School of Cognitive and Computing Sciences

UNIVERSITY OF SUSSEX AT BRIGHTON

Formal Computational Skills

Course Notes

Autumn Term 2002

Andrew Philippides
(notes by David Young)

Contents

Section 1 Some differential calculus

This section provides some background information for sessions 2 and 3 of the course Formal Computational Skills. These deal with the application of ideas from differential calculus to the analysis of neural networks in which the signal can be represented by continuously varying quantities.

Contents

1 Introduction

This cannot replace textbooks — or it would be one. Rather, it's an outline intended to enable you to find out something about a set of techniques that is useful in analysing some kinds of neural nets (as well as many other systems).

Some of you will find all this familiar already. You can skim it in a few minutes and move on to something else.

Some of you will remember doing this once, but it's now rusty. You should look through this file and see whether you can still understand what is going on, especially in the examples. You might have to check the odd thing out in a textbook. You should ask about anything that isn't clear. You might have to spend an hour or two brushing the dust off the material.

Some of you will find this either new, or thoroughly lost in the mists of time. You may well need to go over old notes, or look up textbooks, and you should work through a few examples to make sure you do have the ideas straight. You should ask for help if you can't fathom something.

You are not expected to become fast and expert in all this material — that takes more time than we have. What you should aim for is to understand these techniques, so that you can follow an argument that involves them in a paper or a book.

Textbooks

Schaum's outline series is good on specific topics, but beware information overload.

If parts 2, 3 and 4 of this file are difficult, then you need to refer to an introductory textbook of about A-level standard, though books explicitly for A-level are often too closely tied to the examination syllabus. "Foundation Mathematics" 2nd edition, by D.J. Booth (Addison Wesley, 1994) looks useful, though at present is not available in the University Library. The best book for you is largely a matter of personal taste — you should try to find something that suits you.

For parts 5 onwards, you need to look at a more advanced book. Books for mathematicians spend too much time establishing a rigorous basis for everything — you need a book of mathematics for engineers or physicists. I use "Mathematical Methods in the Physical Sciences" by M. L. Boas (Wiley, 1st edition 1966, 2nd edition 1983) — the library has multiple copies (at QE 7000 Boa), as has the bookshop. "Mathematical Techniques: an Introduction for the Engineering, Physical and Mathematical Sciences", by D.W. Jordan & P. Smith, covers some similar ground but seems to have more introductory material than Boas. The library at present has a single copy at QE 7000 Jor. You may already have a personal preference — if so, stick with it.

A dictionary of mathematics can be surprisingly handy. It won't explain things in the way a textbook will, but it is often very useful to remind oneself of some particular bit of usage. They usually have some useful tables (e.g. of derivatives). The Penguin Dictionary of Mathematics is good, as is the Oxford Dictionary.

2 Functions of a single variable

You should be familiar with the idea of a *function* of a variable. Roughly speaking, a function (sometimes called a mapping) can be thought of as taking as “input” one value and producing as “output” another value. The general notation is $y = f(x)$, where x is the name of the “input” variable, or *argument*, f is the name of the function, and y is the name of the “output” variable. Often x is called the *independent* variable and y is called the *dependent* variable.

Many of the functions we will need take a *real* number as an argument. (A real number is one that can be written as a decimal value, like 3.2712 – but possibly with an unlimited number of digits.) Examples include:

$$y = \sin(x)$$
$$y = \cos(x)$$
$$y = \tan(x)$$
$$y = \log(x)$$
$$y = e^x$$
$$y = 3 \times x + 2$$
$$y = 3 \times x^2 + 2 \times x - 333$$

Note that the last three do not use the *f(x)* notation, but still represent functions. (You should be familiar with the convention that multiplication and division are done before addition and subtraction, and exponentiation is done first of all, by the way. Also, e^x can be written $\exp(x)$ and $3 \times x$ is more often written simply $3x$.)

It’s also possible to have functions like

if x is greater then 13 then $y = 1$, otherwise $y = 0$

Mathematicians use many tools to understand the properties of functions, for example the series expansion. We will not generally need this level of analysis.

The first thing we usually need to know is how to *evaluate* a function — that is, how to find a value of *f(x)* for some specific x. You will nearly always do this with the aid of a computer program in some form — so knowing what functions you can evaluate depends on knowing something about the libraries available with your current programming language. It is possible to evaluate the functions listed above in almost every language. In fact, almost all computed evaluations of functions are *approximate*, and sometimes it is important to know how this affects the result of a program.

It is also often important to be able to *visualise* the function, by drawing its graph. Again, it is now normal to use a computer-based method for this — check out packages like Matlab. When you draw a graph, think of each point on the paper (or screen) as representing a pair of values, x and y. The curve that is plotted represents the subset of values defined by the function. The notation (x, y) can be used to represent a pair of values, as well as the point in the plane that represents that pair.

Finally, you may need to use some *properties* of the function. For example, the trigonometric functions mentioned above are *periodic* — adding $2 \times \pi$ (about 6.283) to the value of x, for any x, produces the same result y (check what this means visually by drawing the graph). This property would be written down as for sin, say, as $\sin(x) = \sin(2 \times \pi + x)$. Another example is that the log function always increases if its argument increases — you could write this as $\log(u) > \log(v)$ if $u > v$. Properties like this are sometimes apparent from the graph, and are worth picking up as you go along when you encounter a particular function.

These ideas should be familiar to most people. If you are rusty, a good way to become familiar with them again is to plot some graphs using a package, or indeed by hand if you prefer. You should have a nodding acquaintance with all the functions listed above.

3 Differentiation

The basic idea of the differential calculus is that of a *rate of change*. Consider a function whose graph is a straight line, such as $y = 3 \times x + 2$. Any change in x produces a change 3 times as big in y. (On the graph, this can be seen by drawing a right-angled triangle below the line, with two of its sides parallel to the axes.) The *slope* of the line is said to be 3 in this case, for every x (because it’s a straight line, the slope is the same everywhere).

When we have a curve instead of a straight line, the amount of change in y produced by a change in x may depend both on how big the change is, and what value of x we started from. However, for many functions (and for most that are practically useful), the idea of the change in y produced by a *small* change in x turns out to be a consistent and valuable one. The change in y divided by the change in x, as we consider smaller and smaller changes, settles down to a steady value called the *derivative* of y with respect to x. This is usually written dy/dx. It can still be visualised as the slope of the curve; now though, it’s a property of a small section of the curve, and so depends on the value of x.

It is often important to know how a change in one quantity affects another, and so to be able to work out derivatives. To do this, there are various rules that you should be aware of. Some of the more important ones are:

Rules for specific functions:

For example,

$$\text{if } y = \sin(x) \text{ , then } \frac{dy}{dx} = \cos(x)$$

It is possible to work these out from first principles, but usually one would look them up in a table in a textbook, or use a symbolic computing package, to remind oneself of them. You should know where to find the rules for the functions mentioned above.

Rules for classes of functions:

Sometimes a rule is more general. One of the most useful is:

$$\text{if } y = x^n \text{, then } \frac{dy}{dx} = n \times x^{n-1}$$

This applies to a *class* of functions; the *parameter n* says which member of the class is being used; you substitute the value for your application. For example, if $y = x^4$, then d*y*/d*x* = $4 \times x^3$.

A simple rule of this type is:

$$\text{if } y = n \times x \text{, then } \frac{dy}{dx} = n$$

which should be obvious by thinking about the graph of the function. Here *n* is to be thought of as standing for a constant, rather than as being itself a variable.

The rule for products:

If a function can be written down as two functions multiplied together, and you can differentiate each of the two functions separately, then you can differentiate the function itself using the rule

$$\text{if } y = f(x) \times g(x) \text{ ,}$$

$$\text{then } \frac{dy}{dx} = f(x) \times \frac{dg(x)}{dx} + g(x) \times \frac{df(x)}{dx}$$

(Note that d*f*(*x*)/d*x* means d*y*/d*x* for *y* = *f*(*x*).)

For example, if $y = 3 \times x \times \cos(x)$, then d*y*/d*x* = $-3 \times x \times \sin(x) + 3 \times \cos(x)$.

The chain rule:

If a function can be written as one function applied to the result of another function, then the derivative of the whole thing can be got using

$$\text{if } y = f(g(x)) \text{ ,}$$

$$\text{then } \frac{dy}{dx} = \frac{df(z)}{dz} \times \frac{dg(x)}{dx}$$

evaluated for $z = g(x)$.

For example, if $y = \sin(x^2)$, then d*y*/d*x* = $2 \times x \times \cos(x^2)$. You get to this result by writing $z = x^2$.

Applying these last two rules, though harder, basically involves substituting one thing for another consistently. If you can't make sense of the rules, then the problem might well lie in the notation for functions, and in remembering what each symbol stands for. Although there is no need to be very fluent in this area, you should be able to understand what is going on (you should be able to see why the examples have the answers they do) and to differentiate most functions that you meet, even if you have to look up the rules.

4 Functions of several variables

For many applications, the idea of a function outlined above needs to be generalised to functions of more than one real variable. A function of two variables might be written $z = f(x, y)$. You can think of *x* and *y* as inputs and *z* as the output. A very simple example is *z* = *x* + *y*.

Usually, such functions are built out of the 1-dimensional functions described above. When there are two inputs and one output, if is often useful to visualise the function as a *surface* or landscape: the arguments *x* and *y* represent position on a 2-D plane, and the value *z* represents height above that plane (or below it if negative). Packages such as Matlab are very good at displaying these surfaces.

For functions of more than two variables, there is no simple way to visualise the whole function. Nonetheless, such functions are often discussed in a way that is analogous with the two-variable case.

If a function has many arguments, it may not make sense to give them all separate names. You might see something like

$$y = f(x_1, x_2, \ldots, x_N)$$

meaning that f is a function of *N* variables, which are distinguished by subscripts rather than by having completely different names. This kind of thing is very common in neural network analysis.

5 Partial differentiation

It is often necessary to know something about how the value of a function with several inputs is changed by small changes to its arguments — that is, we need to differentiate it. How can this be done?

The basic idea is quite simple. Consider the function $z = x \times y$. Suppose that instead of being a variable,

y simply stood for a fixed value — let's say 5. Then the function would be $z = x \times 5$, and so it would follow that in this particular case dz/dx = 5 (it's the straight line equation again). If we didn't know the particular value of y, but we did know that it was fixed, we could still write dz/dx = y, with the understanding that y was being treated as a fixed quantity rather than a variable. This derivative, found by pretending that y is a fixed quantity, is called the *partial derivative* of the function with respect to x.

In order to distinguish this from an ordinary derivative, some special notation is used: a curly d instead of a normal d, looking like this: ∂. Thus the expression

$$\frac{\partial z}{\partial x} = y$$

means "the partial derivative of z with respect to x" — that is, the change in z when x is varied and *all other arguments are kept constant*.

It is generally quite easy to find partial derivatives, once you have understood the principle of pretending that everything except the variable in question behaves just like a numerical constant. For example:

if

$$z = 3 \times y^2 + y \times \sin(x + 10 \times v)$$

then

$$\frac{\partial z}{\partial x} = y \times \cos(x + 10 \times v)$$

$$\frac{\partial z}{\partial y} = 6 \times y + \sin(x + 10 \times v)$$

$$\frac{\partial z}{\partial v} = 10 \times y \times \cos(x + 10 \times v)$$

If you can't verify the results in this example, it's probably because you need to check the rules for basic differentiation, rather than because partial differentiation is itself a problem.

Note that the partial derivative may be a function of all or some of the arguments to the original function.

The partial derivative tells us how a function is affected by a perturbation to one of its arguments. This in itself can be very useful. Sometimes it is necessary, though, to know how a function changes when a change is made to many or all of its arguments. This will only make sense if the changes to the arguments are coordinated in some way; that is, the arguments themselves are functions of some other variable that is changing. (This is often the case in neural networks.)

To be definite, suppose z depends on (is a function of) u and v, so $z = f(u, v)$, and u and v both depend on some other variable x, so $u = g(x)$ and $v = h(x)$. (Here, g and h are names of functions.) The question is, how does z vary if x changes?

The answer is given by the *chain rule for partial differentiation*, which is the most advanced idea to be mentioned in this file. It says that

$$\frac{dz}{dx} = \frac{\partial z}{\partial u} \times \frac{du}{dx} + \frac{\partial z}{\partial v} \times \frac{dv}{dx}$$

Note that all the quantities on the right can be worked out from the expressions for f, h and g. Putting them together gives the result that is needed. Since nothing is kept constant when x changes, the result on the left of the equation is an ordinary derivative.

This should make some kind of intuitive sense, along these lines: x controls each of u and v, and u and v together control z. So a change in x produces a change in z by two different routes. The effect along the u route is the effect of x on u times the effect on u on z. Similarly for the v route. The two effects get added together.

Sometimes, there are other variables which affect u and v, in addition to x. In this case these other variables have to be held constant while we investigate the effect of x on z. Then the ordinary derivatives in the formula become partial derivatives too, to indicate that these other things are staying constant.

Textbooks will give a proof of this formula, and sometimes a graphical way to think about it as well.

6 Some kinds of Ds

As light relief, it may be worth mentioning that differential calculus abounds in variants of the letter D. So far, we have only used 2 kinds, but you will encounter others in the literature. To try to avoid confusing them, here is a little table — although you do not need to be familiar with the use any but the first two at this stage, it is worth knowing that the others exist.

Name	Symbol	Used for
small d	d	Derivative[1]
curly d	∂	Partial derivative
small delta	δ	A small change in a variable[2]
capital delta	Δ	An arbitrary change in a variable
del or nabla[3]	∇	A kind of vector derivative
capital D	D	Differential operator[4]

Notes:

1. Small d is almost always used in the form d*x*/d*y*. The quantity d*x* is called an *infinitesimal*, and means a change in *x* which is smaller than any finite change. Debate has raged about whether it is proper to manipulate infinitesimals in their own right rather than as top or bottom of a derivative. There are hints that they are currently becoming more respectable, but they won't be used here.
2. Though small, δ*x* is finite — i.e. not an infinitesimal.
3. I'm not sure whether this is kind of "D", but it's easily confused with Δ so I've put it in.
4. D *f(x)* is used to mean d*y*/d*x* when *y* = *f(x)*. It is too concise for elementary use but comes into its own in the study of differential equations.

7 Summation

Finally, you should be able to read the notation for forming sums — that is, adding a set of things together. This uses the capital Greek letter sigma, which looks like this: Σ.

Here is a simple example of how it is used:

$$\sum_{k=1}^{5} (k \times x)$$

and this expands into (is equal to)

$$x + 2 \times x + 3 \times x + 4 \times x + 5 \times x$$

In general, there is some variable (in this case *k*) which takes a set of values (in this case 1,2,3,4 and 5). For each of these values, an expression involving the variable (in this case $k \times x$) is evaluated, and the results added together. In the form in which it is being used here, the variable takes integer values, starting from the one specified below the Σ, and going up to the value specified above. (There are a few alternative forms of the notation, but this is the most common.) In this case, there is another variable, x, in the expression, but there might be several other variables, or none.

It is extremely common for the summation variable to form a subscript in the expression, rather than being an arithmetic element as above. For example

$$\sum_{j=3}^{6} x_j^2 = x_3^2 + x_4^2 + x_5^2 + x_6^2$$

The name given to the summation variable (here *j*) can be chosen arbitrarily but must then be used consistently, as for all variable names.

There is a nice concrete way to think about the summation notation, if you are a programmer. A summation sign acts like a loop in a program, and indeed programs that implement theories involving sums do have corresponding loops. If you happen to know C, for example, then it may help to know that the following line of code implements (with suitable declarations of course) the first example above. It leaves the variable `sum` set to the value of the whole Σ expression, assuming that *x* has been given a value beforehand:

```
for (sum = 0.0, k = 1; k <= 5; k++)
       sum += k * x;
```

whilst the second example would translate into something like

```
for (sum = 0.0, j = 3; j <= 6; j++)
       sum += x[j] * x[j];
```

Summation gets complicated when you encounter nested summation signs — one Σ being applied to an expression containing another Σ. There is a fairly safe way to make sure you understand what is going on in cases like this: write out a few terms of the whole expression. You should be able to understand the following:

$$\begin{aligned}\sum_{i=1}^{2}\sum_{j=1}^{2} x_{ij} &= \sum_{i=1}^{2} (x_{i1} + x_{i2}) \\ &= (x_{11} + x_{12} + x_{21} + x_{22})\end{aligned}$$

For those who feel this holds no mysteries, it is worth mentioning that there is a convenient shorthand which is sometimes used for sums, called the *repeated suffix convention*, or *tensor notation*. In this convention, any suffix which appears twice in an expression is taken to be summed over — an implicit Σ appears before the expression with the repeated variable as the summation variable. This is only useful when the range of summation is obvious. The convention is often a very useful alternative to matrix notation.

Finally, something fairly hard. Let's use the summation notation to generalise the chain rule for partial differentiation. Suppose our output variable *z* is affected by a load of different intermediate variables — say *N* of them, which we will call $u_1, u_2, \ldots, u_N$. Suppose that *x* affects each *u* (possibly in a different way for each). Now if we want to know the effect of *x* on *z*, it's going to look like

$$\frac{dz}{dx} = \sum_{i=1}^{N} \frac{\partial z}{\partial u_i} \times \frac{du_i}{dx}$$

(with partial instead of ordinary derivatives if there are some other variables being held constant).

If this looks daunting the first thing to do is to write it out in full with N equal to 2. Then the relationship to the earlier formula for the chain rule should become clear.

Section 2 The backpropagation algorithm

This section complements Section 1 by relating the mathematical ideas described there to an important application: the analysis of a learning algorithm for feedforward neural networks.

Contents

1 Introduction

This section is a complement to Section 1. It applies the mathematics in it to a single problem: the training of a multilayer feedforward nonlinear neural network (sometimes called a multilayer perceptron). Although this application is important in its own right, the purpose here is to give you some feeling for how ideas from calculus get applied to a real computational problem. In addition, you should get some idea of how the formal mathematical notation relates to the more concrete computational structures represented by neural networks.

You should not expect to be able to reproduce this whole argument. You should, however, be able to see in general terms what is going on. The idea of gradient descent is central to many computational systems, and the use of subscripts and summation to manipulate arrays of quantities in a concise way is well worth getting used to, since it translates readily into practical computer programs.

Two relevant books are:

Rumlhard, D.E. & McClelland, J.L. “Parallel Distribution Processing: Explorations in the Microstructure of Cognition”, Vol. 1, MIT Press, 1986.

Hertz, J.A., Krogh, A. & Palmer, R.G. “Introduction to the Theory of Neural Computation”, Addison Wesley, 1991.

Both of these carry out essentially the same development as this file, though more concisely.

2 The response function for a simple model neuron

Artificial neural networks (NNs) are formed by connecting individual units which very approximately correspond to neurons. Each unit receives some inputs from other units, or from outside the network, and produces an output which goes to other units or to outside the network. In an important class of NNs, the inputs and outputs are represented by real numbers, and each unit computes an output which is a function of its inputs. A common form for the function is described here; for justification see books on NNs, and remember that all sorts of other possibilities (such as networks with binary values only, or units with memory) exist.

In this section a single unit is considered. Suppose it has N inputs, which will be called x_1, x_2 etc. up to x_N; the output of the unit will be called y, so we can write $y = f(x_1, x_2, \ldots, x_N)$ to express the fact that the output is a function of the inputs. The form of the response function f determines the behaviour of the network.

2.1 Linear units

A very simple response function would be to just add up all the inputs. However, the unit may need to pay more attention to some inputs than to others, and this is achieved by first multiplying each input by a weight, which is a number expressing how strongly the input affects the unit. (I am regarding the weights as being associated with the unit they directly affect.)

For each input x_i there is a corresponding weight w_i. The unit therefore combines its inputs by computing a *weighted sum* according to the formula

$$a = \sum_{i=1}^{N} x_i \times w_i$$

where a is sometimes called the activation of the unit. The idea of a weighted sum is extremely common in many branches of science.

The unit could simply pass this activation to its output (i.e. it could set $y = a$). Such a unit is called a *linear unit* because if you plot y against any particular input x_i, keeping all the other inputs constant, you get a straight line.

If the weights are fixed, it is convenient to regard them as built-in to the response function f. However, the way that neural networks are trained is to adjust the weights to improve their performance. If we regard the weights as things that can be varied, it makes sense to regard the output as a function of the inputs *and* of the weights, or in symbols $y = f(x_1, x_2, \ldots, x_N, w_1, w_2, \ldots, w_N)$.

We will need to know later how a change in an input affects the output. For a linear unit (i.e. output = activation, or $y = a$), the relation is very simple:

$$\frac{\partial y}{\partial x_i} = w_i$$

That is, the change in the output is just the change in the input multiplied by the corresponding weight. If the mathematics is not obvious, try writing out the formula for y explicitly for the case of two inputs (i.e. $y = x_1 \times w_1 + x_2 \times w_2$). Then treating everying except y and x_1 as constants, use the ordinary rules for differentiation to find the derivative of y with respect to x_1. Generalise the result to any input.

For working out learning algorithms, it is also useful to know how a change in a weight affects the output if the inputs are fixed. The mathematics is identical, and produces the result

$$\frac{\partial y}{\partial w_i} = x_i$$

i.e. for a particular input, the effect of changing a weight depends on how strong the corresponding input was.

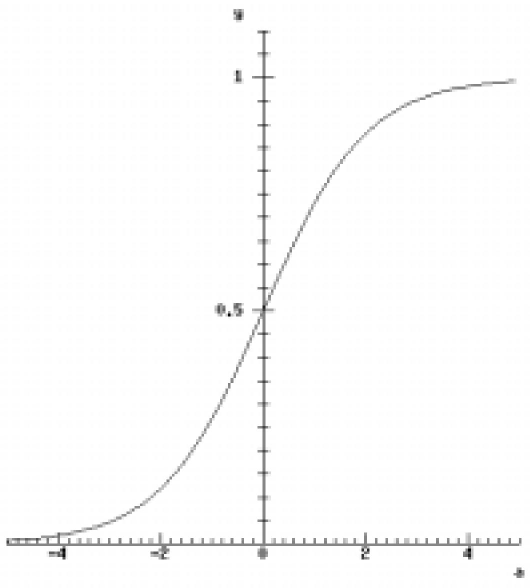
2.2 Nonlinear units

It is the case that neural networks built from linear units have a very limited range of responses (e.g. they cannot produce an output that simply says whether two inputs are different from or the same as one another). However, one modification turns out to give the networks enormously greater computational power, and that is to make the relationship between the activation a and the output y nonlinear. Typically, instead of $y = a$, interesting neural networks use a relationship such as

$$y = \frac{1}{1 + e^{-a}}$$

This is sometimes called the *logistic function*. Note that a is just the weighted sum of the inputs, as for the linear unit.

It is not appropriate here to investigate why this is a useful choice, and anyway it is by no means the only possibility. We will take it as given, but we will have a quick look at its properties. The first thing to do is to look at its graph.



If you look at the graph, you will see that it is vaguely S-shaped. Functions with this shape are called *sigmoidal* functions (nothing to do with the sigma used to indicate summation). Note that the logistic function is merely an example of a sigmoidal function.

You can see that y is 1 if a is large and positive, and y is 0 if a is very negative. Thus if you ignore the bit in the middle this is a little like a binary function. However, y's changeover from 0 to 1 occurs gradually as a crosses 0 — one could describe this as a kind of softened or smoothed step function. (A step function would cause y to jump from 0 to 1 as a crossed some value called the threshold.)

Now we need to answer the same questions about how changing the inputs and weights affects the output of such a unit. This is done in two steps: first we ask how a change in an x or a w affects a, then we ask how a change in a affects y.

The first step has already been done. From the analysis for a linear unit, where $a = y$, we know that

$$\frac{\partial a}{\partial x_i} = w_i$$

and

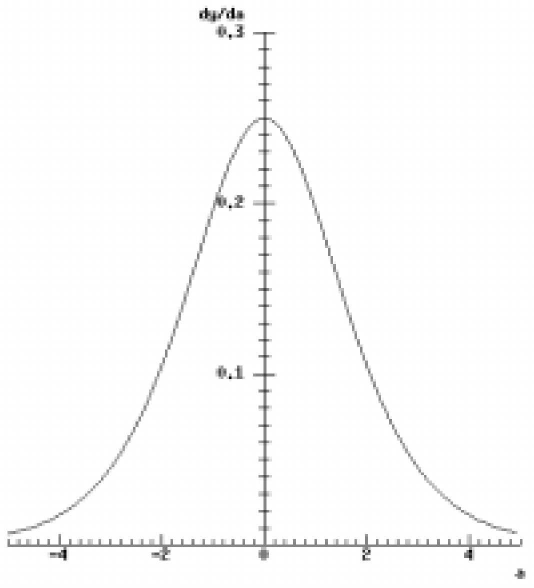
$$\frac{\partial a}{\partial w_i} = x_i$$

The second step is to find dy/da. This requires the application of the rules of ordinary differentiation to the logistic function. I will not do this in detail here; if you want to see the steps involved, please ask. You have to know the rules for differentiating e^x and $1/x$ (x is just a general purpose variable here), and you have to apply the chain rule. The answer comes out as

$$\frac{dy}{da} = \frac{e^{-a}}{(1 + e^{-a})^2}$$

(It is an ordinary derivative because a is the only thing that directly affects y — there's nothing that has to be held constant to do this calculation.)

You ought to be able to guess what the graph of this function looks like without using the formula, simply by looking at the graph of the logistic function and seeing what its slope does. In fact, it's like this:



There is a computationally useful simplification of the derivative formula. It is possible to use the original formula for the logistic function to show that

$$\frac{dy}{da} = y \times (1 - y)$$

If in some program, y has already been calculated, this gives a much faster way of finding the numerical value of the derivative than the formula that uses a.

Finally, the two steps are put together. You need to apply the chain rule, which says that

$$\frac{\partial y}{\partial x_i} = \frac{dy}{da} \times \frac{\partial a}{\partial x_i}$$

which gives

$$\frac{\partial y}{\partial x_i} = y \times (1 - y) \times w_i$$

Similarly, one gets

$$\frac{\partial y}{\partial w_i} = y \times (1 - y) \times x_i$$

This all looks fairly complex, but the final formulae are not too bad. They say how changes to the inputs or weights affect the output for this kind of nonlinear unit. They are in a form which allows them to be used in a program; such a program would have x, w and y available to it, so computing the partial derivatives would now be no problem.

These formulae imply that the unit's sensitivity to a change in a weight or an an input depends on all the other weights and inputs as well. If the unit's activation is close to zero (so its output is close to 0.5), it has its highest sensitivity. If the activation is strongly positive or negative, then the output is close to 0 or 1, $y \times (1 - y)$ becomes very small, and so changes in inputs and weights have relatively little effect on the output. The unit is then said to be *saturated*. The possibility that some inputs can put a unit into saturation, and so prevent other inputs affecting the output of that unit, is an important aspect of the operation of neural networks with nonlinear units.

3 Training a single unit

3.1 The error function for a single unit

Neural networks are often trained using *supervised learning*. In this kind of learning, the network's output for a given input is compared with a *target*, which is somehow known to be the "right answer". The weights are then adjusted to make the output closer to the target. If the weights are adjusted after each new example of an input/target pair, the mechanism is called *online* learning; if the weights are only adjusted after a set of input/target pairs then *batch* learning is taking place. Here, online learning will be considered, as it is slightly simpler to understand.

To decide how close the network's output is to the target, on any particular presentation, an *error function* is used. For our single unit, the target will be called t, and the error is given by

$$E = (y - t)^2$$

This function is chosen partly because *E* is always positive, and the bigger the difference between *y* and *t*, the bigger *E* is. It is clear that *E* is a function of *y* and *t*, and you will sometimes see this expressed as *E* = *E*(*y*, *t*). This is slightly loose usage, in that the same symbol, *E*, is being used for the name of the variable and the name of the function that computes it, but it is quite common and in practice does not cause confusion. You can read *E* = *E*(*y*, *t*) as "*E* depends on *y* and *t*".

Since *E* depends on *y* and *t*, and *y* depends in turn on the inputs and the weights, it is also true to say that $E = E(x_1, x_2, \ldots, x_N, w_1, w_2, \ldots, w_N, t)$.

It is sometimes convenient to think of this error as an *energy* associated with the network. If this was a mechanical system in which *y* represented the position of a robot arm, say, and *t* represented a target to which it was supposed to move, then *E* would be a measure of the energy of a spring connecting the arm to the target. The spring would pull the arm to the target, reducing its energy in the process. This kind of physical analogy can be useful in analysing various kinds of computational system.

If we single two weights or inputs for special consideration, and keep all the rest at some fixed values, then it is possible to draw a picture showing *E* as a surface above a plane. Positions on the plane correspond to values of the two variables under consideration, and the height of the surface above the plane corresponds to the value of *E*. This is a useful conceptual tool, and it is often helpful to think of the *error surface* or *energy landscape* even when more than two things can vary; it is no longer possible to picture the surface then, as it exists in many dimensions, but the ideas from the 2-variable picture are still useful.

Now the standard question: how do changes in inputs and weights affect the error? What we need are $\partial E / \partial x_i$ and $\partial E / \partial w_i$, with the partial derivatives indicating that *t* is kept constant. Since we know how *y* depends on the inputs and weights, all that is needed in addition is to know how *E* depends on *y*. This is got by differentiating the expression above, which gives

$$\frac{\partial E}{\partial y} = 2 \times (y - t)$$

(using the chain rule for simple differentiation). Then we can use (for inputs)

$$\begin{aligned}\frac{\partial E}{\partial x_i} &= \frac{\partial E}{\partial y} \times \frac{\partial y}{\partial x_i} \\ &= 2 \times (y - t) \times y \times (1 - y) \times w_i\end{aligned}$$

Both of the things multiplied together on the right are already known, so the whole expression is easily found. The effect of weight adjustments can be calculated the same way to get

$$\frac{\partial E}{\partial w_i} = 2 \times (y - t) \times y \times (1 - y) \times x_i$$

3.2 The learning rule for a single unit

How can the single unit be trained? A standard and simple procedure is to select an input and a target, present the input to the unit, compare its output with the target, and then change the weights slightly to make the output closer to the target. This is repeated for a large number of input/target pairs. Because each individual change made is small, the total effect over a large number of trials is to make the unit's behaviour closer to the overall optimum (though the details are beyond the present scope). At the very start, the weights are given some random values.

One way of looking at this is to say that at some point in the training, we want to change the weights so as to take a step downhill on the error surface. This method is called *gradient descent*. Basically, each particular weight needs to be changed in the right direction to reduce *E*; the bigger the effect of that weight, the more it should be changed.

It might be helpful to think of this geometrically. Consider the error surface for a unit with only two weights, and some fixed inputs; this surface can be pictured as some kind of smooth canopy above a plane. The current weight values fix the position of a point on the plane and the point vertically above it on the error surface. Changing one of the weights means moving in one direction on the plane; changing the other weight means moving in a direction at right-angles to this. Such moves will take the point on the surface uphill or downhill, depending on how the surface slopes. The change in height for a small change in a weight is given mathematically by the partial derivative, and geometrically by the slope of a line on the surface vertically above the line of motion in the weight plane. It should be possible to convince yourself that if you make a change in each weight proportional to the slope in the corresponding direction, the total movement is directly uphill or downhill.

For a definite example, consider the case where the error does not depend at all on one of the weights (the corresponding input is 0). Then the error surface only slopes in the direction of the other weight, and it is clearly only appropriate to change the latter.

This can be summed up in the *gradient descent learning rule*, which is a core rule in neural networks, and the starting point for many other more sophisticated rules. It says (for online learning) that on each presentation, the adjustment to each weight should be proportional to minus the partial derivative of the error with respect to that weight. In symbols:

$$\Delta w_i = -\alpha \times \frac{\partial E}{\partial w_i}$$

This is just saying that we take a small step downhill in the error surface.

Δw_i stands for the change in (adjustment to) w_i that we are going to make. The minus sign means that we go downhill, not uphill, when we make the adjustment. The constant α is used to keep the step small (in some sense) — in fact it used to be typically set to a value between 0.00001 and 0.2 by the experimenter and adjusted by trial and error, though there are now more principled and adaptive ways to determine a good value. The significance of the final part, the partial derivative or slope of the error surface, should be clear from the discussion above.

Now you are in a position to train a single unit. Putting everything together gives the adjustment to a weight after the presentation of an input as

$$\Delta w_i = -\alpha \times 2 \times (y - t) \times y \times (1 - y) \times x_i$$

The output y would be calculated once using the basic formula for operation of the unit, then the adjustment for each weight would be calculated and applied in turn using the formula above.

4 From a unit to a network

4.1 A single layer of units

One unit can only do so much. For really interesting behaviour, it's necessary to go to a network of interconnected units. Again there are many possibilities, and questions surrounding network topology are the subject of active research, but since the present purpose is to look at techniques for analysing networks, we will stick to one of the more amenable cases: the layered *feedforward* network.

In such a network, information flows from inputs to outputs, without any loops. The output of unit can never affect its inputs, which simplifies matters greatly. The units are arranged in layers, and each unit in any given layer gets its inputs from all the units in the previous layer (though of course it can ignore some of them by setting the corresponding weights to 0). (Sometimes the data from the rest of the world is thought of as coming through a layer of "input units", which simply pass their inputs on without changing them.)

First, take the case of a network that has a single layer (apart from any input layer). This will have one output for each unit in the layer. To analyse and train this network, the modifications to what we have done for a single unit are quite small, since essentially this is just like having a lot of units, all seeing the same inputs, but each doing its own thing.

The first change is in notation. We need to distinguish between the units, and to do this we have to use an extra subscript in some places. The output and activation of unit j will be called y_j and a_j respectively. There will have to be targets for all the units, so the target for unit j will be called t_j. The weight for input i going to unit j will be called w_{ji}. The inputs are the same for all the units so they do not need an extra subscript.

The convention for the order of the weight subscripts might seem perverse (the destination comes before the source). However, it is convenient in the end, because it corresponds to the convention used for matrix notation. You will find this ordering used in the textbooks mentioned above, but other authors sometimes use the opposite convention — look out for which one is in operation.

You should draw yourself a diagram with the inputs, outputs and weights for a single-layer network marked.

It might seem that we need to have an error function for each output unit. However, it is much more elegant (and in the long run more general and useful) to have a single error function for the whole network. The nice thing about the error function for a single unit defined above is that the error function for the whole network can usefully be defined as the sum of the errors for the separate output units. That is, we write

$$E = \sum_j (y_j - t_j)^2$$

There is a slight shorthand for the summation here: when the range of j is not specified, it is assumed that j runs over all the appropriate units. This shorthand will be used quite often.

This is called the *sum of squares* error function. Sometimes a factor of 0.5 is put in front; this makes no real difference to anything, but reduces the number of factors of 2 that occur later.

Having defined *E*, everything now proceeds as for the single unit, except we have to keep track of which unit we are talking about. First we work out how a change in unit *j*'s output affects *E*, by taking everything else as constant. By writing out the sum for the case of just 2 or three units, you should be able to persuade yourself that the result is just

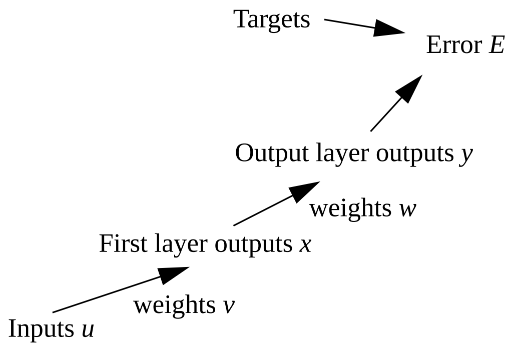
$$\frac{\partial E}{\partial y_j} = 2 \times (y_j - t_j)$$

That is, it is the same as for a single unit, except with a *j* subscript added as appropriate to indicate the unit. This is an extremely handy result — it simplifies the next stage, which is to find the derivative of *E* with respect to w_{ji}. But the analysis is now *exactly* the same as for a single unit, except for the extra subscripts. Thus a single layer is not really any more complex than a single unit, even though we have used a global error function which sums up (literally) the performance of the network as a whole.

4.2 Adding another layer

Adding a further layer makes, as it happens, a big difference to the generality and power of this kind of neural network. It also makes it harder to work out how to adjust the weights; working out an algorithm for doing so was one of the major breakthroughs of neural network research.

The notation adopted here for the extra layer is non-standard, but makes it easier to follow what is going on. The new layer of units will go on the input side of the layer we have already considered. (I will refer to the original layer as the output layer.) Now x_i will stand for the *output* of the *i*'th unit in the new layer and w_{ji} is the weight from the *i*'th unit in the new layer to the *j*'th unit in the output layer (so these symbols mean the same as before in connection with the output layer). The new inputs to the network as a whole will be called u_h and the weights from the inputs to the first layer will be v_{ih}. Thus we have:



This network is to be trained by gradient descent using the same rule as before. For a particular input, you can think of an error surface for *E* over a plane representing any of the weights in the system, the *v*s as well as the *w*s. We need to adjust all of them to go downhill on this error surface. The question is, what additional calculation do we need to do in order to carry this out?

Adjusting the weights *w* is exactly the same as before. The output layer doesn't care whether it's getting its inputs from the outside world or from a previous layer, so nothing changes for adjustments to w_{ji}. To adjust the weights *v*, however, we will need to know the slope of the error surface for these weights. This is the partial derivative written as $\partial E / \partial v_{ih}$. Finding an expression for this is the heart of the backpropagation algorithm.

5 Backpropagation

A change in one of the weights *v* will affect the output of the first layer of units, *x*, which will in turn affect the ouput of the second layer, *y* which will in turn affect *E*. We therefore look at this causal chain to see if it helps us work out the influence of *v* on *E*.

First, we calculate how a change in one of the *x*s affects *E*. A change in x_i will spread out to affect all the units in the output layer, which in turn all affect *E*. In terms of partial derivatives, this appears as

$$\frac{\partial E}{\partial x_i} = \sum_j \frac{\partial E}{\partial y_j} \times \frac{\partial y_j}{\partial x_i} \qquad \Diamond$$

This is probably the most difficult equation in this file, and involves the most advanced mathematics — the chain rule for partial differentiation. Its value is that if you can see how the structure of this equation relates to the information flow in the neural network, then you are in a good position to analyse similar systems. The explanation of the chain rule in Section 1 might help here, as might textbooks that deal with partial differentiation. Writing out the expression in full (i.e. making the summation explicit) for a network with only 2 or 3 output units might be helpful.

Note the different roles of the *i* and *j* subscripts: *j* is a variable which is summed over, like a loop variable in a program, whilst *i* says which particular first-layer neuron's output we are talking about, and is more like an argument to a procedure.

Having written this down, we can evaluate it if we want to, since we already know the formulae for the bits of the right hand side — they both appear higher up this file. The first part is

$$\frac{\partial E}{\partial y_j} = 2 \times (y_j - t_j)$$

(in section 4.1) and the second is

$$\frac{\partial y_j}{\partial x_i} = y_j \times (1 - y_j) \times w_{ji}$$

which is modified from the equation in section 2.2 by putting in the *j* subscript to specify which output unit we are talking about. But in fact these details are not very important right now.

Because E depends on x_i and x_i depends on v_{ih}, we have

$$\frac{\partial E}{\partial v_{ih}} = \frac{\partial E}{\partial x_i} \times \frac{\partial x_i}{\partial v_{ih}}$$

There's no summation because v_{ih} directly affects the unit whose output is x_i — no other variables get in the way.

We have just worked out how to get the first part of the right hand side. The second part is easy since it's just the same as working out $\partial y_j / \partial w_{ji}$ for an output unit, which we have already done. That is,

$$\frac{\partial x_i}{\partial v_{ih}} = x_i \times (1 - x_i) \times u_h$$

So we have everything we need to evaluate the partial derivative of the error with respect to one of the *v*s, and hence to decide how to adjust that *v*.

That completes the mathematics for the backpropagation algorithm — we can now train a 2-layer neural net, by calculating the error gradients with respect to all the weights, the *v*s and the *w*s, and then applying the gradient descent rule to adjust every weight.

Why is this computation called backpropagation? The reason is the recursive nature of the central equation just above, marked $\Diamond$. The derivative of the error with respect to the *x* outputs is found by using the derivative of the error with respect to the *y* outputs, together with a derivative which only depends on the *y* units. It is the quantity $\partial E / \partial o$, where *o* means the output of any unit (an *x* or a *y*), which is backpropagated across the layers. At each stage, the computation only involves one layer of units.

Suppose we put yet another layer of units into our network, again on the input side. Then we'd need the partial derivatives with respect to yet another layer of weights. To get these, we'd need $\partial E / \partial u_h$. And to get *that*, we'd use

$$\frac{\partial E}{\partial u_h} = \sum_i \frac{\partial E}{\partial x_i} \times \frac{\partial x_i}{\partial u_h}$$

This is just the equation marked $\Diamond$, written for the next layer down. We'd have got $\partial E / \partial x_i$ when doing the previous layer, and $\partial x_i / \partial u_h$ presents no problem now (we've done the same thing twice before). It's just the same computation again, and we could go on repeating it for as many layers as we like, working from the output to the input.

It was the realisation that this tractable computation was possible that allowed training of multilayer nonlinear networks to be carried out, and this in turn was one of the most important stimuli of their significant renaissance in the 1980's.

6 Conclusion

This rather long section has given an account of the mathematics behind training a multilayer nonlinear neural network. The essence of it is the equation marked $\Diamond$ — most of the rest is context for that: why it is needed and how it is used.

What this kind of mathematics is really about is keeping track of which variables depend on which others. If that is done, the partial derivative formulae should make some kind of sense, even if you could not carry out a detailed derivation of them. Relating the partial derivative formulae to the flow of information through the network is largely the point of the exercise.

Section 3 Matrices

This section extends Sections 1 and 2 by giving a basic introduction to the use of matrices.

Contents

1 Introduction

In the first two teach files, every variable that was used stood for a single real number. This included subscripted variables: something like x_i meant the *i*'th value in a collection, not the whole collection. This made it possible to interpret every equation according to the normal rules of arithmetic; addition, multiplication, differentiation and so on all had their ordinary elementary meanings, even if the context in which they were used involved, in principle, large numbers of variables.

Sometimes it is useful to be able to manipulate symbols that stand for structured collections of numbers. Matrices are a case of this, and their use is common in a wide variety of fields. Here, it will be illustrated by referring back to the work on neural networks in Section 2.

Discussion of this topic in textbooks crosses the A-level/degree level boundary. Books such as that by Boas (see Section 1) give a discussion, and also contain much background and related material — not all necessarily relevant at present. An outstanding reference book is "Matrix Analysis", by R.A. Horn and C.R. Johnson (Cambridge University Press 1985, later reprints), but this gives an advanced treatment that will only be useful if you already have a fairly mathematical background.

Matrix manipulation is, *par excellence*, an area in which high-quality software libraries and packages have liberated ordinary users from the need to be very familiar with a lot of algorithmic detail. Thus you will find books such as "Numerical Recipes in C", by W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling (Cambridge University Press 1988, later reprints) contain summary discussions and much practical information. The Matlab package, although wider in scope now than matrix manipulation, has particular strength in this area.

The remainder of this file summarises some of the central ideas. As usual, it does not purport to replace the examples and discussion offered by textbooks.

A point about notation

The multiplication symbol is generally omitted in textbooks — two symbols next to each other are multiplied together, so *xy* means $x \times$ y.. In Sections 1 and 2 I put in all the multiplication symbols for clarity and for consistency with the online teach files, but as the "×" symbol is almost never used with matrices, from now on I will omit it everywhere.

2 Matrix-vector multiplication as a neural network operation

Part 2.1 of Section 2 introduced a linear neural network unit with output *y*, inputs $x_1 \ldots x_N$ and weights $w_1 \ldots w_N$. Since its output was simply equal to its activation, it computed the function

$$y = \sum_{i=1}^{N} w_i x_i$$

Section 4 part 1 introduced the idea of a layer of units, in which the units were distinguished by adding an extra subscript to each *y* and *w* (but not to *x* because the inputs are the same for each unit). Thus, if the units in the layer are linear, this equation becomes

$$y_j = \sum_{i=1}^{N} w_{ji} x_i$$

where j just specifies which unit we are talking about.

This can be rewritten as a *matrix-vector multiplication*. What is done is to represent each set of quantities by a single symbol; each such collection is called a matrix or a vector. If the individual quantities have two subscripts, then their collection is a matrix; if the individual quantities have one subscript, then they form a vector. For example, the bold character $\boldsymbol{y}$ stands for y_1, y_2 up to y_N, all put together into a single vector. Likewise $\boldsymbol{w}$ stands for all the weights put together into a matrix and $\boldsymbol{x}$ for all the inputs put together into a vector.

It is important to realise that $\boldsymbol{w}$, which stands for a matrix, is a completely different kind of object to w_{ji}, which stands for a number. The number is said to be an *element* of the matrix.

A vector, in this sense, is really just a particular kind of matrix — it's a matrix for which the second subscript is always 1, and so there's no point in writing it.

There is a rule for multiplying matrices and vectors. Conveniently, it is just the rule used by a single layer linear network to compute its outputs. That is, we can write

$$\boldsymbol{y} = \boldsymbol{wx}$$

to mean *exactly* the same as the last equation above. The Σ formula *defines* the operation of multiplying a matrix and a vector; and the single layer linear network provides a paradigm of the operation.

Part of the convention is that it is the second subscript of the w_{ji} variables that becomes the summation variable when a matrix multiplication is written as a computation of individual values. That is the reason for making i the second subscript to w in the textbooks and in the earlier teach files: it fits in with the standard convention for matrices.

In summary, the computation performed by a single layer linear network is to multiply the weight matrix by the input vector. This is probably one of the easiest ways to give meaning to the idea of matrix multiplication.

3 Writing out matrices as tables

It is sometimes useful to have a convention for writing out the elements of a matrix. In the case of a network with 3 output units and 4 inputs, one might draw up a table of weights like this:

	Input 1	Input 2	Input 3	Input 4
Unit 1	3.2	2.0	-0.5	2.3
Unit 2	-0.4	6.7	1.1	-4.2
Unit 3	1.2	-2.5	0.3	-0.8

Here, for example, -0.4 is the weight on the connection from input 1 to unit 2. (I have just made up some arbitrary numbers.) Therefore for this network, we would have $w_{21} = -0.4$, using our standard convention for ordering subscripts.

The convention for writing down the elements of a matrix in a table is the one adopted here. That is, the first subscript says which *row* of the table an element is in, and the second subscript says which *column* it is in. That is, w_{ji} means $w_{row,\ column}$, which in this case means $w_{unit\_number,\ input\_number}$.

To display a matrix, the elements are written out in a table as above, but without the row and column labels, and the whole thing is enclosed in square brackets (or in a few books, round brackets).

We can summarise this convention by writing, for our 3-unit 4-input network

$$\boldsymbol{w} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix}$$

The vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are written as if they were matrices with a single *column* (and for this reason are sometimes called column vectors). That is, for example

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

and likewise for $\boldsymbol{x}$, except it has 4 elements in this example.

If you write out the matrix multiplication

$$\boldsymbol{y} = \boldsymbol{wx}$$

using the display notation, you get

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

If you look at the original formula for the neural network, you should be able to see that the rule for working out y_i is to take the elements of the i'th *row* of $\boldsymbol{w}$ and multiply each one by the corresponding element of the only *column* of $\boldsymbol{x}$, and add them up. This "row into column" idea is one of the standard ways of presenting matrix multiplication. You should realise, though, that it is merely a result of the convention that is usually adopted for writing down matrices as tables — there is nothing fundamental about it.

You should be able draw a diagram of the 3-unit 4-input network, write the weights in the table above in the right places, invent some input values, and work out some output values (a) by looking at the diagram, (b) by using the Σ formula and (c) by using the matrix display method. And you should get the same results each time. Ideally you might also do it (d) using Matlab and (e) using your own program written in the language of your choice — but that's hardly necessary if you understand what is going on.

4 Matrix-matrix multiplication

Now suppose the single-layer network can be applied to a lot of different input vectors, and we want to specify which one we are dealing with. The obvious thing to do is to add yet another subscript to the original equation to specify a particular input.

$$y_{jk} = \sum_{i=1}^{N} w_{ji} x_{ik}$$

The subscript k is used to specify which of the set of inputs we are referring to. Here the ws don't get another subscript because they're going to stay the same all the time (we're ignoring learning now), but the y does get another subscript because it will be different for each different input vector.

Matrix notation handles this extension very easily. The objects $\boldsymbol{x}$ and $\boldsymbol{y}$ must now be proper matrices rather than vectors, because their individual elements have two subscripts. Given that, the equation above defines matrix-matrix multiplication just as the earlier equation defined matrix-vector multiplication; we can still write

$$\boldsymbol{y} = \boldsymbol{wx}$$

In terms of the display convention, each *column* of the $\boldsymbol{x}$ matrix refers to a different input example, whilst each *row* of $\boldsymbol{x}$ refers to a different input line into the network. Each *column* of $\boldsymbol{y}$ refers to the output from a particular input example, whilst each *row* refers to a different output line from the network.

It should be reasonably obvious from this that a matrix-matrix multiplication is just like treating each column of the rightmost matrix as a separate vector, doing a matrix-vector multiplication on it, and assembling the results into the output matrix. This is like saying that our neural network treats each different input vector separately from all the others.

Matrix multiplication does not get any more complex than this.

5 Some simple matrix operations

There are some simple operations on matrices that you should know the conventions for. *Matrix addition* just means adding each element in one matrix to the corresponding element in another; obviously they must be the same shape. In symbols:

if $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$, then $z_{ij} = x_{ij} + y_{ij}$

Matrix subtraction is similar.

Scalar multiplication of a matrix just means multiplying each element by the same number. In symbols:

if $\boldsymbol{z} = k\boldsymbol{x}$, then $z_{ij} = k\, x_{ij}$

where k is an ordinary number (called a *scalar*).

Sometimes it's useful to swap the order of the subscripts in a matrix. This is called *transposing* it. In terms of the display convention, one writes the rows as columns, and vice versa. A "T" superscript is used to indicate the operation. In symbols:

if $\boldsymbol{z} = \boldsymbol{x}^T$, then $z_{ij} = x_{ji}$

Remember what these symbolic statements represent: the bit before the "then" refers to operations regarded as somehow happening to the matrices as entire objects, whilst the bit after the "then" refers to what happens to individual elements when this operation occurs.

6 Matrix inverses

Given that we can do matrix addition, subtraction and multiplication, what about matrix division? That is, if we can write

$$\boldsymbol{y} = \boldsymbol{wx}$$

to find the outputs given the inputs to our network, can we write

$$\boldsymbol{x} = \boldsymbol{y}/\boldsymbol{w}$$

to express a computation that finds out what inputs caused a given output, as we could for ordinary numbers?

The answer is no, for two reasons. The trivial one is that the “division” notation just isn’t used (or at least it’s almost never used) to mean what we want it to mean here. The more serious one is that the computation might not be possible: there might just not be enough information in ***y*** and ***w*** to say what ***x*** is.

First the notation bit. When this computation can be done, what is actually written is

$$\boldsymbol{x} = \boldsymbol{w}^{-1}\boldsymbol{y}$$

where $\boldsymbol{w}^{-1}$ is called the *inverse* of ***w***, and is itself a matrix. It is used to find ***x*** by doing a matrix multiplication on ***y***.

Second, the problem with whether the computation is possible. It is easy to find an example of when it is not: if there are more inputs than outputs, then the inverse problem (going from outputs to inputs) is said to be *underdetermined* — there’s (usually) not a unique solution. As a very simple example, say the net has two inputs and one output and both weights are 2. Then inputs of x_1=2 and x_2=2 will give an output y_1=8. But so will x_1=4 and x_2=0, and so will any number of other input combinations, so there’s not a single solution to the problem of finding the inputs.

The computation may also be impossible if there are more outputs than inputs. Then, the inverse problem is said to be *overconstrained* and there’s (usually) no solution at all. Again, a trivial example will illustrate this. Suppose there is one input feeding two output units, and again both weights are 2. If y_1 is different from y_2, then there’s no possible value for the input — we can only solve the problem if we assume that the outputs are compatible with the weights, which in this case means that they are both the same, and we can ignore one of them.

Sometimes, however, it is possible to find ***x*** given ***y*** and ***w***: in other words the matrix $\boldsymbol{w}^{-1}$ exists. Given what has just been said, it clearly helps if there are the same number of inputs as outputs; that is, ***w*** is *square*. In addition, w must in some sense transmit all the information in ***x*** through to ***y*** in order for us to be able to go backwards. For square weight matrices the inverse exists unless one network output effectively duplicates the information available in some other outputs. Matrices where the inverse exists are called *nonsingular.*

Computing the inverse of a matrix is an important computational operation. Although we do not often want to literally work out the inputs to a linear network layer from its outputs, there are, for example, training methods based on the inverse of a matrix of partial derivatives (which is beyond our present scope).

What *is* useful to know is that if the inverse of a matrix exists, then it can be computed for particular numerical values, and that almost all numerical packages devote considerable effort to providing routines for this purpose. What you probably do not need to know are the algorithms such routines use — though any textbook on numerical analysis will give a great deal of detail to this topic.

7 Matrix analysis

Matrices play such an important role in many systems, especially simulations of various kinds, that there is a large literature concerned with analysing their properties. In particular, the decomposition of a square symmetrical matrix into *eigenvalues* and *eigenvectors* is an important tool both theoretically and computationally. This is not the place to embark on a discussion of these ideas, though you might meet them in mathematics text books.

It is worth mentioning that there is one particularly useful way of tackling underdetermined and overconstrained inverse problems. This is the *singular value decomposition* of a matrix. When applied to an underdetermined problem (i.e. short fat matrix), it allows us to pick out one of the many solutions. (In fact, the solution it picks out is the one with the smallest sum of squares of the elements of the ***x*** vector.) When applied to an overconstrained problem (i.e. tall thin matrix) it finds an approximate result, in that it finds an input that when fed into the network will produce an output that is as close as possible to the ***y*** that we started with. (As close as possible means that the sum of squares error is the minimum.) The SVD is useful in other ways also; it is worth being aware of the existence of this technique so that you can look it up if you want it.

In general, the analysis of matrices is closely tied in to a geometrical view of mathematics, and in particular the idea of transformations in a vector space. Such increasingly abstract ideas lead to increasing power and generality, but require considerable time and study.

8 Conclusion

This file has introduced the idea of the matrix (and vector) as an object that can be manipulated mathematically in various ways. The central idea is that a collection of quantities can be given a single symbol and manipulated as a single entity, but that what is “really” going on is a set of more elementary operations on the individual elements. It is essential to see the relation between these levels of description.

Matrices are useful both as a notational shorthand, and also in a more fundamental way when properties such as the inverse are exploited. The shorthand aspect has a programming metaphor: writing down a matrix equation is like calling a procedure that takes an array as an argument and carries out an operation on it. If the operation is well defined, the procedure can be treated as a black box, and we do not need to know what happens in detail to the individual elements.

In practice, too much use of matrix shorthand can get in the way. It is often clearer to write down what happens to individual elements, as was done throughout Section 2, and it is easier to keep track of what is meant, especially when calculus is involved. However, as soon as you want to do something equivalent to propagating information backwards through a single-layer network, then matrix analysis is the area you need to look at.

Section 4 Vectors as geometrical objects

This section provides an introduction to vectors as geometrical objects.

Contents

1 Introduction

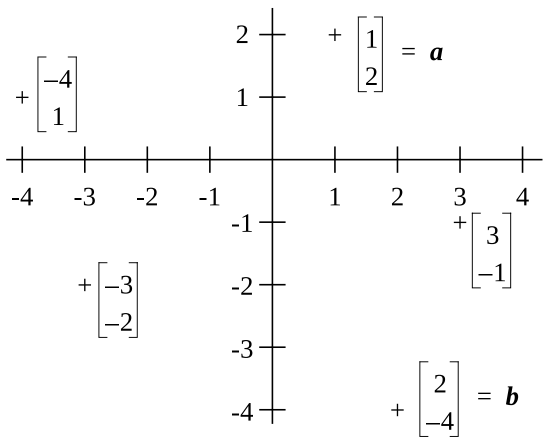
In Section 3, the term *vector* was used to mean, in effect, a matrix with a single column. In other words, a vector was a collection of numbers in a definite order. A vector $\boldsymbol{x}$ was said to have elements x_1, x_2, etc., up to x_N, and the general element was denoted by x_i. That teach file set out the conventions for adding vectors together, multiplying a vector by a scalar, and multiplying a vector by a matrix. One significant application of the last of these operations was given: matrix-vector multiplication is another way of writing down what a single-layer linear neural network does. Vectors were used to represent the inputs and outputs of the network by single symbols.

Vectors have, however, many other uses. One particularly important class of uses is in *geometry*, when vectors are used to represent spatial relationships and operations. Applications of this can be found in many areas, but analysis of the perceptual and motor systems of autonomous agents benefits particularly strongly from vector geometry.

Here, we look at the idea of vectors as geometrical objects in general, before outlining two significant applications in perception and motor control in the next teach file. As with Sections 1 and 2, those already familiar with the main ideas of vectors might skip straight away to Section 5.

2 Position vectors

The idea of representing a point in space using *coordinates* is probably a familiar one. The coordinates of a point can be assembled into a single computational structure, which is then a *vector*. Using the matrix notation of Section 3, the correspondence between some points in the 2-D plane and their vector representation works like this:


$\begin{bmatrix}1\\2\end{bmatrix} = \boldsymbol{a}$
$\begin{bmatrix}-4\\1\end{bmatrix}$
$\begin{bmatrix}3\\-1\end{bmatrix}$
$\begin{bmatrix}-3\\-2\end{bmatrix}$
$\begin{bmatrix}2\\-4\end{bmatrix} = \boldsymbol{b}$
-4 -3 -2 -1 1 2 3 4
2 1 -1 -2 -3 -4

The points themselves are marked with the "+" symbol, with the corresponding vectors written using matrix notation beside them. I have also given two of them names, $\boldsymbol{a}$ and $\boldsymbol{b}$.

It is straightforward to extend this idea to 3 dimensions.

I have not given names to the axes in the diagram. The convention generally used is that the first component of the vector represents the coordinate along the horizontal axis, which in turn is often called the *x*-axis. The second component represents the coordinate along the vertical or *y*-axis. With these names for axes, the components of the vector ***a*** might be called a_x and a_y instead of a_1 and a_2 as we have been doing so far. If we wish to retain numerical subscripts, it might be more sensible to refer to the axes as the 1-axis and the 2-axis.

In fact, both conventions are in use. For geometrical work in 2 and 3 dimensions, *x*, *y* and *z* subscripts are common, and the axes are labelled with these letters. However, for more abstract uses of vectors (such as to represent quantities in a neural network), numerical subscripts tend to be used, and if pictures are needed the axes might be designated by numbers too. Here, I am going to stick to numerical subscripts because it makes generalisation to abstract uses easier, fits in with the notation of the earlier teach file, and corresponds to what happens in practice when you represent vectors by data structures in a computer program.

Sometimes a vector is drawn using an arrow. For position vectors like ***a*** and ***b*** above, the arrow would be drawn starting from the origin (the intersection of the axes), with its tip at the point in question. This is simply an alternative convention for indicating a position. A vector does not intrinsically have a "start" and a "finish".

This use of vectors to represent positions in space is a paradigm for all their other applications. However, vectors are used to represent many other kinds of things — both physical quantities such as velocities, which are measured in ordinary space-time, and more abstract things such as the state of a network, which are measured in a higher dimensional space.

The fundamental mathematical concept of a vector is actually more general and abstract than this. In practice, however, the crucial ideas are that position in space is exactly the kind of thing that vectors can represent, and that a vector can in turn be represented by a column of numbers.

3 The geometry of simple vector operations

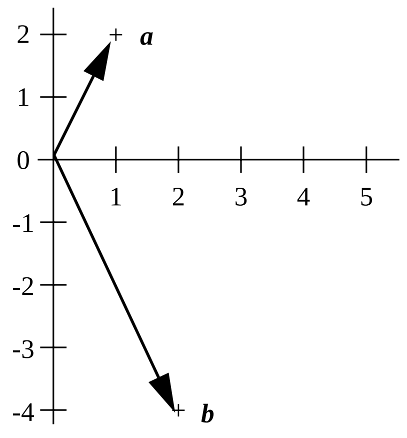
3.1 Adding and subtracting vectors

From Section 3, part 5, you should be able to see that for the two named vectors in the diagram,

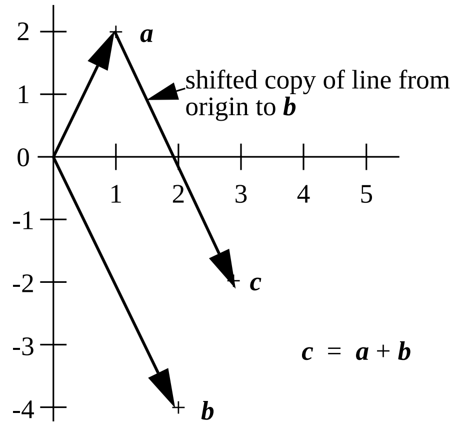
$$\boldsymbol{a} + \boldsymbol{b} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 2 \\ -4 \end{bmatrix} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$$

because adding the vectors means, by definition, adding their components individually.

There is a simple geometrical interpretation of the operation. First draw the diagram above with arrows or lines from the origin to each of the points:



Then move a copy of one of the lines so that it starts from the end of the other line, without changing its length or orientation. Here we move a copy of b so that it starts from the end of a:



The new position, marked ***c***, obtained by this graphical operation, is represented by the arithmetical operation of adding the vectors by components. The same result would have been obtained if a copy of ***a***'s arrow had been tacked onto the end of ***b***'s. This result is general: the formal operation of vector addition corresponds to composing the 2-D position vectors together.

A classical application of this is in navigation. Adding a plane's velocity through the air to the wind velocity gives the plane's velocity over the ground. Velocities are not position vectors; but they do add like position vectors and so the calculation, done either numerically on the components or graphically, gives the right answer. A more esoteric application is in the superposition of wave functions in quantum mechanics; later we will see an application to robot navigation.

Subtracting one vector from another is done by components, like addition. You should be able to figure out its geometrical meaning; bear in mind that in the diagram above ***a*** = ***c*** - ***b***.

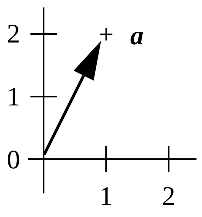
3.2 Multiplying a vector by a scalar

In Section 3 I said that multiplying a vector by a scalar (an ordinary number) meant multiplying each element by that number. Multiplying a vector by a positive scalar *k* corresponds in geometrical terms to making the arrow *k* times as long but giving it the same direction. If you think about 2***a*** = ***a*** + ***a***, and draw a diagram like that above, this should become reasonably obvious.

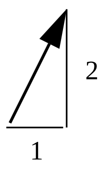
Multiplying a vector by -1 makes the arrow point in the opposite direction (or puts the point it represents on the opposite side of the origin).

3.3 The length of a vector

How far is the point indicated by the vector ***a*** from the origin? This is easy to answer from the diagram in which ***a*** was first defined:



which contains a triangle with this shape and dimensions:



Applying Pythagoras' theorem to the triangle gives the length of the hypoteneuse (the sloping side) the value of √(1^2+2^2) or √5 (where √ means taking the square root). This is the distance required. In terms of the components of ***a***, the formula for the distance is

$$\sqrt{a_1^2 + a_2^2}$$

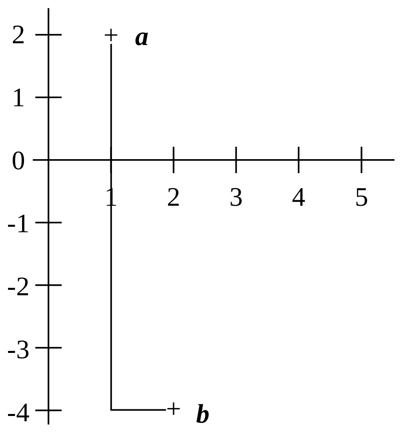
If ***a*** is thought of as representing the line from the origin to the point, then this formula gives the length of that line, and so it is called the length of vector. The formula generalises to many dimensions. If ***y*** is an *N*-dimensional vector, then its length is given by

$$\sqrt{\sum_{i=1}^{N} y_i^2}$$

This is more precisely called the *Euclidean norm* of the vector. It is sometimes written as |***y***| or ||***y***||; occasionally also *y* (in light type) is used to stand for the norm of the vector ***y*** (in bold type).

3.4 The distance between two points

How far apart are the points marked ***a*** and ***b*** in the diagram above? Again, one can draw a right-angled triangle and apply Pythagoras' theorem. In this case, the picture is:



where I have drawn on two sides of the triangle — the hypoteneuse is the line from point ***a*** to point ***b***. The distance between them is thus √(1^2 + 6^2) = √37. The general formula for 2-D vectors is

$$\sqrt{(b_1 - a_1)^2 + (b_2 - a_2)^2}$$

Note that the components are subtracted first, then the differences are squared.

This is a particular case for the distance between two points represented by vectors. The general formula for the distance between ***x*** and ***y*** is just ||***x*** - ***y***||. Putting the definition of the norm together with the procedure for subtraction should allow this to make sense. Now, ***x*** and ***y*** can be in a space with any number of dimensions.

The Euclidean norm has a nice geometrical meaning in 2-D and 3-D, but you have already met it in a different context. Look at the error function for a neural network in part 4.1 of Section 2. The error *E* is the square of the Euclidean distance between the output vector ***y*** and the target vector ***t***.

4 Multiplying a vector by a matrix

In Section 3 we looked at what matrix-vector multiplication meant in terms of computations on the components, and we noted the relationship of this to the operation of a linear neural network. There is also a geometrical way to look at this, though it does not yield a single simple picture. The general idea is that the multiplication leads to a geometrical transformation of one vector into another, and the nature of the transformation can be related to properties of the matrix.

I will not give a comprehensive treatment of this, but will give examples of two important special cases that illustrate the idea.

4.1 Diagonal matrices

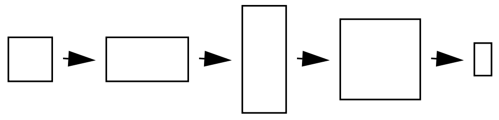
First, consider multiplying ***a*** by a matrix that only has non-zero elements on its top-left to bottom-right diagonal (called a *diagonal matrix*). Since ***a*** has only two components, and we want the result to be another vector like ***a***, the matrix has to have 2 rows and 2 columns. It looks like this:

$$\begin{bmatrix} p & 0 \\ 0 & q \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} pa_1 \\ qa_2 \end{bmatrix}$$

(You should be able to verify this equation by applying the matrix-vector multiplication rule from Section 3.) Here p and q are ordinary numbers.

What does this do to ***a***? For a start, putting $p = q$ is the same as multiplying ***a*** by a scalar, so this just stretches ***a*** out by a factor p. If p and q are different, then, roughly speaking, ***a*** is stretched out along the 1-axis by a factor p and along the 2-axis by a factor q. If it is not clear what this means, try plotting the result for the vector ***a*** (with components (1, 2)) and various different values for p and q. The numbers p and q are sometimes called expansion factors, though if they are less than 1 they cause contraction rather than expansion.

Suppose a shape is represented by a number of points on its periphery, and each of these is represented by a vector. Multiplying each vector by the same matrix will produce a new shape. For a diagonal matrix, the shape may be expanded or contracted, and it might be squeezed up or stretched out more along one axis or another. Thus diagonal matrices can make transformations with effects like this:



If $p = q$ the shape is simply expanded or contracted, but otherwise its *aspect ratio* (the ratio of its width to its height) changes.

Above, I wrote the components of ***a*** as (1,2). Strictly, I should have written them in a column with square brackets, but it gets tedious laying those out. I will sometimes therefore write the components in the text line in round brackets; this should not cause any confusion, and in any case the convention is often used in books.

4.2 Rotation matrices

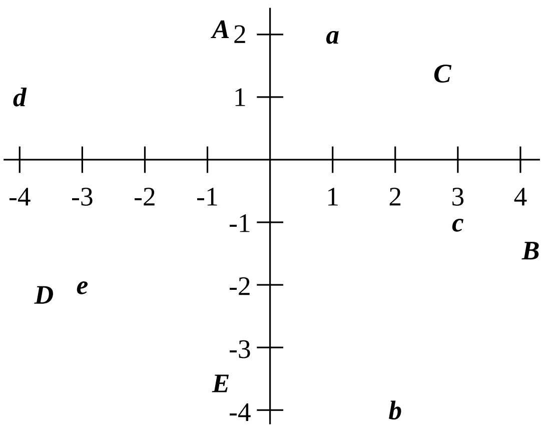
Now consider multiplying the vector a by the following specific matrix:

$$\begin{bmatrix} 0.7 & -0.7 \\ 0.7 & 0.7 \end{bmatrix}$$

For ***a*** with components (1, 2), you should be able to verify that the multiplication gives a vector with components (-0.7, 2.1). Multiplying this matrix with the other vectors in the initial diagram gives the following transformations when the multiplication is done (remember, these ought to be written as column vectors, but I'm being lazy):

$$(-4, 1) \rightarrow (-3.5, -2.1)$$
$$(-3, -2) \rightarrow (-0.7, -3.5)$$
$$(3, -1) \rightarrow (2.8, 1.4)$$
$$(2, -4) \rightarrow (4.2, -1.4)$$

Plotting these transformations on the original diagram has this effect, marking the original points by lower case letters and the new points by the corresponding upper case letters:



If you join the corresponding points, you will see that they have all been approximately rotated about the origin by about 45˚. This suggests that a matrix can effectively act to *rotate* vectors. If the set of vectors defined a shape amongst them, then this shape would get rotated by the matrix multiplication.

In fact the matrix I have used to demonstrate this is only approximately a pure rotation matrix; if you plotted out the results really accurately you'd detect a little contraction as well. To construct a pure rotation matrix for 2 dimensions, you choose an angle you want to rotate through, (call it θ), then get the matrix element values using

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

This produces an anticlockwise rotation of the vectors if θ is positive. If $\theta = 45^\circ$, then $\cos\theta = \sin\theta =$ approximately 0.7, which is why the example above gave the results that it did. A matrix like this produces no expansion or contraction or change in aspect ratio. Such matrices are useful in many areas; one currently very important use is in computer graphics, where objects represented by sets of position vectors must often be rotated for display from different viewpoints.

Rotation matrices generalise to 3-D and higher dimensions.

In general, a transformation produced by a matrix multiplication can be broken down into a rotation through some angle, followed by multiplication by a diagonal matrix, followed by another rotation. This turns out to be the geometrical interpretation of the singular value decomposition mentioned at the end of Section 3. And if a matrix cannot be inverted, it means that it transforms more than one input vector into the same output vector, so that there is no way of going backwards unambiguously.

4.3 Linear transformations

The multiplication of a vector by a matrix is a *linear transformation*. What this means is that if you transform two vectors separately, and add the results together, you get the same answer as if you add the two vectors together first, and then transform the sum. In symbols:

$$\boldsymbol{M}(\boldsymbol{x} + \boldsymbol{y}) = \boldsymbol{M}\boldsymbol{x} + \boldsymbol{M}\boldsymbol{y}$$

The mathematics of transformations that have this property is fundamentally far simpler than that of transformations that do not. Conversely, using non-linear transformations can yield richer behaviour (e.g. in the context of neural networks) than linear transforms can.

4.4 Coordinate transformations

There is another way of looking at the effect of a matrix multiplication. Instead of thinking of it as moving a position vector around in a coordinate system, you can think of the vector as being fixed and the axes as changing. The matrix multiplication is a way of expressing what the vector is in a different coordinate system. This interpretation can be very useful; whether it is appropriate depends on the application, but you need to be clear about which interpretation you are using at any time.

The effects of the matrices described above on the coordinate system are the opposite of their effects on the vectors. For instance, the diagonal matrix shrinks the 1-axis by a factor of *p* and the 2-axis by a factor of *q*, whilst the rotation matrix turns the axes clockwise through an angle θ. The numerical calculations are of course identical whichever interpretation is in use.

5 The dot product and basis vectors

5.1 The dot product

Suppose we take two vectors and multiply the transpose of one by the other. Taking the transpose means swapping rows and columns, so an ordinary column vector becomes a row vector. It looks like this:

$$\begin{bmatrix} a_1 & a_2 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

which, if you apply the normal multiplication rule and write out what you get, comes to

$$a_1 b_1 + a_2 b_2$$

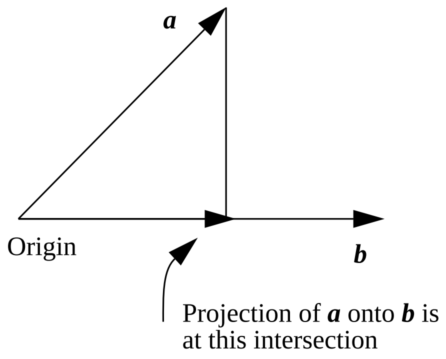
The generalisation of this to *N*-dimensional vectors ***x*** and ***y*** is

$$\sum_{i=1}^{N} x_i y_i$$

This is called the *dot product* of the vectors. It is an example of a kind of relationship between vectors called an *inner product*.

You have met the dot product already in a different guise. A single linear unit in a neural network forms the dot product of its weights vector and its input vector (part 2.1 of Section 2).

Does this have a geometrical significance? It does, but first we need to explain what is meant by projecting one vector onto another. Suppose we have two position vectors, say ***a*** and ***b***, and we draw the arrows from the origin to the points they represent. Then we draw a line from the tip of the ***a*** arrow, at right angles to the ***b*** arrow, and mark where this perpendicular meets the ***b*** arrow. This point is the *projection* of ***a*** onto ***b***. In a picture:



The dot product of ***a*** and ***b*** is the length of ***b*** times the length of the projection of ***a*** onto ***b***. If the intersection occurs on the opposite side of the origin to the point ***b*** (i.e. ***a*** is moved to the left in the diagram above so that you have to project ***b***'s arrow backwards to get an intersection), then the dot product is negative. The relationship is symmetrical — you can swap the ***a*** and ***b*** and get the same result.

If ***a*** and ***b*** point in the same direction, then their dot product is just the product of their lengths. If they point in exactly opposite directions, the dot product is minus the product of their lengths. If they are at right angles, the dot product is zero.

Another formula you might see for the dot product is

$$\boldsymbol{a} \cdot \boldsymbol{b} = \|\boldsymbol{a}\| \, \|\boldsymbol{b}\| \, \cos(\text{angle between } \boldsymbol{a} \text{ and } \boldsymbol{b})$$

Textbooks give the proof that this is the same as the definition above in terms of the components. For high-dimensional vectors, this formula is used to *define* what is meant by the angle between two vectors.

The dot product is biggest for two vectors of fixed length if the angle between the two vectors is zero — that is, one of the vectors is just a scalar constant times the other. For example, if a linear unit in a neural network has only a fixed amount of weight to distribute (in the sense that the sum of the squares of its weights is fixed), it can optimise its response to a given input by making the weights match the inputs — if the weights are proportional to the given inputs, then the unit will be giving as big an output as possible.

5.2 Basis vectors

The dot product gives another way of thinking about linear transformations. Each row of a matrix can the treated as the transpose of a column vector. Then when we multiply that matrix by a vector, what we do is to form a set of dot products. The first component of the output vector is the dot product of the input vector with the first row of the matrix, and so on.

The rows of the matrix are sometimes called *basis vectors* (though the matrix must be invertible for this to make proper sense). The elements of the new vector are the projections of the old vector onto each of the basis vectors in turn, multiplied by the lengths of the basis vectors. The coordinate transformation produced by the matrix can thus be seen as a set of projection operations onto a set of basis vectors.

In fact, this allows us to give abstract definitions of vectors that do not depend on how they are represented. The ordinary components of the vectors, that we have used so far to represent them, are actually just the dot products of the vectors with the standard basis vectors, which have components (in 3-D) of (1,0,0), (0,1,0) and (0,0,1).

This idea will not be pursued further here, but it provides important tools for analysing some systems. To go on in this direction requires an analysis of the abstract ideas of *vector spaces* — Horn & Johnson (see Section 3) gives information.

Section 5 Vector applications

This section complements Section 4. It provides two examples of the application of ideas in vector analysis.

Contents

1 Introduction

The most immediate and obvious application of vectors is the representation of geometrical relationships in ordinary 3-dimensional space. This is not surprising, since the prototype vector is a position vector. This file concentrates on this kind of use for vectors, but it is worth bearing in mind that much more abstract entities also lend themselves to vector representation, as we have already seen in the case of input and weight vectors for neural networks.

The need to represent geometrical relationships arises particularly often in those parts of robotic or a-life systems which have to interact with the physical world (or a simulation of it). Perceptual and motor control systems frequently use vector representations of space (and sometimes of space-time).

Here, I focus on matrix-vector multiplication, since this linear operation is one of the most important in applied mathematics.

2 A vector velocity field

2.1 The idea of a velocity field

The common functions, such as sin and log, take a number as argument and produce (or "map onto") another number. There is nothing to stop us extending the idea of a function to include vector functions, which take one or more vectors (and maybe numbers as well) as arguments and which produce vectors as their result.

If the input to such a function is a position vector, and the result is another vector, then we have a *vector field*. For example, if the wind speed and direction at each point at ground level in Sussex is represented by a vector, the set of vectors forms a 2-D vector field. Graphically, the wind over Sussex might be displayed as arrows drawn on a map, and this provides a way to visualise the vector field idea.

The wind velocity vector at a point has two components (one might represent it using northerly and easterly components for example). It's not absolutely obvious that velocities should be manipulated in the same sort of way as positions, but in fact it's the case that the same vector rules are appropriate for velocities, and the rules have straightforward physical meanings (at least as long as the speeds are small compared to that of light). In other words, if positions are appropriately represented by vectors, then so are velocities.

If the wind velocity was represented by a vector $\boldsymbol{v}$, and position on the ground by a vector $\boldsymbol{r}$, then we could write

$$\boldsymbol{v} = \boldsymbol{f}(\boldsymbol{r})$$

where $\boldsymbol{f}$ is the name of the vector function that assigns a wind velocity to each point on the ground. Of course in this example the function would not be a simple thing to write down — the best you could probably do would be to tabulate an approximation to it — but the idea that you can associate a velocity vector with each position vector in the *domain* of the function (here Sussex) is what matters. Often the shorthand notation $\boldsymbol{v}(\boldsymbol{r})$ is used to indicate that $\boldsymbol{v}$ depends on $\boldsymbol{r}$, without giving the function a separate name.

If the vertical component of the wind velocity was represented as well, and the position included height above the ground, then we would have a function from 3-D position vectors to 3-D velocity vectors. (The manipulation of large arrays representing approximations to such functions is one of the main tasks of the Meteorological Office's supercomputers.)

2.2 An optical velocity field

Now I give a specific example of a velocity field that is useful for studies of perception in the control of robot and animal locomotion.

Suppose a camera attached to a robot is moving through the world. The image formed by the camera will be changing. If all the visible objects in the environment had closely textured high-contrast pattern on them, we could track the images of features of the pattern across the image plane of the camera. We can imagine drawing arrows on the image representing the speed and direction of motion of features at a given moment. If this set of arrows could be made very dense, so that an arrow could be associated with any point on the image plane, then we would have a 2-D vector field.

This field is closely related to what is known as the optic flow field for the moving camera.

In practice, an approximation to the image velocity field is easy to obtain by hand. You simply get two images from different positions of a camera (not too far apart), superimpose them, and join corresponding features with arrows. Each arrow effectively represents an image velocity vector.

In general, this velocity field will be a complex function like the wind field over Sussex — there is not a simple formula for it. However, under certain circumstances there is a good approximation which does obey a simple rule. If the following apply:

(a) the camera is viewing a smooth surface;

(b) the field of view is reasonably small;

(c) the camera is panned and tilted so that it tracks a feature at the centre of the image, which is taken as the origin for position;

then the image velocity field is approximately a linear function of image position. What this means is that if position in the image is represented by $\boldsymbol{r}$ and image velocity by $\boldsymbol{v}$, the relationship is just a matrix multiplication:

$$\boldsymbol{v} = \boldsymbol{M}\boldsymbol{r}$$

where $\boldsymbol{M}$ is a 2x2 matrix. The 4 components of $\boldsymbol{M}$ depend on the direction the camera is moving and the slant and tilt of the surface. To write down this relationship is not trivial and will not be done here (references are available on request).

2.3 Matrix multiplication as a field operation

This example gives us another opportunity to look at the geometrical use of matrix multiplication. Particular combinations of components of the $\boldsymbol{M}$ matrix produce very different and distinctive image velocity fields.

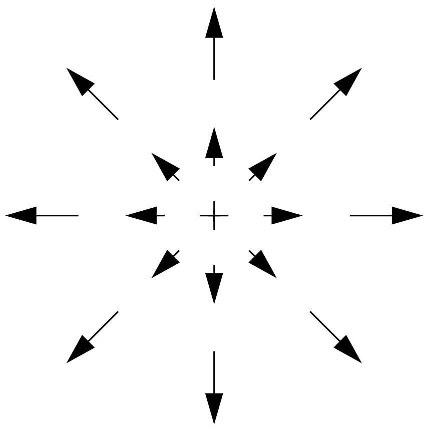
Suppose, for example, that $\boldsymbol{M}$ is

$$\begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

Then it is easy to work out some examples of mappings from $\boldsymbol{r}$ to $\boldsymbol{v}$:

$$\begin{array}{ccc} \boldsymbol{r} & & \boldsymbol{v} \\ (1, 0) & \rightarrow & (0.5, 0) \\ (1, 2) & \rightarrow & (0.5, 1) \\ (0, -1) & \rightarrow & (0, -0.5) \\ & \text{etc.} & \end{array}$$

Given such a table, it is easy to plot out some samples of the image velocity field as an arrow diagram. One naturally draws the arrow representing $\boldsymbol{v}$ at the position given by $\boldsymbol{r}$, just as one would for the wind arrows on a meteorological map. If you do this for the table above and some more examples, you should get a picture looking roughly like

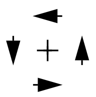


where the arrows get longer the further out from the origin you go. In other words, the diagonal matrix produces a *dilating* flow pattern.

This kind of flow pattern is generated by motion towards a surface.

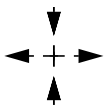
Note the difference between this and matrix multiplication viewed as a transformation, as in Section 4 part 4. There, the idea was that a shape was mapped into another shape, or that a different coordinates system was used to represent the same shape. Here, a function from the position vectors maps onto a different kind of thing: a velocity. The actual matrix manipulations are, of course, the same, but their interpretation is different.

The effects of other forms of ***M*** are easy to calculate. You should be able to check, by calculating one or two vectors in each case, that

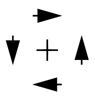
$\begin{bmatrix} 0 & -0.5 \\ 0.5 & 0 \end{bmatrix}$ produces something like 

This *rotational* flow field can be produced by spinning the camera about its axis. Note that the matrix is just 0.5 times the matrix for a rotation of 90˚ (see Section 4, part 4.2), so it is not surprising that it has this effect.

The matrix

$\begin{bmatrix} 0.5 & 0 \\ 0 & -0.5 \end{bmatrix}$ produces something like 

which is one component of a *shear* field, and

$\begin{bmatrix} 0 & 0.5 \\ 0.5 & 0 \end{bmatrix}$ produces something like 

which is the other component of shear. Shear flow fields are produced when a camera moves sideways in front of a slanted surface. For instance, the image of the ground in front of a mobile robot has the first component of shear combined with dilational flow, whilst the image of the ground to the side has the second component combined with rotational flow.

The details are not important; the point here is to illustrate the idea that a vector field, in this case generated by one of the simplest vector functions (multiplication by a matrix), can represent a rich variety of patterns — in this case patterns related to an important part of the perceptual process of a mobile agent.

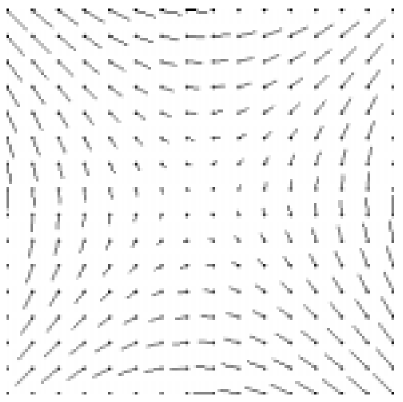
In practice, the elements of the matrix have to be estimated from an image sequence. This can be done in a variety of ways, but typically a model, represented by the linear equation above, is fitted to the partial derivatives of image intensity with respect to space and time coordinates. The matrix elements can then be used in a variety of ways, either to help build a 3-D representation of the environment, or more directly in a motor program.

2.4 Flow patterns illustrated

You may be able to generate representations of flow patterns using a package such as Matlab, or the Poplog system. For example, using such a program the matrix

$$\begin{bmatrix} 0 & 0.1 \\ 0.1 & 0 \end{bmatrix}$$

generates the flow field representation shown here:

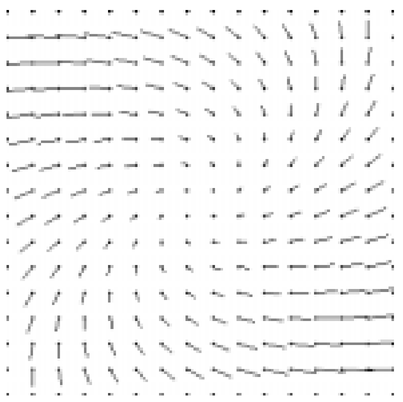


The program used does not draw the arrow heads, but puts a dot at the *base* of each arrow. The dots mark the positions corresponding to the ***r*** vectors.

A more complex example generated by the matrix

$$\begin{bmatrix} 0.1 & -0.1 \\ 0.05 & 0.05 \end{bmatrix}$$

which produces a mixture of shear, dilation and rotation, appears as



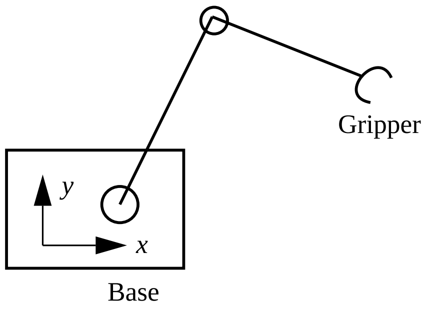
3 Coordinate transformations for control

A second application is to the control of robot arms.

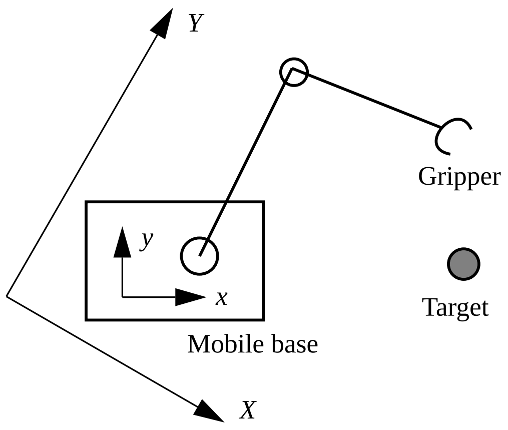
This will merely be outlined here to give the general idea; robot control textbooks give details and examples. I will also mention the use of alternatives to the Euclidean coordinate system.

3.1 Simple robot arm kinematics

Suppose a robot's gripper is operated from some base, and designed so that it can be moved on command to a given position above a plane, expressed in (x, y) coordinates, relative to that base. (Some piece of electronics on the base converts these coordinates to the signals which are actually sent to some motors to move the arm.)



Provided there is a program that can work out the positions to send the arm to, there's no problem with this. However, the arm might not be able to reach everything that is needed, so it is mounted on a mobile base. The controller for this can turn it to face in any direction and can move it around the lab. Its position in the lab is also expressed in Euclidean coordinates, relative to some axes fixed to the floor, which will be denoted by (X, Y). The orientation of the base is indicated by an angle θ, which is the angle anticlockwise from the X-axis on the floor to the x-axis on the base. Thus we have something like



Here θ is about 30˚. Note that it is independent of the base's position in the room.

The question is: if the position of an object in the room is specified in terms of its (X, Y) coordinates (i.e. its position on the floor), how do we move the gripper to it? Since the gripper control works in terms of (x, y), we need to work out where the target is relative to these coordinates. Although you could probably work out the answer using elementary geometry and trigonometry, the formula is very simple when expressed in terms of vector transformations. Suppose the robot's base (to be precise, the origin of its (x,y) coordinates) is at position $\boldsymbol{B}$ relative to the floor coordinate system, and the target is at position $\boldsymbol{T}$ in this coordinate system. What we need is the target's position in the *base's* coordinates, which we will denote by $\boldsymbol{t}$. Then we can move the gripper directly to it, provided it is in range.

The formula to get from $\boldsymbol{T}$ and $\boldsymbol{B}$ to $\boldsymbol{t}$ is much simpler than the one you might work out using elementary mathematics. It is

$$\boldsymbol{t} = \boldsymbol{R}(-\theta)(\boldsymbol{T} - \boldsymbol{B})$$

Here $\boldsymbol{R}(-\theta)$ means the 2x2 matrix that rotates a vector anticlockwise through an angle -θ, or equivalently clockwise through an angle θ. The multiplication is a matrix-vector multiplication. From Section 4, part 4.2, we know that

$$\boldsymbol{R}(-\theta) = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

(changing the signs, and remembering that cos(θ) = cos(-θ) and sin(θ) = -sin(-θ)). The formula is easy enough to evaluate in a computer program, possibly using library routines for the matrix operations.

You should be able to see why the formula is correct: $\boldsymbol{T}$-$\boldsymbol{B}$ is the vector from the base to the target in the floor coordinate frame, and to express it in the base coordinate frame, which has been rotated anticlockwise by θ, the vector is in effect rotated clockwise by θ.

Why is this more useful than elementary geometry to do the same task? First, it is quicker and easier to write down and implement; provided you know what a rotation matrix looks like, or you know where to find out, you can have a procedure for the conversion set up in a few minutes.

Second, it is easier to manipulate. For example, we might want to go in the other direction. The robot might have a sensor on its base that detects the target and produces the components of $\boldsymbol{t}$. We might then want to know $\boldsymbol{T}$ in order to plot the position of the object in a map that is being built of the lab. The equation can easily be reversed to yield

$$\boldsymbol{T} = \boldsymbol{R}(\theta)\boldsymbol{t} + \boldsymbol{R}$$

(by default, multiplications are always done before additions). This equation could be got either by reasoning about the situation, or simply by solving the first equation for ***T***. Since ***R***(θ) clearly reverses the effect of ***R***(-θ), we can note in passing that ***R***(-θ) is both the inverse and the transpose of ***R***(θ).

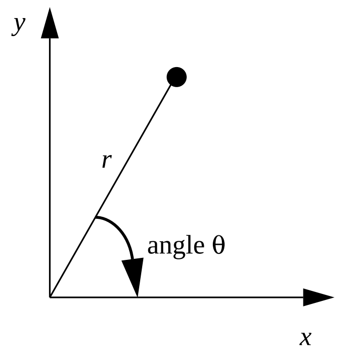
Third, the method generalises better, in a variety of ways. The most obvious is to 3-D: the equations stay the same, though the rotation matrix becomes 3×3. It is also easy to incorporate additional links into the chain. For simplicity, we assumed that the gripper controller could simply move the gripper to given coordinates; in practice, this would be achieved by having several separate links in the arm. For each of these there is a coordinate transformation so that the target positition could be expressed in terms of that particular link's controller. For a chain of links, the equation for the target position can be applied recursively as information is passed along the chain — each link knows its own orientation relative to the previous one, and so can work out the target position in its coordinate system. And indeed, the coordinate systems that are used do not even have to have the same units, or have right-angles between their axes: these differences can be taken into account by using transformation matrices that are not pure rotation matrices. Then, the matrix inverse really comes into its own if the equations have to be solved to find the reverse transformations.

There are some tricks that are commonly used in this kind of computation. One important one is the use of *homogeneous coordinates*, which allow the operations in the equations above (a vector addition and matrix multiplication) to be carried out in a single multiplication step. This is done at the expense of including an extra element in the representation of each vector, and an extra row and column in each matrix. The extra vector elements are in fact redundant, but allow the vector to be added to be specified in the additional matrix elements. The technique will not be elaborated here, though it is not particularly difficult; it can speed up the algebraic manipulations needed for complex systems.

The central idea in this section is that of a *frame of reference*. Any physical vector must be expressed relative to such a frame but a given frame may not be suitable for all the operations that need to be carried out. Transformations between frames are therefore an essential operation.

3.2 Alternative coordinate systems, especially polar coordinates

So far, vectors have been represented using Euclidean (*x*,*y*) coordinates. It is often useful to represent them in other ways; one of the most common is *polar coordinates*. In 2-D polar coordinates, a position is represented by its distance from the origin (often denoted by *r*) and the angle a line joining it to the origin makes with the *x*-axis (often called θ). Such a coordinate system might be used because it corresponded better to the physical setup of a sensory or motor system; wherever there is a natural centre (such as the eye position in a visual system), polar coordinates merit consideration.



The relationships between polar and Euclidean coordinates are:

$$r = \sqrt{x^2 + y^2}$$
$$\theta = \mathrm{atan}(y/x) \text{ if } x \text{ is positive}$$
$$\theta = -\mathrm{atan}(-y/x) \text{ if } x \text{ is negative}$$

assuming that atan (also called arctan) returns a result in the range -90˚ to +90˚ (or -π to +π radians). In the maths libraries provided with almost all computer languages there is a function, often called `atan2`, which does the θ calculation given *y* and *x*. This should always be used in preference to doing the division and calling `atan` or `arctan` explicitly.

To go in the opposite direction, use

$$x = r\cos\theta$$
$$y = r\sin\theta$$

Note that if a vector is expressed in terms of *r* and θ, the rules for addition and multiplication no longer apply. Essentially, you have to translate it to Euclidean coordinates before adding it to another vector by components or transforming it using matrix multiplication.

The transformation between polar and Euclidean coordinates is *nonlinear* because, for example, multiplying both *x* and *y* components by a constant multiplies *r* by that constant but does not change θ.

Often, however, it is necessary to transform small *changes* in a vector, represented in polar coordinates (or some other system), between coordinate frames. This might arise in the robot arm control system if we wanted to know what direction and speed the gripper will have relative to the floor if a motor somewhere in the middle of the arm is actuated at a given rate. A transformation of changes or velocities, can be carried out by a matrix multiplication without calculating the Euclidean coordinates. The elements of the matrix are derived from the coordinate relationships (they are actually partial derivatives), and such a matrix is known as a *Jacobian* matrix. Books such as that by Boas (reference in Section 1) go into considerable detail about this. Note that the inverse of the Jacobian (if it has one) may indicate how to activate the motors to produce a desired motion.

You may encounter the idea of *linearising* a problem by considering small changes in a quantity rather than the quantity itself. The use of the Jacobian matrix to calculate gripper velocities is a good example of this.

Polar coordinates can be extended to 3-D. Other forms of non-linear coordinate transformations can be found — for example log-polar coordinates are useful in some areas of vision.

Section 6 Numerical integration of differential equations

This section provides an elementary introduction to the ideas behind the numerical integration of differential equations.

Contents

1 Introduction

Often, the physics of a situation — or of a simulation — tells us how variables *vary* with one another, but does not directly tell us what values the variables have. The rule for how the variables co-vary might be expressed as a *differential equation* (that is, an equation involving derivatives), or it might be that we are simply have some method of obtaining values of derivatives. In either case, what we often want to do is to work out the values of the variables themselves.

For example, a mobile robot might be able to measure its velocity — it knows how position varies with time — but it might need to know its actual position in order to determine when it has reached its target, or which way to turn towards it. To estimate its position, it could use a kind of dead reckoning: it could add up the distance covered in successive steps, assuming each step is made at a constant velocity. Adding up small steps is known as *integration*; in mathematics it is carried out on infinitesimal steps, and is formally the inverse process to differentiation.

Sometimes, differential equations can be integrated symbolically, so that we obtain a straightforward formula for the variables in question. Here, though, we look at the case where this is not possible, either because the equation is too difficult to integrate symbolically, or because the derivatives are not given by a formula at all, but are tabulated or measured.

Textbooks on numerical analysis deal with the techniques needed in detail. A good example is "Numerical Analysis", by R.L. Burden & J.D. Faires (3rd Edition, PWS Publishers, 1985) — see chapter 5. A summary and practical advice can be found in "Numerical Recipes in C", by W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling (Cambridge University Press 1988, later reprints). Anyone needing to seriously use these techniques will need to spend time with at least one such book; in this file the aim is only to illustrate the underlying principle — which is in fact very simple.

2 Euler's method for initial value problems

A mobile robot moves down a corridor (a 1-D space), measuring its speed at successive time steps. We need to know how far down the corridor it is at any given time, perhaps so that its position can be displayed on a monitor, or so that we can stop it when it has gone far enough.

If the robot's position is $x(t)$ at time t, then its speed $v(t)$ is given by the simple differential equation

$$v = \frac{dx}{dt}$$

We assume that we know where the robot starts from. This is called an *initial value problem* because the differential information is used to work out how the state evolves from a specified initial state.

There is an obvious way to integrate the equation to get values of x given values of v. Suppose v is sampled regularly at $t = 0$, $t = 1$, $t = 2$, etc., and that v changes slowly enough that we can assume that it is constant during the time between samples. In fact, we will just assume that v keeps the value it has when one sample is taken until the next sample is taken. Then the distance travelled between time $t = 0$ and time $t = 1$ is just $v(0)$, and so on. If the robot starts at $x = 0$ at time $t = 0$, we can just compute the distance travelled in the first time step, then in the second, and so on, and add them up to get the current position. Negative speeds mean going backwards.

For example, if the speed is given by the middle column below, then this method gives the positions in the right hand column

t	v	x
0	2	0
1	4	2
2	3	6
3	2	9
4	0	11
5	-1	11
6	-2	10
7	-3	8

and so on. The calculation is trivial and generalises to

$$x(t) \ = \ x(t-1) + v(t-1)$$

If the interval between samples is not 1 unit of time, then the velocity must be multiplied by the interval to get the distance covered. We would write

$$x(t) \ = \ x(t-T) + Tv(t-T)$$

where T means the interval between samples.

This is Euler's method for integrating the differential equation. It is evident that the shorter the time step that is used, the more accurate the results will be. Also, errors will accumulate, so the overall error will increase as time goes on. It shares these characteristics with all other numerical integration methods, but is extremely easy to implement. In a sense, it is the model for all the more sophisticated algorithms that you will encounter. Although numerical analysis books describe many more complex methods for integrating differential equations, Euler's method and simple variations on it are the workhorse of many physical simulations.

Euler's method extends to higher-order differential equations (those with multiple derivatives). For example, we might know the acceleration of the robot, not its speed (though we would need to know the initial speed). Euler's method can be applied twice, first to integrate the acceleration to get the speed, and then to integrate the speed to get the position.

Euler's method also applies to vector equations. If vectors are represented using rectangular coordinates, then each component can be integrated independently of the others. Thus a robot that measures its speed and direction can integrate them to find its position in 2-D or 3-D space.

A second example of the application of Euler's method occurs in recent work in computer vision. One of the major difficulties of image analysis is to find meaningful structures in an image. A very promising technique is to simulate a physical system which responds to “forces” generated by the image data. For instance, a string which is elastic and stiff, and which is attracted to local changes in image intensity, can be used to find connected smooth contours. Such a simulated string is let loose in the image, and allowed to attach itself to any structure it can find. The behaviour of these computational objects, known as active contours, is described by differential equations, and their evolution in time is simulated out using Euler's method.

3 More complex one-step methods

One-step methods are those which, like Euler's, try to make a prediction for time t+T using information only from time t or later, where T is the interval between times for which the function is estimated.

The limitation of Euler's method is that it assumes that the value of the derivative (the speed in the example above) is constant over each successive interval. More sophisticated methods attempt greater accuracy by weakening this assumption.

For example, in the *midpoint method*, the derivative (the speed in the robot example) is estimated at the centre of the interval rather than at its beginning, which clearly makes more sense. The problem with this that the speed might depend not only on elapsed time, but also on the position the robot has reached by the middle of the time interval. In a simulation, we do not have a direct measurement of the speed, so we need to know the robot's position to find the midpoint speed estimate. This seems circular. The way out is to use Euler's method to estimate the position the robot will have reached half-way into the time interval, then get a midpoint estimate of speed from that, and use that to get the position at the end of the interval.

This kind of thing can be elaborated further: a second mid-point estimate of the speed might be made, and combined with start-point and end-point estimates to get an overall speed for the final prediction. A class of methods based on this general idea, but carefully designed using an analysis of the errors that will occur, is the *Runge-Kutta* methods, one particular case of which forms a kind of standard in the area.

A different kind of refinement is to vary the step size T according to how rapidly the function seems to be fluctuating. The result obtained by taking two half-size steps can be compared with that from one full step to see whether significant accuracy is being lost on the full step, and the step size adjusted accordingly.

The main point to realise is that all these methods involve trade-offs amongst accuracy, computation time, and ease of implementation. There is no simple rule for choosing an algorithm, since this depends on the details of the problem — every methods exploits assumptions about the smoothness of the functions concerned in some way. The books mentioned above give considerable guidance, and give formal analyses of the errors: but the error expressions are in terms of higher-order derivatives, so the assumption of smoothness is still present. In the end the consistency of the results when parameters are varied across trials is probably the best check on whether a simulation is using sensible methods.

4 Multistep methods

The one-step methods discard information from before time t when they are calculating the prediction for time *t+T.* It seems reasonable to suppose that for smooth functions, this information should be retained and used. In general, some weighted sum of the previous estimates of the function and previous values of its derivative is going to be a good start. Methods which use this information are called multistep methods; a common example is the *predictor-corrector* method, which involves a weighted sum of the previous values and their derivatives to make a prediction, and then uses the derivative at this predicted point to improve the prediction further.

5 A generalisation

The weights for the weighted sums are calculated by considering the theoretical errors, and assuming that higher order derivatives of the functions are relatively small. However, the use of a weighted sum inevitably suggests that there might be something to be gained by making the weights adaptive, if it is the case that we find out what the true value of the function is at some point. For example, in financial forecasting, one might supply a set of recent values of a share price as inputs to a neural network, together with the values of other variables which might be expected to affect the price (and so might be related to its derivative with respect to time). The network's output could be taken as a prediction of the next day's price, and then when the true value came along, the network could be trained using any supervised learning technique. Such a network could learn to simulate at least the predictor step of the predictor-corrector method; since the network would be data adaptive, it should be capable of outperforming other methods for at least some classes of input. For a relatively simple signal, such as the level of the tide measured each hour, a simple weighted sum with adaptive weights can produce good predictions over surprisingly long periods.

Section 7 Some probability and statistics

The area of probability and statistics is an enormous one with vast applicability. This section summarises, in sketchy outline, a few of the more basic and useful topics in this field.

Contents

1 Introduction

Mathematically, a probability is a number between 0 and 1, which is manipulated according to certain rules, and these are well established and well understood. The interpretation of probabilities remains, however, a matter for debate. Very roughly, there are two schools of thought. For the frequentist school, the probability of an event is the fraction of times that the event will occur if the situation leading up to the event is replicated as exactly as possible. For the subjectivist school, the probability of an event is a measure of the strength of belief of a rational being that the event will occur. Whilst these differences are important both at a philosophical and practical level, they will be left to one side here.

In classical physics, probabilities reflect ignorance on the part of an observer (we express the fall of a coin in terms of a probability because we do not know enough about the exact spin, lift etc. imparted when it is tossed), whilst in quantum physics the uncertainty entailed in the use of probabilities is a fundamental property of nature (the future state of an atomic particle cannot, even in principle, be predicted exactly). In Artificial Intelligence, probabilities are generally used to deal with the ignorance of a reasoning system about the exact details of a situation. In simulation studies, probabilistic reason is used both to set up initial conditions and to interpret the outcomes of repeated trials.

Textbooks such as that by Boas (see Section 1) give an introduction to probability. The use of probabilities will be encountered in almost every branch of the study of evolutionary and adaptive systems.

2 Probability fundamentals

2.1 Basics

The probability of an event E is written P(E), or sometimes as Pr(E). For example, if H is the event that a tossed coin turns up heads, and the coin is fair, then we might write P(H) = 0.5.

A fundamental property of probabilities is that if two events A and B are *mutually exclusive* (i.e. they cannot both occur), then

$$P(A \text{ or } B) = P(A) + P(B)$$

(Notation: where I have used the word "or", books often use the symbol $\cup$, which also means set union, or the symbol $\vee$ as in predicate calculus.)

For example, if P(H) = 0.5 and P(T) = 0.5, and H and T cannot both occur (perhaps because they stand for heads or tails in a single coin toss), then P(H or T) = 1.

This last example illustrates another fundamental property: if a set of events is *complete* (i.e. one of the events in the set must occur) and all the events are mutually exclusive, then their probabilities add up to 1. It is occasionally helpful to think of probability as a limited resource which has to be distributed over the possible events in a way which reflects how likely each one is.

More formally, if events $X_1 \ldots X_N$ are complete and mutually exclusive (i.e. exactly one of them must occur), then

$$\sum_i P(X_i) = 1$$

In fact, what I have said so far is sufficient to establish the formal basis of probability theory. One consequence of these properties is that if A and B are two events that are not necessarily mutually exclusive, then

$$P(A \text{ or } B) = P(A) + P(B) - P(A \text{ and } B)$$

where "A and B" means that both events happen. (This is often written A & B, or using the symbol $\cap$ (set intersection) or $\wedge$. You may also see $P(AB)$ and $P(A,B)$, meaning the same thing.) The formula makes sense, in that if the two events are mutually exclusive, then $P(A \text{ and } B) = 0$, giving the previous formula.

2.2 Estimating probabilities

Textbooks (e.g. Boas — see Section 1) devote a large amount of space to the matter of estimating the probabilities of different kinds of event. Such questions as "What is the probability that you and a friend have different birthdays?" are the grist to this mill, which also involves a lot of throwing of dice and drawing of cards.

Questions like this generally involve summing probabilities for mutually exclusive events. A *sample space* of all possible mutually exclusive primitive events is set up; usually there is some symmetry argument that makes the probabilities for all these events equal. Then primitive events have to be combined to create the set of circumstances that allows the question to be answered; the probabilities for the primitive events are summed. Although this sounds simple, subtle and complex arguments are often needed to get it right.

As this is standard textbook material, I will not dwell on it. Here is a simple example, just to give the general idea. Suppose that in some genetic code strings of length 5 are generated by randomly selecting from the characters A, B, C and D with equal probability. What is the probability of finding the combination AAAA somewhere in a given string? First, the sample space is the space of all possible strings (AAAAA, AAAAB, AAAAC etc., to BDDDD, CDDDD, DDDDD). Each of these has the same probability, and there are 4^5 of them. The probability of each one is $1/4^5$, or approximately 0.001. 4 of the strings have AAAA at the left end, and 4 have AAAA at the right end, but AAAAA is common to both these groups, so the total number containing AAAA is 7. Thus the probability required is $7/4^5$ or about 0.007. This process of effectively counting the number of ways to get a given observation is the basis of combinatorial probability computations.

(The general case of finding the probability of a given pattern in a random string is covered by a variety of formulae depending on the exact circumstances — textbooks will give great detail when needed.)

Another way to do the example is simply to generate all the possible strings and count the number of them containing the sequence AAAA, dividing by the total number to get the exact probability. Another way is to use a *Monte Carlo* method, generating strings at random and estimating the probability by counting those that have AAAA. Such techniques are valuable fallbacks when textbook methods cannot be applied. Many simulation experiments are, in a sense, just Monte Carlo probability estimations.

2.3 Conditional probability and independence

It is often useful to talk about the probability of one event, given that another event is known to have occurred. For example, for an autonomous robot, it may be useful to consider the probability that there is a particular object — say an apple — in front of its camera, given that the robot's perceptual system is reporting the presence of a particular image feature — say a circular shape. Such a probability is a *conditional probability.* If A is the event that an apple is present, and C is the event that a circular shape has been detected, we write

$$P(A \mid C)$$

to mean the probability of there being an apple when a circular shape has been seen.

One way of thinking about this is to use a *possible worlds* model for probabilities. $P(A)$ means the fraction of all possible worlds in which there is an apple in front of the camera. $P(C)$ means the fraction of all possible worlds in which a circle is detected. $P(A \mid C)$ means the fraction of worlds in which there is an apple, but considering *only* those possible worlds in which a circle has been detected. Whether you like a "possible worlds" way of thinking about probabilities is partly a question of your philosophical stance.

Some books use the notation

$$P_C(A)$$

to mean the same as $P(A \mid C)$.

It seems reasonable that event C does tell the robot something about event *A*. Consider event *B*, which is the presence of an apple on a tree in the garden. Knowing *C* does not tell the robot anything about the likelihood of *B*. It seems reasonable to express this by writing

$$P(B \mid C) = P(B)$$

This formula is actually one definition of statistical *independence*. If the probabilities for *B* and *C* obey this rule, then *B* and *C* are statistically independent. It is often useful to assume statistical independence between variables even when this is not strictly justified.

One warning — unnecessary I hope. If two events really are physically independent, then they will be statistically independent and observing one of them will not affect the probability of the other. Failing to realise this is the classical gambler's error. Observing six heads in six tosses of a coin does not mean that the probability of a head on the next toss is anything other than a half, provided the coin is fair. It is easy for subtle versions of this error to crop up, and it is advisable to watch out for them. (The chairman of the National Lottery was quoted in a newspaper as saying that high numbers had a good chance in a particular week because there had been a preponderance of low numbers before that — one might have hoped that someone in his position would have known better.)

There is a second way to express statistical independence numerically. If *B* and *C* are independent, then the probability of both events occurring is given by

$$P(B \text{ and } C) = P(B)\,P(C)$$

This can be shown to be equivalent to the definition above in terms of conditional probability. It is an important relationship, in that it is often useful to assume that different events are independent, because we know of no causal link between them, and to work out probabilities of combinations of events using this product rule.

2.4 Bayes' theorem

There is a very important relationship concerning conditional probabilities. We will approach this by putting some numbers into the perception example above.

Suppose that the robot "knows" that apples happen to be in front of its camera on one-tenth of the occasions that it looks, so

$$P(A) = 0.1$$

and that it detects a circular shape in the image on one-fifth of the occasions that it looks, so

$$P(C) = 0.2$$

In addition, the robot knows from previous experience (when it has been shown something and told that it definitely is an apple), that it detects a circular shape three-quarters of the time when an apple is in front of it. That is,

$$P(C \mid A) = 0.75$$

Now the robot, roaming its world in search of food, detects a circular shape. What is the chance that an apple is in front of it? In other words, given the data above, what is $P(A \mid C)$?

This can be answered by considering how often there is an apple in front of the camera, *and* the robot sees a circular shape; that is $P(A \text{ and } C)$. It is easy to work this out: an apple is in front of it one tenth of the time, and on three-quarters of those occasions a circle is seen, so the answer is three-quarters of a tenth or 0.075. In symbols

$$P(A \text{ and } C) = P(A)\,P(C|A) = 0.1 \times 0.75 = 0.075$$

Now we know that the robot has seen a circle. The probability that there is *also* an apple present, given the circle, is the fraction of times an apple and a circle occur together, divided by the fraction of times a circle occurs without regard to the apple. In symbols

$$P(A|C) = P(A \text{ and } C) \,/\, P(C) = 0.075 \,/\, 0.2 = 0.375$$

So the answer is 0.375 or three-eighths. One way of saying this is that the *evidence* of the visible circle has increased the probability of the *hypothesis* that an apple is in front of the camera from 0.1 to 0.375.

If this is unfamiliar, work through the argument again using different numbers and a different example. For instance, imagine a patient going to a doctor. *M* might be the event that a patient has meningococcal meningitis (with say $P(M) = 0.00002$, or one in 50,000), *S* might be the event that the patient has headache with fever (with say $P(S) = 0.004$, or one in 250), and it is known that patients with this kind of meningitis display headache with fever half the time (so $P(S \mid M) = 0.5$). If a patient turns up with this pair of symptoms, what is the chance that he or she has meningococcal meningitis — what is $P(M \mid S)$? (On these data, it is 0.00025 or one in 400 — but check that you get the same answer. These numbers are made up and may not correspond to reality.)

Given the numerical examples, it is possible to see how they generalise and so to write down the formula for the general case. This is *Bayes' Theorem*:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Terminology: $P(A)$ is known as the *prior probability* of event A — prior, that is, to knowing that B has occurred. $P(A \mid B)$ is the *posterior probability* of A.

The theorem itself is uncontentious: it follows directly from the fundamental properties of probabilities. Extremely contentious, though, are its interpretation and its application in practice. Very often, the use of the theorem is associated with the philosophical stance which interprets probabilities as measures of strength of belief; hence “Bayesian inference” is a phrase with connotations well beyond the mere use of the formula.

The greatest practical problem with Bayesian inference is the estimation of the prior probability $P(A)$. Usually, the conditional probability $P(B \mid A)$ is estimated from some kind of experiment (or, equivalently, as part of a learning process in an a-life system), and $P(B)$ is just a normalising factor to make all the probabilities for the alternatives outcomes to A add up to 1. The prior is much harder to estimate — if there is nothing else to go on, all the possibilities are given equal prior probalities, but this can be hard to justify, and causes technical problems when the outcome A, rather than being a definite event, is one of a continuum of possibilities, as when an estimate is being made of a real-valued number.

Nonetheless, Bayes’ Theorem is well worth remembering, partly for the insight it gives into the meaning of conditional probabilities. It is of increasing importance in AI — for example, expert systems that used ad hoc measures such as confidence factors are giving way to those based on Bayesian methods. The approach has received impetus from the work of Judea Pearl, who has described algorithms for efficiently propagating probabilities through a “belief network” using Bayesian rules.

2.5 Summing conditional probabilities

There is a second formula that is useful involving conditional probabilities. It is an extension of the basic summation relationship for probabilities of mutually exclusive events. If $X_1 \ldots X_N$ is a complete set of mutually exclusive events, and A is some event that depends on them, then

$$P(A) = \sum_i P(A|X_i)P(X_i)$$

This follows because the expression being summed is equal to $P(A \text{ and } X_i)$. This is one of a set of mutually exclusive events, and if any one of them occurs, then A occurs, so the probability of all of them added together is the probability of A.

This is yet another weighted sum, and so gives another interpretation to the action of a linear neural network unit (Section 2). The weights on such a unit can sometimes be sensibly regarded as representing the conditional probabilities of event A given different input events X_i. The input data represent the probabilities of the different events, and the output is the probability of A. Such an interpretation can only be applied straightforwardly, of course, if all the signals are in the range 0-1 and both weights and inputs are normalised to sum to 1.

2.6 Random variables and probability distributions

Like an ordinary variable, a random variable can take any one of a set of values. For a random variable, the values represent a complete set of mutually exclusive events, and there is a probability associated with each event. The *probability distribution* for a random variable is just the association of a probability with each of its possible values.

For a finite set of values, this is straightforward. If X is a random variable whose values are h (for heads, perhaps) and t (for tails), then a typical probability distribution might be $P(X = \mathrm{h}) = 0.5$ and $P(X = \mathrm{t}) = 0.5$. This is an example of a discrete distribution.

If the random variable X can take on any of a continuous set of values — for example, if X represents any real number in the range 0 to 1 — then we have a *continuous* distribution and the assignment of probabilities needs some extra formalism. One way to handle this is to describe the distribution by using the probability that X is less than some particular value:

$$P(X < x)$$

where X is the random variable and x is some specific value that it might or might not exceed. This probability is a function of x — we might write it as $F(x)$ where F is the name of the function — and is called the cumulative probability distribution.

For example, if X is equally likely to have any value between 0 and 1, then its cumulative probability is given by $F(x) = x$. This distribution is called a *uniform distribution*.

The cumulative probability does not seem to tell us the probability of X having any particular value. In fact, there is no answer to that question for a continuous variable; the nearest approach is to consider the probability that X lies within a range of values, from say x to $x + \delta x$, or

$$P(X \geq x \text{ and } X < x + \delta x)$$

We then consider what happens when δx is made smaller and smaller. Assuming that the cumulative probability varies sufficiently smoothly with x, we would expect that when δx is sufficiently small, the probability for a range of values will be proportional to the size of the range, and we expect that, approximately,

$$P(X \geq x \text{ and } X < x + \delta x) = f(x)\delta x$$

where *f*(*x*) is some function of *x*. We expect that the approximation will be exact when δ*x* is reduced to be infinitesimally close to zero. The function *f*(*x*) is called a *probability density function*, and bears the same relation to probability that the density of a substance does to mass. You need to multiply a density by a volume to get a mass, and you need to multiply a probability density by the size of part of sample space to get a probability.

There is a relationship between the cumulative distribution and the probability density; it is

$$f(x) = \frac{d}{dx}F(x)$$

(and the cumulative distribution is the integral of the density).

Sometimes the notation p(*X*) is used to indicate the probability density function of a random variable *X*.

There is, again, a large literature on specific distribution functions. By far the most common (apart possibly from the uniform distribution) is the *Gaussian* or *normal* distribution, for which the probability density function is

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}}\exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)$$

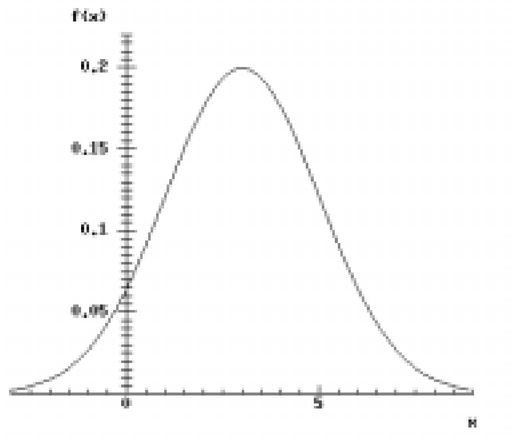
where μ and σ are called the *mean* and *standard deviation* respectively of the distribution.

The graph of this function is sometimes called the bell shaped curve. It is particularly beloved by psychologists and social scientists; so much so that a recent highly controversial book on IQ took “The Bell-Shaped Curve” as its title.

There are two reasons why this distribution is so common: one is a theoretical result which says that if a random variable is the sum of a large number of other random variables with their own arbitrary distributions, then it will tend to have an approximately Gaussian distribution; the other is that Gaussian-distributed random variables have some properties that make them easy to manipulate.

Any standard textbook will discuss this distribution, along with various other important distributions. Statistics books nearly always have a table of its values, and also a table of the corresponding cumulative distribution *F*(*x*). This is is called the *error function* (abbreviation erf) and is important for some statistical tests. Some mathematical software packages include a procedure for computing erf.

The bell shaped curve for the particular case of mean = 3 and standard deviation = 2 appears below as an example.



3 Statistics of distributions

It is often necessary to summarise the distribution of a random variable using a few numbers. This might be because the exact details of a distribution are not known, or because only some of its properties are relevant to a situation. The quantities that summarise a random distribution are called its *parameters*, particularly when they appear explicitly in the formula for the distribution. Often, however, a distribution is not fully specified theoretically, but some aspects of it must be estimated from some data. Quantities that help describe a distribution and which have been calculated from data are called statistics. The process of estimating parameters from statistics is called *statistical inference*.

Here we look briefly at a few of the more important statistics. Statistical inference in general is too large a topic to embark on here, but the use of averages to describe properties of data is something that everyone should be familiar with.

3.1 Averages

One of the most important descriptors of a distribution is its *mean*. This term is used in two ways. In the formula for the Gaussian distribution, the mean was one of the parameters (and was denoted by μ). On the other hand, the mean is also a statistic, obtained by adding together a set of observed numbers and dividing by the number of observations. The mean is also called the arithmetic average, or just average. In the case of a Gaussian distribution, this statistic is a good estimator of the parameter μ.

It is useful to define the mean for any distribution, whether or not it appears as an explicit parameter.

For a discrete distribution with a finite number of values of the variable, the mean is

$$\sum_i x_i \mathrm{P}(X = x_i)$$

where X is the variable, and its possible values are x_1, x_2 etc. This is often abbreviated to

$$\sum_i x_i \mathrm{P}(x_i)$$

where x_i is a shorthand for the event $X = x_i$.

This makes sense in terms of the mean as a statistic. If we make a large number of observations, say N, we expect each x_i to occur about $N\,\mathrm{P}(x_i)$ times (from the meaning of probability). Thus if we average the observation values, adding them and dividing by N, we expect to get approximately the result

$$\frac{\sum_i x_i N \mathrm{P}(x_i)}{N}$$

which is just the distribution mean. For this reason, the mean of a distribution is sometimes called its *expectation* (especially in quantum mechanics).

The mean of the distribution of a variable X is often written

$<X>$ or $\overline{X}$

Angle brackets will be used here.

The idea of an average can be extended to any numerical function of a random variable. For a function f, the general formula is

$$\langle f(x) \rangle = \sum_i f(x_i) \mathrm{P}(x_i)$$

In a cellular automaton, for example, the different states might be labelled with a set of symbols. An energy might be defined for each state; the average energy (which is important in some analyses) would then be obtained using a formula such as that above, if the probability of each state was known or could be estimated.

For continuous variables, the sums in the formulae above become integrals.

3.2 Variances and standard deviations

The mean of a distribution says, loosely, something about where its centre is. The next most useful thing to know is how spread out the distribution is. One way to measure this is to work out the average distance of the values from their mean. In practice, squaring the differences between the values and the mean not only avoids negative numbers, but makes various calculations simpler. The average squared distance from the mean is called the *variance*, and its formula is given by

$$\mathrm{Var}(x) = \sum_i (x_i - \langle X \rangle)^2 \mathrm{P}(x_i)$$

It is not difficult to show that this is equivalent to

$$\langle X^2 \rangle - \langle X \rangle^2$$

which gives a quick way to estimate the variance from data.

The square root of the variance is called the *standard deviation*. It turns out that if the sum above is replaced by an integral and applied to the Gaussian distribution, the standard deviation as defined here is just the parameter sigma, justifying the use of the term in the description above.

3.3 Entropy

A final average that is often of great interest is minus the average of the logarithm of the probabilities of a distribution. This is at first sight a strange thing to calculate; the formula is

$$\mathrm{Entropy}(x) = -\sum (\log \mathrm{P}(x_i)) \mathrm{P}(x_i)$$

This does not depend on the values of X at all — only on how the probability is spread out across them. This quantity has a variety of interpretations and uses, but essentially it measures the smoothness of the distribution.

For example, suppose there are N different values for X with equal probability $1/N$. Then the entropy is just -log(1/N) which is equal to log(N). For a flat distribution like this, the entropy thus increases with the number of possibilities. On the other hand, suppose the probabilities are all zero, except for a single value of X which always occurs — a maximally peaked distribution. For this, the entropy is zero (since log(1) = 0).

Entropy has a central role in *information theory.* If information about a variable’s value is to be transmitted, then on average the amount of information needed to specify the value is given by the entropy of the distribution. If the logarithm is to base 2, entropy is measured in *bits*. If there are two equally likely possibilities for a variable, then both the formula and common sense indicate that 1 bit of information is needed to specify the variable’s value.

Consider, for example, a linear neural network unit whose two inputs are random binary variables, which each independently take values -1 or +1 with equal probability. The variable *X* stands for the input vector, so its values are (-1,-1), (-1,+1), (+1,-1), (+1,+1), each with probability 1/4. The entropy of this distribution, using the formula above, is

$$\begin{aligned} & -\left(\frac{1}{4}\log\frac{1}{4} + \frac{1}{4}\log\frac{1}{4} + \frac{1}{4}\log\frac{1}{4} + \frac{1}{4}\log\frac{1}{4}\right) \\ = \; & -4 \times \frac{1}{4} \times \log\frac{1}{4} \\ = \; & 2 \end{aligned}$$

because log(1/4) = -2 if we use logarithms to base 2. That is, *X* requires, not surprisingly, 2 bits of information to specify it.

Now suppose this unit simply sums its inputs. The output, *Y*, is a number whose values are -2, 0, 0, +2 respectively for the 4 possible inputs. There are only 3 values of *Y*, with probabilities P(-2) = 1/4, P(0) = 1/2, P(+2) = 1/4 (using the rule for combining probabilities for mutually exclusive events). Thus the entropy of *Y* is

$$\begin{aligned} & -\left(\frac{1}{4}\log\frac{1}{4} + \frac{1}{2}\log\frac{1}{2} + \frac{1}{4}\log\frac{1}{4}\right) \\ = \; & 1.5 \end{aligned}$$

That is the output *Y* has a lower entropy — it transmits less information — than the input *X*. The reason is, of course, that information has been lost in the addition, as the two cases when the inputs are different cannot be distinguished in the output.

The idea of 1.5 bits might seem strange. Remember, though, that this is a measure of the amount of data needed to specify *Y on average*, if the process is done a lot of times. The entropy measures how well a perfectly efficient coding scheme could do in compressing the information carried by the output of the network when it is used repeatedly.

Entropy can also be viewed as a measure of *disorder.* The more disordered a system is, the more information is needed to specify it exactly. If the cells of a cellular automaton are all likely to be in any state with equal probability, then to transmit the state of the whole system we will need to transmit the state of each cell. The disorder is high, and calculating the entropy will give a large number. If, however, one state is much more likely than the others, then we need only transmit the state of the cells that are in one of the lower-probability states. The order is higher, and the entropy will work out as a lower number. Minus the entropy is sometimes called the *negentropy* and used as a measure of order in a system.

4 “Random” numbers

It seems worthwhile to include a brief note about random numbers in the context of computer simulations. Such simulations rely very heavily on so-called random numbers, usually generated by calls to library procedures such as C’s `rand` routine. In fact, such numbers are actually *pseudo-random*, since they are generated using a deterministic algorithm that produces a completely predictable sequence. True random numbers which are independent of one another form a completely unpredictable sequence.

Pseudo-random number algorithms are designed to be (a) fast and (b) produce numbers whose statistics are the same as far as possible as those of true random numbers. Thus the mean, standard deviation and so on of the numbers themselves, and higher-order statistics such as the mean of differences between successive pairs of numbers, will be within the expected ranges for uniformly distributed true random numbers.

Usually, there is no need to worry unduly about using these generators: their properties have been well worked out and they are adequate for most simulations. However, there have been one or two cases in which standard libraries have contained very poor generators, in that, for example, successive “random” values have been correlated with one another. It is worth bearing in mind this possibility: if a simulation is not behaving as expected, and all else fails, it might be worth trying using random numbers from a different routine.

In addition, for large, delicate simulations, there is a rule of thumb that suggests that the number of values extracted from a pseudo-random generator should not exceed the square root of the cycle length (or period) of the generator (the number of values it produces before repeating itself). A reputable generator will state its cycle length in its documentation — e.g. the implementation of `rand` on my machine has a period of 2^{32}, so should ideally not be used for more than 2^{16} = 65,536 values in any one simulation. Fortunately much better generators are easily available.

Do not be tempted to try to improve the properties of a pseudo-random generator by resetting it from time using, say the system clock, memory usage or some other “random” value from outside the program. Such strategies will almost always degrade the statistical properties of the generator.

One question that often arises in practice is how to get pseudo-random numbers that have approximately a given distribution. Most pseudo-random generators provide numbers drawn from a uniform distribution with cumulative probability $F(x) = x$, with $0 \le x < 1$. Suppose we need to simulate a random variable *Y* with cumulative probability distribution $G(y)$. If X and Y are related by $X = G(Y)$, then it is possible to show that Y has the required distribution. Thus to convert directly from values produced by the generator to the required values, it is necessary to compute the inverse of the required cumulative distribution function for each value generated. This can be difficult and computationally costly in some cases.

There is a very rough-and-ready short cut for *approximate* Gaussian distributions: adding 12 uniformly distributed random numbers in the range 0-1 together gives a value which is roughly from a Gaussian distribution with $\mu = 6$ and $\sigma = 1$. You can scale and shift this to approximate any Gaussian distribution, provided an accurate distribution is not needed. Using more than 12 inputs gives a better approximation; the mean will be half the number of values added and the standard deviation the square root of the number of values divided by the square root of 12.

5 Conclusion

Dealing with uncertainty is the central problem in prediction and control. Probability is the calculus of uncertainty, and statistics are the tools used to draw inferences from data within a probabilistic framework. The topics mentioned above — probability distributions and simple statistics — are the starting point for a great deal of sophisticated analysis. In particular, the theory of statistical inference is a large and complex one.

However, for many practical problems, a clear idea of what is meant by a probability distribution and by the mean and standard deviation (and possibly the entropy) can be put to good effect in straightforward ways.

Section 8 Statistical analysis of experiments

This section gives an introduction to some techniques relevant to the analysis of the results of experiments on non-deterministic systems.

Contents

1 Introduction

In traditional AI, it has been common for researchers to make their points by building systems that illustrate particular techniques or demonstrate particular competences. In some ways, this is rather like the approach of an engineer, in that the production of an object that performs a given task within given resources is sufficient to show an advance in the state of his or her art.

Increasingly, though, there is a kind of investigation that demands a different approach, more like that of a behavioural scientist. This occurs particularly when systems cease to be transparent; it is not enough to build such a system: its properties must also be explored. In addition, in artificial life and evolutionary systems simulations, such systems involve the use of random numbers to mimic environmental variability, whilst robotic systems that interact with the real world are subject to the genuine thing. Characterising systems which involve variability involves the use of statistical methods.

The use of statistics applies the theory of probability (see Section 7) to the description of processes which are subject to random variation. Various kinds of descriptive statistics are useful in general exploration of a system, and are the main method of trying to obtain some degree of understanding of it. More formal methods of *statistical inference* are used to draw quantitative conclusions or to attempt to determine specific properties of a model of the process.

Here I mention a few techniques of descriptive statistics, and discuss one particular approach to statistical inference, known as hypothesis testing.

Books aimed at psychologists are probably the most useful for an initial understanding of this material, and for practical help in applying it. Two that are widely used are “Learning to use statistical tests in psychology”, bu J. Greene and M. D’Oliveira (Open U.P., 1982, in the library at QZ 210 Gre), and “Non-parametric statistics for the behavioural sciences”, 2nd edition, by S. Siegel and N.J. Castellan (McGraw Hill, 1988, in the library at QD 8320 Sie).

2 Descriptive statistics

2.1 Graphs

When looking at results from an experiment, the first set of tools to turn to are graphical ones. Tools such as Matlab, Maple and AVS provide a wide variety of ways to display data graphically. The area is too large and complex to discuss here, and methods such as 2-D and 3-D graphs and bar charts are probably familiar already. The main point is that time spent producing graphical output is usually well spent, but that when data have multiple dimensions, it can be difficult to find the appropriate combinations to display. It is essential to spend time finding the right way to display data in order to reveal relationships which may be present.

2.2 Simple numerical statistics and correlation

The mean and standard deviation of a set of data have been discussed in Section 7. Calculating these statistics for results obtained when an experiment is repeated is often the first step in gaining a clear view of what is going on. The mean gives a measure of the location of the centre of some numerical data; the standard deviation gives a meaures of its spread.

An additional descriptive statistic, not introduced in Section 7, is the *correlation coefficient* between two sets of data, which can be used when the individual data values can be paired off between the two sets. This gives a measure of whether the two random variables being sampled vary together or are independent. For instance, in an experiment involving a simulated visual system, it may be interesting to look at whether the time to pick out some target varies with the number of distracting objects in the field of view.

If two random variables X and Y are being sampled, the correlation coefficient is defined as

$$r = \frac{\langle (X - \langle X\rangle)(Y - \langle Y\rangle)\rangle}{\sqrt{\mathrm{Var}(X)\mathrm{Var}(Y)}}$$

That is, it is the average of the products of the deviations of the variables from their means, normalised using the variances. It lies between -1 and +1, and either -1 or +1 means that there is a perfect linear relationship between the two variables, whilst 0 corresponds to no *linear* relationship betwen the two.

2.3 The histogram

The mean, standard deviation and other similar measures provide some indication of the *distribution* of a variable (such as the fitness of a population) which is being measured. A graphical way of looking at the distribution generally is to use the *histogram* of the values found.

The simplest way to produce a histogram is to create, in effect, a set of bins covering the range of values of the variable. Each bin initially contains the value zero. After each trial, the value of the variable being measured is used to pick out a bin, and the value held in the bin is incremented. For instance, in a simple case, a measure might range from 0 to 99. We create 10 bins, covering the ranges 0-9, 10-19, 20-29 and so on. If a trial yields the value 63, we increment the 60-69 bin, and so on. After a large enough number of trials, the values in the bins will be an approximation to the underlying probability distribution of the variable.

There is a trade-off between the number of bins and the accuracy of the probability estimate each one holds. A lot of bins gives a narrow range of values for each, giving a higher accuracy on the position of any feature of the distribution, but lower accuracy on the probability estimates because fewer votes will be cast for each bin.

There are more sophisticated ways of generating histograms which do not involve discrete bins. (Treating the data as a set of delta functions and convolving this with a smoothing kernel is one such method.) All of them, however, involve essentially the same trade-off.

Looking at graphs, descriptive numerical statistics, and histograms are all important ways of understanding a system. More formal methods are also sometimes called for, particularly in the context of statistical variability.

3 Hypothesis testing

3.1 Basic framework

Suppose you run a simulation and measure some outcome — say the average level of fitness in a population after a certain number of generations, or the number of times a robot succeeds in reaching its goal. You then make some adjustment, perhaps by varying a parameter of the simulation such as the mutation rate of a genetic algorithm or the rate of learning of a neural network, and repeat the simulation. If the outcome changes, how can you say whether this was a result of the adjustment you made, or simply a random fluctuation which might have been expected to occur regardless?

This kind of question is at the heart of the dominant statistical methodology of the behavioural, social and medical sciences. The question of whether a new drug has an effect on the outcome of a particular disease is, for example, a crucial one in medicine.

The method generally used is called *hypothesis testing*. The approach is to ask whether it is reasonable to attribute any differences observed to random fluctuations, assuming that the manipulation (application of the drug, change of the mutation rate, or whatever) has no effect. If the changes are too big for this to be reasonable, then the experiment is taken as evidence for a real effect. The reason for doing it this way round is that if there is no effect, then it is possible to calculate the probabilities associated with the measurements, and see how unlikely they are.

Some terminology is needed to set this up formally.

The *null hypothesis*, denoted by H_0 is the hypothesis that the differences in conditions between the two runs of an experiment have no effect. The *alternative hypothesis*, H_1, is that H_0 is false, i.e. there is an effect of the manipulation. If we decide that the experiment shows an effect when in fact there is none, we have made a *Type I error.* Conversely, if we decide that there is no evidence for an effect, when in fact one exists, we have made a *Type II error.*

Usually, the differences between the experimental results are summarised in a single statistic. This might be something like the change in the success rate of the robot. We then calculate the probability, *assuming the null hypothesis*, of getting either the observed value of the statistic, or *a more extreme value*. This probability is always given the symbol *P*, and is known as a *significance level*. If *P* is low, then the result we have is unlikely under the null hypothesis.

3.2 A simple example

Suppose you conduct an experiment in which you run a simulation of a system, setting your pseudo-random number generator to a particular seed before you start, and using value *a* for some parameter you are interested in. You then change the parameter to *b*, reset the random number generator to the same seed as before, and rerun the experiment. You then look to see whether the performance is better or worse then it was before. You then repeat the pair of tests some number of times — say 10 — recording for each pair whether performance increased or decreased when the parameter was changed from *a* to *b*. Different random numbers are used in each pair of tests.

Suppose the performance gets better on 8 trials out of 10, and worse on 2 trials. How likely is this under the null hypothesis that changing the parameter from *a* to *b* produces no improvement in the peformance? Is the overall improvement attributable to the change in the parameter?

The null hypothesis says that changing the parameter has no effect, so the performance is equally likely to get better or worse; each trial is like tossing a coin. In this case, there are $2^{10} = 1024$ different equally likely ways the experiment can turn out (see Section 7). In one of these, performance will improve on all 10 trials, in 10 of them performance will improve on 9 trials, and in 45 performance will improve in 8 trials (you can check this by enumerating the different cases, or by using the binomial expression if you happen to know it). In other words, there are 1 + 10 + 45 = 56 cases that give the observed result or a better one in the sense of more improvements. If better is interpreted to be "more extreme", then it follows that $P = 56/1024$ or about 0.055. That is, on about 55 in 1000 repetitions of the whole sequence, you would expect to get 8 or more improvements, just by random fluctuations in the total.

You may ask whether a result of 2 or fewer improvements out of 10 would not be just as "extreme" as a result of 8 improvements. This depends on whether you simply want to test that the change had an effect of some sort, or whether you want to test that it produced an improvement. If the former, then these other cases would also have to count as "extreme", and the *P* value would double to 0.11. This is called a *two-tailed test*. If however, the alternative hypothesis is that the change does produces specifically an *improvement*, then exceptionally poor results are lumped in with the run-of-the-mill ones, only success rates greater than that observed are counted as more extreme, and the test is called *one-tailed*.

This kind of experimental design, incidentally, is called a *related samples* design. Within each pair of tests, everything is the same except for the parameter of interest. Thus every single run of the simulation with parameter *a* has its own *control* with parameter *b*. The tests come in *matched pairs*. An alternative way to do it would be to use new random numbers for every single trial. This is called an *independent samples* design; there is no natural pairing. One could still apply the test described above to arbitrary pairs, but an effect would be much more likely to be masked by random fluctuations in the results (that is, the test would not be very powerful). In an independent samples design, you would be more likely to adopt a statistical test in which you put the results into different classes before doing any comparisons between the two conditions.

3.3 General methodology

Does the result $P = 0.055$, obtained in the imaginary experiment above, mean that changing the parameter produces an improvement, or not?

The received version of how to answer this is as follows. Before doing the experiment, you decide on a critical value of *P*. This is called α (alpha). If *P* turns out to be less than α, you reject the null hypothesis (you accept the existence of an effect). Otherwise, you accept the null hypothesis.

How do you choose α? The α value is in fact the probability that you will make a Type I error — that you will think you have seen an effect when there isn't one. That is, if you decide to use $\alpha = 0.05$, and you do lots of independent experiments, then if all the null hypotheses are true, on one experiment in 20 you will get $P < \alpha$ and you will decide that there is an effect that is not there. This follows directly from the definition of *P*. So if you do not mind making this kind of error one time in 20, you choose $\alpha = 0.05$; if you want a stricter criterion, you might choose $\alpha = 0.01$.

This approach means that one never has to calculate the exact value of *P* for a given experiment. What you do is to look up the value of the statistic you are using that would give $P = \alpha$. Then if, when you do the experiment, the statistic is more extreme than this critical value, you reject H_0; otherwise you accept

H_0. More extreme values are said to lie in the *critical region* or *rejection region* for the null hypothesis; the rest lie in the acceptance region. For the experiment described above the statistic is the number of times an improvement occurred. In the one-tailed test, values of 9 and 10 lie in the rejection region for $\alpha = 0.05$, but 8 does not quite make it. Eight out of 10 improvements would not allow us to reject the null hypothesis at the 0.05 significance level.

You can picture these regions by drawing a graph of the distribution of a statistic. Suppose for this purpose that the statistic has continuous values. For commonly used statistics, the distribution will have a hump in the middle for the likely values and tail off for extreme values. For a one-tailed test, split the area under the curve into two parts: a part under one tail occupying 5% of the area and a part under the rest occupying 95% of the area. The 5% region represents the rejection region for $\alpha = 0.05$. For a two-tailed test, the 5% has to be split across the two tails of the distribution.

Knowing the probability of a Type I error is useful, but of course the probability of a Type II error (not seeing an effect that is really there) is useful too. However, estimating this probability is generally quite messy and difficult. Intuitively, the more data you have, the lower the chance of a Type II error ought to be (for a given α), and indeed this is the case for any reasonable test. Different tests are compared on their power, which is 1 - P(Type II error), but working this out often involves making more detailed assumptions about the distribution of the data than does the null hypothesis. The reason for this is that to say something about Type II errors means that you have to say something about the distribution of the data when the null hypothesis is false — and that might be much harder to specify precisely.

Essentially, a significance test gives a good measure of the probability that an observed effect has occurred by chance. A low value for P can thus be a reliable indicator that something real is going on. On the other hand, if the null hypothesis is accepted because P is large, there is nothing simple to say about what the chance of a Type II error is. There might be a real effect which is not shown up, either because there is not enough data, or because the statistic used is not a good one for detecting the particular kind of difference that has occurred.

3.4 Another example

There are significance tests to cover many different situations. The books mentioned at the head of this file describe many different tests and give guidance on making an appropriate choice. In the event of your needing a test for an experiment, you will need to spend time analysing the nature of the measurements and the experimental design to find the correct one.

The test used above on the experiment with binary outcomes (improvement or non-improvement) is called the *sign test*. Here, I give one further example of a test to illustrate the general idea. This test is of quite wide applicability; it is used when you want to know if two independent (not matched) sets of data obtained under different conditions differ significantly. For each condition some outcome is observed in a number of trials; the outcome must be measured with a number (strictly speaking, it must be an ordinal measure). We want to know whether the outcome is significantly different in the two conditions. Since we have a number of trials in each condition, we have an indication of what the spread of likely values of the outcome measure is, so it seem reasonable to suppose that information about whether the conditions differ significantly is available without making further assumptions.

One test that will handle this situation is the *Kolmogorov-Smirnov* two sample test. The statistic that this uses is the maximum difference in the cumulative distributions of the two outcome measures. This is easiest to explain with an example.

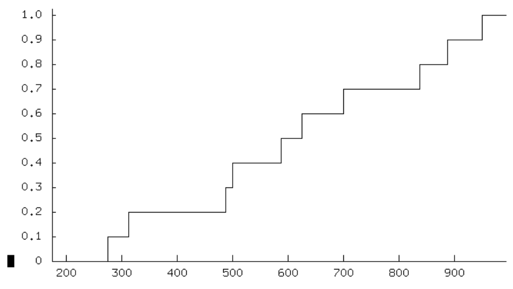
Suppose we conduct a series of trials — say 10 — in one condition — say using one kind of crossover operator in a genetic algorithm. In each trial we measure, say, the number of generations to reach a particular state of the population, and get the following results:

630 890 700 270 500 480 320 950 836 585

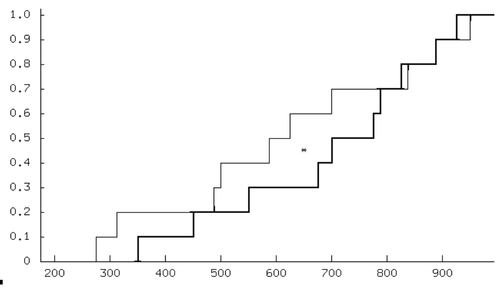
We then do the same thing in an independent set of trials (no pairing with the first set) using a different operator. Suppose this gives

784 456 893 555 678 699 350 821 921 772

Is there a significant difference between these two sets of numbers? To find the statistic for the K-S test, imagine making a cumulative frequency graph for the first data set by counting the number of values that are less than any given value. It looks something like this:



What this graph means is that, for example, 0.2 of the values are less than 400 (in fact the values 270 and 320), 0.4 of the values are less than 550, 0.7 of the values are less than 750, and so on. Now we superimpose the graph for the other dataset. That gives something like:



The statistic needed for the K-S test is the largest vertical difference between the two graphs, which we will call *K*. You can see by inspecting them that the largest such difference is *K* = 0.3, near the asterisk, between values of the outcome from 630 to 678. In practice, one would calculate this statistic by ordering the two data sets independently, then comparing the ordering between them thus:

270	
320	
	350
	456
480	
500	
	555
585	
630	
	678
	699
700	
	772
	784
	821
836	
890	
	893
	921
950	

and finding the point in the sequence with the biggest difference in contributions from the two datasets above it. It is straightforward to write a program to do this, but many packages will do it for you. If there are different numbers of trials in the two conditions, division by the number of trials has to be carried out in counting the fraction of trials to the left of any point in the sequence.

The statistic *K* = 0.3 can then be looked up in tables for the test. Not surprisingly, this turns out to be not significant for 10 trials in each dataset even at α = 0.05 — these data do not seem to show any non-random differences using this test. In fact, a difference of *K* = 0.6 would be needed (for 10 trials in each condition) before the results were significant at the 0.05 level.

The clever thing about this is that the distribution of the statistic *K* under the null hypothesis (both sets of data come from the same underlying distribution) is known independently of what the distribution of the data actually is. There is no assumption that the data come from a Gaussian distribution, or indeed any other distribution. A test with this property is known as a *non-parametric* test.

Tests that make assumptions about the distributions are known as *parametric* tests; typically they assume the distribution is Gaussian. A parametric test that could be applied to these data, if you were willing to make the necessary assumption, is the unrelated t-test, which uses as its statistic the difference in the means of the two data sets, normalised by an estimate of the standard deviation of the data. In general, parametric tests are more powerful and involve simpler calculations; but if the assumption of a Gaussian distribution of the data is incorrect, they can give misleading results.

There are numerous other significance tests that can be useful. One important one is the chi-square test, which is useful when some data need to be compared with expected frequencies.

3.5 Combining significance levels

Sometimes a hypothesis is tested in two experiments which yield independent *P* values. The best way to combine the results is to find a way of treating the two experiments as one, and finding an overall statistic that can be used in a test of significance. When this is not possible, it can be useful to know how to combine more than one significance level in a sensible way.

In particular, the correct way to combine them is not to take their product, or their maximum or minimum (though all of these are sometimes suggested). If the two significance levels are P_1 and P_2, the significance level of the two experiments taken together is in fact

$$P = P_1 P_2 (1 - \log(P_1 P_2))$$

where the logarithm is to base *e* (a natural logarithm). Thus two experiments each significant at 0.05 yield a combined significance level of 0.017. The argument to reach this conclusion depends on a particular definition of “more extreme”, to mean combinations of results that would have lower probability under the null hypothesis than the results actually obtained.

The generalisation of this formula to N experiments is

$$P = g \sum_{r=0}^{N-1} \frac{(-\log g)^r}{r!}$$

where g is the the product of the N separate significance levels, and $r!$ is the factorial function of r.

3.6 Problems with hypothesis testing

Hypothesis testing is a respectable and sometimes valuable way to assess the results of experiments. However, it has difficulties.

An important one of these is that if the methodology were taken literally, hypotheses about, say, the effectiveness of a new drug would be accepted or rejected when it was known that there was a definite probability that an error was being made. This problem is exacerbated by the asymmetry in the treatment of the null and alternative hypotheses, which means that probabilities of Type I errors are accurately controlled but probabilities of Type II errors have to be largely guessed.

In practice, the approach is not followed literally: common sense prevails. Rather than setting an α in advance and then acting accordingly, most researchers tend to treat the P value obtained for their data as a kind of standardised descriptive statistic. They report these P values, then let others draw their own conclusions; such conclusions will often be that further experiments are needed. The problem then is that there is no standard approach to arriving at a final conclusion: everything remains tentative. Perhaps this is how it should be; but it means that statistical tests are used as a component in a slightly ill-defined mechanism for accumulating evidence, rather than in the tidy cut-and-dried way that their inventors were trying to establish.

The rejection/acceptance paradigm also leads to the problem of biassed reporting. Usually, positive results are much more exciting than negative ones, and so it is tempting to use low P values as a criterion for publications of results. By definition, though, a P value below 0.05 will be found in roughly 1 experiment in 20 even when no real effects are present. If this experiment is reported and the others are not, it is clear that the publication will be misleading. This is a serious worry in the medical and psychological literature.

There are alternatives to significance tests. Bayesian techniques can be used to place confidence limits on the values of particular parameters, and an approach using likelihood reasoning has been proposed by A.W.F. Edwards, whose book “Likelihood” (Cambridge U.P., 1972, expanded edition Johns Hopkins U.P. 1992, in library at QD 8000 Edw) contains a trenchant attack on significance testing.

Despite these difficulties, those who seek rigorous analysis of experimental results will often want to see P values, and provided its limitations are borne in mind, the hypothesis testing methodology can be applied in useful and effective ways.

Section 9 Chaotic systems and fractals

This section introduces some ideas related to the development of dynamical systems, and especially those that show chaotic behaviour. The associated idea of a fractal dimensions is introduced.

Contents

1 Introduction

Classical mechanics deals with how deterministic systems, such as swinging pendulums and orbiting planets, change with time. The dynamics of such a system are described by its *state*, which captures the values of all the variables that are needed to predict the future of the system, and a set of rules, often in the form of *differential equations*, which say how the state changes with time. For example, the state of a simple (undriven) pendulum is given by its angle to the vertical and the speed with which it is swinging. The rules that govern the pendulum are a pair of differential equations that involve only these two variables. (Parameters such as the length of the pendulum are not part of the state, because they are constant; such parameters are thought of as part of the rules.) The differential equations are solved, with the current state as an initial condition, to predict or simulate the evolution of the system. This is done analytically in very simple cases, but more often numerically (see Section 6).

(Quantum mechanics, by contrast, describes systems that are fundamentally non-deterministic, and have unavoidable randomness. Real physical systems appear to have this quality, though on a large scale they are well described by deterministic laws. In quantum mechanics, the probabilities of the system's being in various states evolve according to deterministic rules, but the future state itself cannot be predicted exactly. A third approach, statistical mechanics, is used to describe systems that are supposed to be fundamentally deterministic, but where probabilities are used to represent our ignorance of the microscopic details of the system.)

Two things might seem to be sufficient for the study of a deterministic system: (1) discovering the rules governing it; and (2) finding sufficiently accurate analytic or numerical techniques to use these rules to predict its evolution. Indeed, this is true for many practical purposes. However, understanding a dynamical system involves more than this, and we often want to characterise and classify systems rather than to treat each in isolation, in order to gain some insight into their structure. There are many ways to approach this, but here we concentrate on one or two of the more graphical ones, which have come to increased prominence recently, although their roots were established many years ago.

Simple physical systems often provide analogies for the study of living or artificially living systems. One important class of behaviour that is increasingly identified in physical systems is *chaotic* behaviour, which is of particular interest because simple deterministic rules produce high degrees of complexity and unpredictability in the system itself. Chaos can readily be demonstrated with simple simulations, but may play in important role in understanding the evolution of complex simulations with large numbers of parameters and variables.

Central to the idea of chaos is that tiny fluctuations in the current state of a system produce large changes in its future state. To describe this property, it is necessary to look at smaller and smaller variations in the state. This idea of moving to every smaller scales and ever greater detail leads to the use of *fractal* descriptions of the state trajectories associated with this kind of dynamics.

A good introduction to chaotic dynamics can be found in “Chaotic Dynamics: an introduction” by G.L. Baker & J.P. Gollub (Cambridge U.P., 1990, in library at QE 5360 Bak). There are many books on fractals, which because of the application to computer graphics are often very attractively illustrated; a good example is “The Science of Fractal Images” edited by H.-O. Peitgen & D. Saupe (Springer-Verlag, 1988, in library at QD 2500 Sci). Also possibly of interest is “Complexity : life at the edge of chaos”, by R. Lewin (Dent, 1993, in library at Q 1200 Lew) — and see also the various books in the library catalogue with “chaos” and “fractal” in their titles.

2 System description using phase space

2.1 Basic idea

A central concept in dynamical systems is that of *state*. The idea is that the state of a system is known, then no further information will be any help in making predictions about the system. Specifying the state separates the future from the past, in that everything in the system’s history is irrelevant to predicting the future provided the state is known.

One way of thinking of this in relation to simulations is to imagine a procedure which is called whenever the simulated time is to be advanced by one tick of the clock. If the procedure has no local memory in which it can store variable information between calls to it, the arguments it is given and the results it returns will necessarily represent the state of the system.

A physical example is given by the solar system. The laws of gravity provide the rules which govern the motion, which together with the masses of the Sun and planets provide the fixed information. The state is given by the positions and velocities of the heavenly bodies in some suitable frame of reference. If these are known, then there is no point in knowing where Jupiter was last week, or what its current acceleration is; such extra information is simply redundant.

Similarly, in a genetic program, the state at a point between generations might be fully represented by the genomes of all the current individuals. Extra information, such as their previous levels of performance on some task, or their ancestry, could be irrelevant to what will follow, in that the algorithm progresses with no information other than the genetic sequences.

The state will usually be a collection of the values of some variables. It seems reasonable to put this collection into a single structure, and this is then known as a *state vector* (see Sections 4 and 5). Given that, it is a small step to decide to see whether the geometrical interpretation of the vector is any use, and to start thinking of it as representing position in an abstract *state space* or *phase space*.

Note, however, that these vectors do not represent simple physical entities, and their components have varying interpretations. For example, if a system consisting of a single object moving in 3-D is being analysed, its state vector might well need 6 components, 3 for position and 3 for the components of velocity. Phase space for this object will be 6-dimensional. If the object also had a direction it was facing different from its direction of motion, then a further 2 components might be needed to represent its orientation in space. As with any high dimensional representation, the value as a conceptual tool depends in part on whether there is a sensible way to look at it in 2 or 3 dimensions in order to produce pictures.

2.2 Example: the undriven small-angle pendulum

An example of the use of phase space is provided by a simple pendulum. The point of using such a simple system is that its phase space is already low-dimensional, so it is easy to produce graphical representations.

The state of the pendulum is given by two variables. The first is its angle to the vertical, θ, which will be positive when it is to the right of centre and negative when it is to the left, say. The second is the rate of change of this angle, or the angular speed, ω. The state might be written as (θ, ω), and phase space is a 2-D plane with θ and ω as the axes.

If the amount of swing is small, the rule governing the state is given by a matrix relation (Section 3):

$$\begin{bmatrix} \frac{d\theta}{dt} \\ \frac{d\omega}{dt} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & -1/q \end{bmatrix} \begin{bmatrix} \theta \\ \omega \end{bmatrix}$$

To simplify the equation, I have assumed that the pendulum has a certain period (actually 2π or about 6.283 in whatever time units are being used). The parameter *q* is a number which describes how much damping (such as air resistance) there is: large *q* means little damping. The variable t stands for time.

The vector on the left is a vector of rates of change of the state variables. If the state vector is thought of as the position of a point, then the vector on the left is simply the velocity of that point. If this is not obvious, try replacing θ and ω by x and y, and remember that velocity components are just time derivatives of position components. The vector on the left gives the direction and speed of motion of the state vector through phase space.

Suppose q is very large, so the bottom right-hand element of the matrix is almost zero. Then the matrix is just a rotation matrix through -90˚ (see Section 4, part 4.2), since cos(-90˚) = 0 and sin(-90˚) = -1. That means that wherever the state is in phase space, its motion will be at right angles to the line joining it to the origin. In other words, the state will go round in a circle. (See also the discussion of flow fields in Section 5.) The *trajectories* of the system in phase space are circles around the origin. This set of circles characterises the behaviour of the pendulum — and of course, the cyclical nature of the circle represents the oscillation of the undamped pendulum. The circles are known as *limit cycles*.

If q is made smaller, so that there is some damping, then $d\omega/dt$ will have an extra contribution of $-\omega/q$. This allways acts to pull the point inwards towards the θ axis. (Plot a few of the vectors corresponding to this contribution if this is not clear.) This means that the state will spiral inwards towards the origin, wherever it starts from. This corresponds to the running down of a free pendulum which is damped by air resistance. Since all trajectories end up at the origin (which is the state of no motion and a vertical pendulum), the origin is called an *attractor* of the system.

In fact, the idea of an attractor is generalised to include limit cycles and other structures in phase space that characterise the long term behaviour of a system.

You may have met the elementary analysis of the simple pendulum, in which the differential equations are solved to get an explicit equation for θ as a function of t. The graphical way of looking at it using phase space provides a much more powerful and general technique for studying dynamical systems.

On a practical note: the matrix equation can be used in writing a simulation of the pendulum. At each time step, the derivative $d\theta/dt$ is multiplied by the size of the time step in order to compute the amount by which to change θ (see Section 6). Likewise for ω. Baker & Gollub's book (see above) uses Runge-Kutta integration to get the numbers, but Euler integration with a small time step works well, and will replicate the behaviour demonstrated in Baker & Gollub's examples. It is not difficult to write a program which will display both the pendulum's motion and the phase space trajectories — see below for details of one such program to display trajectories.

2.3 Some properties of state trajectories

It is worth noting that trajectories in phase space cannot cross one another. The reason is simple: at the crossing point, you would need to know which trajectory you were on in order to know in which direction to continue. But the position of the point by itself is sufficient, by the definition of 'state', to know which way to go next — knowing which of two trajectories was being followed is bound to be superfluous. So two trajectories cannot pass through the same point and continue on the other side, though they can converge on an attractor.

A second property is that for systems that do not lose energy, such as the pendulum with very large (infinite really) q, area is conserved in phase space. What this means is that if you take a bunch of states that cover a region of phase space, and let them all evolve for a while, then they will be covering a new region of phase space, but with the same total area. This obviously happens for trajectories that move round concentric circles. If the system loses energy, as when the pendulum is damped, then the area covered reduces with time — the points get squeezed together as they approach the attractor.

In general, the advantage of a phase space description is that it allows a *qualitative* description of the system, in terms of the main features of the trajectories rather than in terms of arrays of numbers. In addition to point attractors and limit cycles, phase spaces are characterised by features such as critical points which are *unstable* points, where trajectories diverge, or *saddle points* where trajectories converge along one direction and diverge along an orthogonal direction. Areas round attractors are described by *basins of attraction*: any trajectory starting in the basin of attraction round an attractor will finish up at the attractor, like water in the catchment area of a reservoir.

A practical use of phase diagrams is in research in biomechanics, where they are used in the analysis of human motion. For example the angle at a person's elbow might be plotted against the rate of change of that angle during a reaching task, in order to make disorders of motor control graphically apparent. Phase descriptions of the cardiac cycle are important in trying to understand how certain failures of cardiac rhythm arise.

3 The phase space of a chaotic system

3.1 The driven nonlinear pendulum

The simple pendulum served to introduce the idea of phase space, but it did not illustrate the idea of chaos. To do this, two modifications are needed. The first is that the system is made *nonlinear* — that is, that the velocity of the state vector is not just a matrix multiplication of the state vector itself. For the pendulum, this can be achieved by using the more physically realistic equation that is needed to describe a pendulum with a larger amplitude of swing. The equation is

$$\begin{bmatrix} \frac{d\theta}{dt} \\ \frac{d\omega}{dt} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & -1/q \end{bmatrix} \begin{bmatrix} \sin\theta \\ \omega \end{bmatrix}$$

where the only change is the nonlinear function sin applied to θ before the matrix multiplication is done.

The second change is that the pendulum needs to be sustained by a periodic force which operates at a different frequency from the pendulum's natural frequency. Such a force must be applied by some external mechanism, and will be taken as an angular force on the pendulum which varies sinusoidally. The current value of this force is another state variable, which will be called phi, so the state vector has three elements. The equations with the driving force are

$$\begin{bmatrix} \frac{d\theta}{dt} \\ \frac{d\omega}{dt} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & -1/q & g \end{bmatrix} \begin{bmatrix} \sin\theta \\ \omega \\ \cos\phi \end{bmatrix}$$

$$\frac{d\phi}{dt} = k$$

where *g* and *k* are two new parameters. The last equation just means that ϕ increases steadily, which means that cos(ϕ) oscillates. The effect of the force is to add a term *g* cos(ϕ) to the ω component of the state velocity, so that this vector will wobble in phase space as time progresses.

Since ϕ changes steadily in an uninteresting way, the usual way to view the trajectories is to plot ω against θ as before, but to bear in mind that this is only a *projection* of the real 3-D trajectory, which winds outwards from the θ-ω plane along the ϕ axis. If the 3-D trajectory is imagined as a wire winding across a room, it is as if the shadow of the wire was projected onto one wall by a distant light source. Although the trajectory cannot intersect itself, its projection can.

The details of these particular equations are not important. For particular values of *q*, *g* and *k*, this wobble has extreme consequences for the trajectories, and what is important is to see the level of complexity that is needed in the equations to allow a qualitative change in the behaviour.

The nature of the change is that the trajectories are no longer confined to tidy paths in phase space, but wander across it. The new equations produce a radically different picture when these trajectories are plotted. This occurs, for example, for $q = 2$, $g = 1.5$ and $k = 2/3$. An example program that will plot these trajectories is described below, or they can be found in books on chaotic dynamics.

A program which carries out Euler integration of the equation above is described in part 3.3.

3.2 The Poincare section

It is quite hard to interpret a diagram consisting of chaotic trajectories that wander across phase space or a projection of it. One way of simplifying the picture is to make use of the fact that the driving force is periodic: from any starting point, it returns exactly to what it was after a time interval of $2\pi/k$. If the trajectories are going to have any regularity, this is bound to show up if we look at the state of the system each time the driving force returns to any particular point in its cycle. If the system is behaving simply, then the (θ, ω) point will occupy one of a small selection of locations in phase space at this stage of the cycle, whereas if the system is chaotic, then the point might be found in a much larger number of locations.

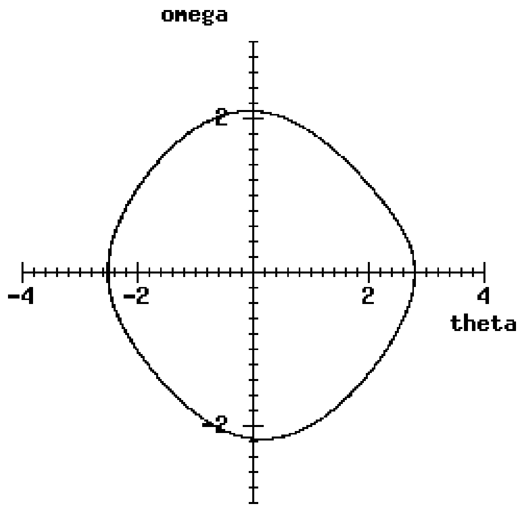
This is the idea of the Poincare section. Instead of plotting a more or less continuous succession of θ and ω values, we choose an arbitrary state of the driving force, and then plot θ and ω only at times when this state is reached. This can give a clearer picture of what is happening in phase space, and allows the structure of attractors to be seen more clearly. This is like superimposing a set of slices through the trajectory, all perpendicular to the ϕ axis and spread out along it. Another way of thinking of it is that it is like taking snapshots of the pendulum's state using stroboscopic lighting triggered by the driving force.

3.3 A demonstration

It is easier to understand this material by looking at pictures than by reading words. Examples can be found in books, but there is also a program which can be run if you are reading the online version of this file from Ved at a graphics terminal.

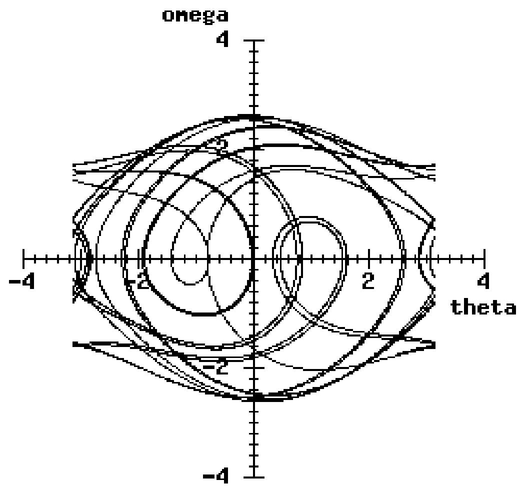
Although no substitute for the interactive version, two examples of the results are given here. First, for

$q = 2$, $k = 2/3$ and $g = 1$, the state trajectory is displayed as

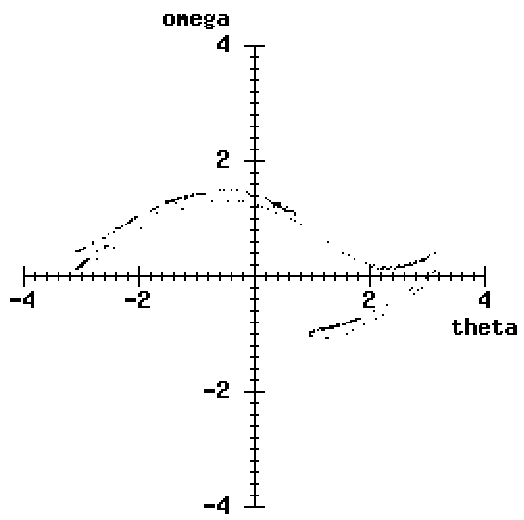


The Poincare section with these parameters is just a dot, somewhere on the trajectory shown above.

If, however, g is set to 1.5, the trajectory looks like



whilst the Poincare section is diffuse:



You could set up a similar demonstration fairly easily using a package such as Matlab.

3.4 Properties of chaotic systems

One of the most important characteristics of chaotic system is that small changes in initial conditions produce, over time, rapidly diverging trajectories in phase space. A block of neighbouring starting points in phase space will be smeared out across a wide region as time progresses.

A physical system which demonstrates such behaviour more obviously than the pendulum consists of a set of static billiard balls on a table, into which one ball is fired. If friction is somehow made very low, there may be many collisions. Now if a small change is made to the initial direction of motion of the moving ball, a much larger change will result its direction of motion after the first collision. The reason for this is that the initial angular change gets converted into a change in the point of collision between the two balls, and the point of collision affects the outward direction strongly. After only a few collisions, a microscopic change in the initial direction will produce a radically different configuration of the balls. In fact, a real billiard table is not chaotic because there is friction but no input of energy, and the trajectories end up on an attractor corresponding to stasis, but it could be made so by some means of adding energy to the system to keep the balls in motion, perhaps by vibrating the edges of the table.

Similar arguments are often applied to large-scale physical systems, with the atmosphere providing one of the chief examples: it seems likely that the weather is actually chaotic, and so a tiny perturbation in one part of the world will produce large changes elsewhere subsequently.

This unpredictability fits in with the diffuse nature of the Poincare section in these cases: at a fixed point in the driving cycle, the pendulum might be at all kinds of different states, and short of running the full simulation, there is no way of computing the state an arbitrary time into the future.

The pendulum simulation involved 3 parameters: the amount of damping, the strength of the driving force, and the frequency of the driving force. These parameters have to be given numerical values for a given simulation. Some combinations of values produce chaotic behaviour and some do not. One way to explore this division of the *parameter space* is to fix two of them and vary the third. If q and k are kept fixed, at say 2 and 2/3 respectively, then the behaviour shows an interesting variation with g. As g is increased through a range from say 0.5 to 1.5, the behaviour changes several times from non-chaotic to chaotic. The transition into chaos is not abrupt, however; at $g = 1.5$ there is chaos, but at $g = 1.49$, say, the trajectories are orderly but complex. In fact, as g is increased through the range just below 1.5, the trajectories display a phenomenon called *period doubling*, in which the trajectory of the pendulum in phase space requires 2, then 4, then 8 (and so on) cycles of the pendulum before it repeats itself. This can be displayed using *bifurcation diagrams*, which will be found in textbooks, and can be observed using the simulation mentioned above. The transition into chaos — the *"edge of chaos"* — is regarded as highly important in some theories of self-organising systems (see the book by Lewin mentioned above).

3.5 Other chaotic systems

This introduction has been in terms of a simulation of a simple physical system with continuous values for its state variables. Many other kinds of system exhibit chaos, however, and these include systems whose state space is *discrete* rather than continuous. Cellular automata are examples of such systems. An interesting example of a chaotic cellular automaton is given in the paper 'Evolutionary games and spatial chaos', by M. A. Nowak & R. M. May (Nature, 359, 826-829, October 1992). Here, agents on a 2-D grid interact with each other, playing a game called 'Prisoner's Dilemma', and adjusting their behaviour to copy that of their most successful neighbour. Remarkable chaotic graphic patterns ensue. If you read the online version of this from Ved, you can run this system from the file TEACH PD_GAME. (You don't need to know Pop-11 for this — just mark and load the examples in the file.) Although this is superficially a completely different system to the pendulum, there is again an adjustable parameter which causes transitions between chaotic and non-chaotic behaviour.

4 Fractals

The Poincare section for a chaotic system looks characteristically smeared out across the projection of phase space. Is it possible to make a more quantitative measurement that reveals the nature of this structure? There are various ways of doing this (including using the entropy — see Section 7), but one particularly interesting mathematical tool is the idea of *fractal dimension*. This idea will be briefly outlined here; there are now many textbooks dealing with the topic, and these are often worth looking at because of their excellent graphical presentation of the material.

4.1 Basic idea

The idea that a room has 3 dimensions, a sheet of paper 2 dimensions, and a line on it 1 dimension may already be familiar. Of course, the paper and the line are idealised: we pretend they have no thickness or width. The number of dimensions corresponds to the numer of coordinates that are needed to represent a position within the structure. In addition to these everyday examples, space-time is a 4-dimensional structure, whilst a point has 0 dimensions. In all these cases, the dimension is a whole number.

In fact, it is possible to give a fractional dimension a well-defined meaning. Some structures can sensibly be regarded as having 1.5 dimensions, for instance (and it turns out that the set of points in a Poincare sections is sometimes one of this type). Such a structure must be somehow more than a line, and less than a plane. One way to generate an object like this is to draw a straight line, then replace its middle third with two sides of an equilateral triangle, then to repeat this operation on each of the lines we now have, and to keep doing this indefinitely, with the lines getting shorter and shorter. (This is called the von Koch curve.) The structure is more than a line in the sense that there is a 2-D region of the plane where every point is less than some infinitesimal distance from it, but less than a plane in that every point that is part of it has a neighbouring point that is not. The structure is something like a sponge, occupying a part of some space but not filling it solidly.

The most significant aspect of a fractal is that it has structure on all scales. That is, if you look at it under a magnifying glass you will observe the same sort of detail that you could see with the naked eye; if the magnifying glass is replaced with a microscope, the appearance will still be roughly similar; and so on, however powerful the microscope is made. This property is known as *self similarity.* The von Koch curve was designed to have a particularly simple form of self similarity, but fractal structures in general exhibit it.

Apparently bizarre structures like this are not only relevant to describing chaotic systems. They appear to be useful models of a variety of natural systems. The classic example is the coastline of Britain, which on a large scale shows headlands and estuaries, on a smaller scale shows small inlets and rocky points, and so on down to the interstices between grains of sand. Recently, much has been made of fractal systems as representations of the structure of images of natural scenes, and the application of this to image encoding and compression.

4.2 Fractal dimension

A fractal is mainly characterised by its fractal dimension. This is measured by exploiting the self-similarity, and asking how the object appears to change as the scale of observation is adjusted. We start by looking at a different way of defining the dimensionality of non-fractal objects.

Take an ordinary 2-D shape and draw a grid over it. Let the size of one side of each box be *L*, and count the number of boxes lying over or partly over the shape. Call the number of boxes N_0. The ordinary area of the shape is roughly the number of boxes times the area of each one, or

$$A = N_0 L^2$$

so the initial number of boxes is given by

$$N_0 = A\left(\frac{1}{L}\right)^2$$

Now make the boxes *r* times smaller. That is, make the side of each box equal to *L*/*r*, by drawing extra lines between the original grid lines. The area of each box is now $(L/r)^2$, so the new value of *N*, which I write as a function of *r*, is

$$\begin{aligned} N(r) &= A\left(\frac{r}{L}\right)^2 \\ &= A\left(\frac{1}{L}\right)^2 r^2 \\ &= N_0 r^2 \end{aligned}$$

This won't be exact, as the curved edges of the shape will pass through the middle of some boxes at each scale, but as long as the original boxes are small enough that the estimate of *A* is reasonable, it will be a good approximation. And as the original boxes are made smaller and smaller, the approximation will be more and more accurate.

The quantity $A(1/L)^2$ is a constant, N_0, so what we have is that the number of boxes needed to cover the shape is proportional to r^2 in the 2-D case, where *r* is the scaling factor for the boxes.

Now consider a 1-D line. Let *A* stand for the length of the line. Split the line up into segments of size *L* and count the number needed to cover the line. The number of segments is $N_0 = A/L$. Then divide each segment into *r* parts and count again. The number needed is just

$$N(r) = N_0 r$$

i.e. in 1-D the number is proportional to the scaling factor.

Do the same thing in 3-D: split an object occupying some 3-D volume up into cubes, with *L* being the side of each cube. *A* is now the volume of the object. Then you get

$$N(r) = N_0 r^3$$

In each of these everyday cases, the power that *r* is raised to is the dimensionality of the object. Note that if you have, say, a 2-D surface embedded in a 3-D space, and you look at how many boxes you need to contain it, you still get the power of 2: it's the dimensionality of the object, not the dimensionality of the space it lives in, that is measured.

Now do the same for a fractal pattern. You start off as before with

$$N_0 = A\left(\frac{1}{L}\right)^2$$

as the number of boxes covering it. You only need to count boxes that actually have black dots in them. Now, *A* is a sort of "effective area" at the scale defined by the length *L* — it's more or less the area that would get covered if the dots, rather than being infinitesimal points, were dots of a size comparable with *L*. (They'd overlap of course.)

Then you make *L* smaller by dividing by *r*, as before. What happens then, though, is that you don't need as many boxes as you expect, because as the boxes get smaller, more of them fit into the white spaces between the dots. What happens is that instead of getting $N = N_0\, r^2$, you find that you get $N = N_0\, r^D$, where *D* is a number less than 2. But however small you make the boxes, you never get down to $D = 1$ as you would with a smooth curve — *D* flattens off at some definite intermediate value.

The number *D* is the fractal dimension of the object. For the von Koch curve it's actually about 1.26. The curve has the fractal dimension because however closely you look at it, there is always more structure to see, so the effective area gets smaller and smaller as you reduce the scale. However, the effective area never gets down to zero as it would for an ordinary curve (which has a fractal dimension of 1).

4.3 Iterated function systems — fractal dynamics

One way of generating fractal structures, exploited particularly in image processing, is the *iterated function system* (IFS) technique, associated particularly with M.F. Barnsley (see the book edited by Peitgen & Saupe mentioned in the introduction). This is of interest here as a second example of a dynamic system which can generate fractals, although this time the process is more direct in that simulation of a physical system is not involved; phase space is more or less manipulated directly.

In an IFS, a vector (here 2-D) is manipulated using matrix multiplication and addition. Consider a point (*x*, *y*) subjected to a matrix multiplication and a vector addition. The new point, (*x'*, *y'*) is given by

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

where *a*, *b*, *c*, *d*, *e* and *f* are 6 parameters of the transformation. If this is done repeatedly, replacing (*x*, *y*) by (*x'*, *y'*) at each stage, and the successive positions are plotted, the dots will form a trajectory in the plane. If the elements of the matrix satisfy a certain condition (it defines a contractive mapping), the trajectory will converge to an attractor.

In an IFS, a set of transformations like this is first defined, each with its own set of 6 parameters. There are typically a small handful of transformations in an IFS. Then at each iteration of the algorithm, one of the transformations is selected at random (using predefined probabilities) and applied to the point. The next iteration will again select a transformation at random, either the same one or a different one. The result is that the point defined by (*x*, *y*) follows a complex random walk around the plane, being continuously pulled to different attractors. The result is not, however, a superimposition of simple shapes, but (in general) a fractal pattern, showing self similarity.

Examples abound in books: the most standard is a fern leaf, generated by four transformations with the parameters

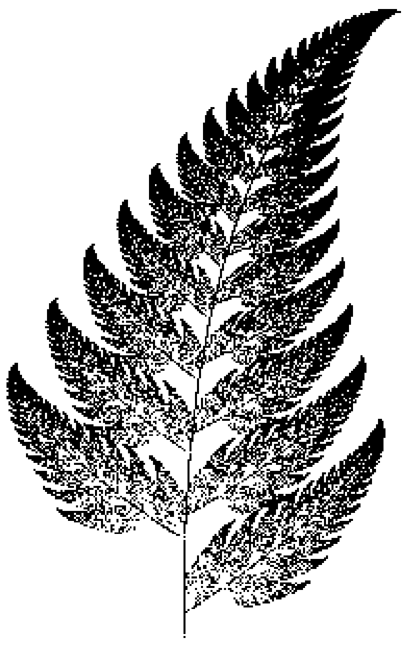
a	*b*	*c*	*d*	*e*	*f*	*P*
0.85	0.04	-0.04	0.85	0	0.3	0.85
-0.15	0.28	0.26	0.24	0	.0825	0.07
0.2	-0.26	0.23	0.22	0	0.3	0.07
0	0	0	0.16	0	0	0.01

(the probabilities *P* are not critical). The result is shown below. It is straightforward to write a program that generates the image of the pattern from these parameters. It is not difficult to come up with sets of parameters that generate other patterns.

Part of the interest in these systems is that an object that looks immensely complex if taken piecemeal is shown to be summarised by a rule that has only a small number of numerical parameters. This has both practical application to areas like image compression, and theoretical value in understanding complex dynamic systems.

4.4 Other fractal generators

The IFS brings together nicely the idea of a dynamic system described by a phase space trajectory, and the fractal dimension. However, many other rules can be used to produce fractal objects: two examples are the *logistic map*, used in studies of population dynamics, and the *L-system* which operates in the domain of



formal languages and their grammars.

5 Conclusion

This teach file differs from the preceding ones in that no new mathematics is involved. Differential equations, matrices, vectors and probability have all been encountered already. However, their application to chaotic dynamical systems opens up a rapidly developing area, which has major applications to the understanding of complex systems. Understanding systems with many interactions will involve a vast variety of techniques, from transparent simulations to extremely abstract formal mathematics. You will need to use imagination to find the level of analysis that is appropriate for your problem and your way of looking at it.