1)+(1-Psa)*...
            Vstar(x+1),(1-Psa)*Vstar(x-1)+Psa*Vstar(x+1)]);
        policy(x)=(tmp2-1)*2-1;
    end
    [tmp1,tmp2]=max([Psa*Vstar(N-1)+(1-Psa)*Vstar(N),...
        (1-Psa)*Vstar(N-1)+Psa*Vstar(N)]);
    policy(N)=(tmp2-1)*2-1;
```

The whole procedure should be run until the policy does not change any more. This stable policy is then the policy we should execute in the agent.

10.4.3 Monte Carlo methods and Q-learning

We have just seen that it is possible to solve the Bellmann equations exactly, but this requires to go repeatedly through every possible state, and in case of value iteration, also through every possible action. This is not only demanding, but seems also not the way a biological agent would go about learning the environment. It seems more biological plausible that an agent would take a specific action and end up in a new state, and the agent could then only use the corresponding feedback to evaluate the the value function. It might then use the new value function to update its policy. This approach looks a little bit like policy iteration without evaluating the Value function for every state. Monte Carlo methods follow this ideas by taking only specific actions that are a combination of biased choices to maximize the expected payoff, and some randomness to allow for an exploration of the state space. A big advantage of such methods is also that they do not require a full knowledge of the environment as the previous methods and can hence be used in an online mode.

When using specific actions rather than iterating over all possible states and actions, one major problem is the **exploration-exploitation tradeoff** mentioned at the beginning of this chapter. To avoid getting trapped in self-reinforcing suboptimal policies, we need include some kind of escape mechanisms. A simple example is using an **ϵ-greedy decision algorithm** instead of the to greedy algorithm used above. Above we chose the action that maximizes the expected reward to calculate the optimal policy. In the ϵ-greedy decision algorithm we choose this maximal policy most of the time but chose another action with probability ϵ. Such an stochastic escape algorithm is important to find good solutions.

There are several variant of such algorithms. For example, it is natural to go to the state that is recommended by the policy and use this sequence of actions to evaluate the value function. Such a strategy is labeled as **on-policy**. In contrast, it is also possible to take different actions to explore environment while still suing the recommended actions to calculate the value function. Such methods are labeled as **off-policy**. Many other strategies can be devised to improve the appropriate exploration of specific environments.

Finally, we have mainly discussed situations where each state has a unique value in terms of reward, and a slightly more general case is the situation where each state can have different values when combined with different actions. Thus, instead of considering a state value function $V^\pi(s)$, we can consider a **state-action** value function $Q(s,a)$. This slightly more general case is often called **Q-learning**. The principles and formulas are very similar to the previous case, but will not be discussed here further.

10.5 Temporal Difference learning

One of the most influential and practical methods in reinforcement learning has been invented by Rick Satton and Andrew Barto. This methods is essentially a Monte Carlo method of policy iteration in the sense that it is an on-line method of exploring the environment. The difference lies in the estimation of the value function. To derive this algorithms, we start again with Bellman's equation for a specific policy (eq.10.8),

$$V^\pi(s) = R(s) + \gamma \sum_{s'} P_{\pi(s)a}(s')V^\pi(s'). \tag{10.18}$$

The sum on the right-hand side is over all the states that can be reached from state s. Another difficulty is that we typically don't know the transition probability and have to estimate this somehow. The strategy we are taking now is that we approximate the sum with only the step actually taken by the agent, the state which put the agent into state $s+1$,

$$\sum_{s'} P_{\pi(s)a}(s')V^\pi(s') \approx V^\pi(s+1) \tag{10.19}$$

While this term makes certainly an error, the idea is that this will still result in an improvement of the estimation of the value function, and that other trials have the possibility to evaluate other states that have not been reached in this trial. The value function should then be updated carefully, by considering the new estimate only incrementally,

$$\begin{aligned} V^\pi(s) &\leftarrow (1-\alpha)V^\pi(s) + \alpha(R(s) + \gamma V^\pi(s+1)) && (10.20)\\ &= V^\pi(s) + \alpha(R(s) + \gamma V^\pi(s+1) - V^\pi(s)). && (10.21)\end{aligned}$$

The constant α is called a learning rate and should be fairly small. We can then use the latest estimate of the value function to update our policy such as choosing the policy that maximizes the newly calculated expected future reward,

$$\pi(s) = arg \max_a \sum_{s'} V(s'). \tag{10.22}$$

This correspond to a **greedy policy** which works only in the case where there are distributed transition probabilities that enable some exploration of the environment. If the system to too deterministic it is advisable to use some other stochastic policy instead. For example, the TD algorithm can be summarized in the ϵ-greedy case as shown in Figure 10.4.

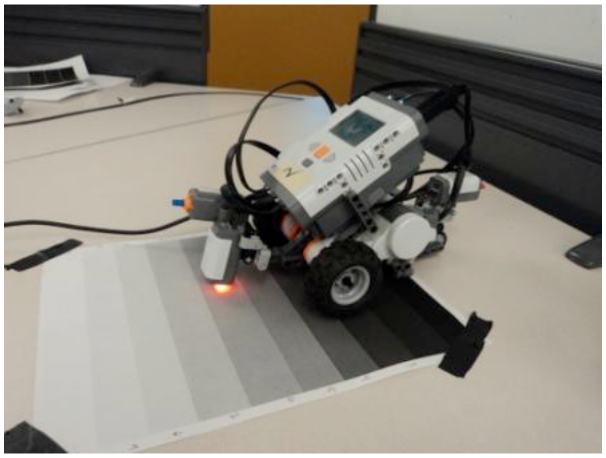
```
Choose initial policy and value function
Repeat until policy is stable {
          1. Policy evaluation
          Repeat until change in values is sufficiently small {
          Remembering the value function and reward of current state (eligibility trace)
          If rand>$\epsilon$
                    Go to next state according to policy of equation 10.22
          else
                    go to different state
          Update value function of previous state according to (equation 10.20)
          $V^\pi(s-1) \leftarrow V^\pi(s-1) + \alpha(R(s-1) + \gamma V^\pi(s) - V^\pi(s-1))$
                    } convergence
          2. Policy improvement
          new policy according to equation 10.11, assuming $V^* \approx$ current $V^\pi$
} policy
```

Fig. 10.4 On-policy Temporal Difference (TD) learning

10.6 Robot exercise with reinforcement learning

10.6.1 Chain example

The first example follows closely the chain example discussed in the text. We consider thereby an environment with 8 states. An important requirement for the algorithms is that the robot must know in which state it is in. As discussed in Chapter **??**, this localization problem is a mayor challenge in robotics. We use here the example where we use a state indicator sheet as used in section 5.3. You should thereby use the implemented of the calibration from the earlier exercise.



Our aim is for the robot to learn to always travel to state 8 on the state sheet from any initial position. It is easy to write a script with explicit instruction for the robot, but the main point here is that the robot must learn the appropriate action sequence from only given reward feedback. Here you should implement three RL algorithms. The first two are the basic dynamic programming algorithms of value iteration and policy iteration. Note that you can assume at this point that the robot has full knowledge of the environment so that the robot can find the solution by ‘contemplating’ about the problem. However, the robot must be able to execute the final policy.

The third algorithm that you should implement for this specific example is the temporal difference (TD) learning algorithm. This should be a full online implementation in which the robot actively explores the space.

10.6.2 Wall Avoider Robot Using Reinforcement Learning

The goal of this experiment is to teach the NXT robot to avoid walls. Use the Tribot similar with an ultrasonic sensor and a touch sensor mounted at the front. The ultrasonic sensor should be mounted to the third motor so that the robot can look around. An example is shown in Fig. 10.5. Write a program so that the robot learns to avoid bumping

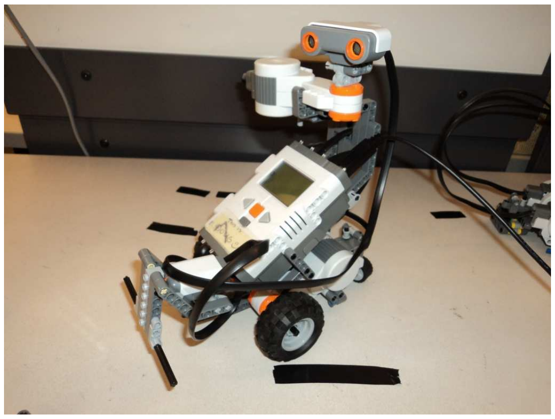


Fig. 10.5 Tribot configuration for the wall avoidance experiment.

by giving negative feedback when it hits an obstacle.

Appendix A Basic NXT toolbox commands

The following instructions have been adapted from RWTH's website. Installation instructions from:
http://www.mindstorms.rwth-aachen.de/trac/wiki/Download4.03
Coding instructions from:
http://www.mindstorms.rwth-aachen.de/trac/wiki/Documentation

You can find more instruction on installation and usage of the RWTH Mindstorms NXT Toolbox from both of these sites.

Startup NXT

The first thing to do is make sure the workspace is clear. Enter:

```
COM_CloseNXT('all');
close all;
clear all;
```

To start, enter:

```
hNXT=COM_OpenNXT;           %hNXT is an arbitrary name
COM_SetDefaultNXT(hNXT);    %sets opened NXT as the
                            %default handle
```

NXT Motors

Motors are treated as objects. To create one, enter:

```
motorA = NXTMotor('a');   %motorA is an arbitrary name, 'a' is
                          %the port the motor connected to
```

This will give:

```
NXTMotor object properties:
                 Port(s): 0   (A)
                   Power: 0
        SpeedRegulation: 1   (on)
             SmoothStart: 0   (off)
              TachoLimit: 0   (no limit)
  ActionAtTachoLimit: 'Brake'   (brake, turn off when stopped)
```

A.0.1 Basic Motor Commands & Properties

Below is a list of these properties and how to change them:

Power
Determines speed of the motor

```
motorA.Power=50;  % value must be between -100 and 100 (negative
                  % will cause the motor to rotate in reverse)
```

SpeedRegulation
If the motor encounters some sort of load, the motor will (if possible) increase it power to keep a constant speed

```
motorA.SpeedRegulation=true;  % either true or false, or
                              alternatively, 1 for true, 0 for
                              false
```

SmoothStart
Causes the motor to slowly accelerate and build up to full speed.
Works only if ActionAtTachoLimit is not set to 'coast' and if TachoLimit>0

```
motorA.ActionAtTachoLimit= true;  % either true or false, or
                                  % 1 for true, 0 for false
```

ActionAtTachoLimit
Determines how the motor will come to rest after the TachoLimit has been reached.
There are three options:
1. 'brake': the motor brakes
2. 'Holdbrake': the motor brakes, and then holds the brakes
3. 'coast' the motor stops moving, but there is no braking

```
motorA.ActionAtTachoLimit='coast';
```

TachoLimit
Determines how far the motor will turn

```
motorA.TachoLimit= 360;  % input is in terms of degrees
```

Alternative Motor Initiation
Motors can also be created this way:

```
motorA=NXTMotor('a', 'Power', 50, 'TachoLimit', 360);
```

Other Motor Commands

SendToNXT
This is required to send the settings of the motor to the robot so the motors will actually run.

```
motorA.SendToNXT();
```

Stop
Stops the motor. There are two ways to do this:
1. 'off' will turn off the motor, letting it come to rest by coasting.
2. 'brake' will turn cause the motor to be stopped by braking, however the motors will need to be turned off after the braking.

```
motorA.Stop('off');
```

ReadFromNXT();
Returns a list of information pertaining to a motor

```
motorA.ReadFromNXT();
```

Entering `motorA.ReadFromNXT.Position();` will return the position of the motor in degrees.

ResetPosition
Resets the position of the motor back to 0

```
motorA.ResetPosition();
```

WaitFor
Program will wait for motor to finish current command. For example:

```
motorA=('a', 'Power', 30, 'TachoLimit', 360);
motorA.SendToNXT();
motorA.SendToNXT();
```

This command will cause problems as the motor can only process one command at a time. Instead, the following should be entered:

```
motorA=('a', 'Power', 30, 'TachoLimit', 360)
motorA.SendToNXT();
motorA.WaitFor();
motorA.SendToNXT();
```

The exception to this is if TachoLimit of the motor is set to 0.

Using Two Motors At Once

Some operations, for example driving forward and backwards, require the simultaneous use of two motors. Entering:

```
B=NXTMotor('b', 'Power', 50, 'TachoLimit', 360);
C=NXTMotor('c', 'Power', 50, 'TachoLimit', 360);
B.SendToNXT();
C.SendToNXT();
```

will start the bot moving, but the signals for both motors to start at will not be sent at exactly the same time, so the robot will curve a little and fail to drive in a straight line. Instead, you should enter:

```
BC=NXTMotor('bc', 'Power', 50, 'TachoLimit', 360);
```

OR

```
BC = NXTMotor('bc);
BC.Power=50;
BC.TachoLimit=360;
```

Turning left or right can be achieved by only running one motor at a time, or by moving both motors, but one slower than the other.

Sensors

The following commands are used to open a sensor, plugged into port 1:

```
OpenSwitch(SENSOR_1);              % initiates touch sensor
OpenSound(SENSOR_1, 'DB');         % initiates sound sensor, using
                                   % either 'DB' or 'DBA'
OpenLight(SENSOR_1, 'ACTIVE');     % initiates light sensor as
                                   % either'ACTIVE' or 'INACTIVE',
                                   % plugged into Port 1
OpenUltrasonic(SENSOR_1);          % initiates ultrasonic sensor
                                   % plugged into Port 1
```

The following commands are used to get values from the sensor plugged into port 2:

```
GetSwitch(SENSOR_2);        % returns 1 if pressed, 0 if depressed
GetSound(SENSOR_2);         % returns a value ranging from 0-1023
GetLight(SENSOR_2);         % returns a value ranging from 0 to
                            % a few thousand
GetUltrasonic(SENSOR_2);    % returns a value in cm
```

To close a sensor, ex. Sensor 1:

```
CloseSensor(SENSOR_1);   %properly closes the sensor
```

Direct NXT Commands

PlayTone
Plays a tone at a specified frequency for a specified amount of time

```
NXT_PlayTone(400,300);   % Plays a tone at 400Hz for 300 ms
```

KeepAlive
Send this command every once in a while to prevent the robot from going into sleep mode:

```
NXT_SendKeepAlive('dontreply');
```

Send this command to see how long the robot will stay awake, in milliseconds:

```
[status SleepTimeLimit] = NXT_SendKeepAlive('reply');
```

GetBatteryLevel
Returns the voltage left in the battery in millivolts

```
NXT_GetBatteryLevel;
```

StartProgram/StopProgram
To run programs written on LEGO Mindstorms NXT software, enter:

```
NXT_StartProgram('MyDemo.rxe')  % the file extension '.rxe' can be
                                % omitted, it will then be automatically
                                % added
```

Entering `NXT_StopProgram` stops the program mid-run