First Principles of Machine Learning
Notes

Jack Fraser-Govil

October 2024

Contents

Part I

Background Mathematics

Chapter 1

Introduction

This section is designed to provide a brief overview of the background mathematics which will be touched upon during the workshop.

It is **not** intended to be a comprehensive overview of the topics, and I will freely gloss over things that are uninteresting or irrelevant for the purpose at hand. This document is meant to be a quick reference guide, or a refresher for people who once knew these things, but have long since forgotten them.

If you read nothing else, it will be vitally important for you to have a grasp of the **chain rule**, since this is the cornerstone of backpropagation. Similarly, understanding Matrix multiplication (even if you can't actually do it by hand) and the dot product will be of vital importance for our work.

1.1 Some Basic Notation

I will be using (mostly) formal mathematical notation throughout, but will endeavour to ensure that whenever notation is first introduced, it is explained. I will also ensure that the surrounding text explains what is going on such that you should be able to gather from context clues what an expression means.

That being said, some basic notation is helpful to get out of the way:

- Scalar quantities (i.e. simple numbers and placeholders), will be written in lower case italics. x, y, z and so on.
  - An exception to this is *scaling parameters* (i.e. numbers which control how large a function is) will be written in curly fonts, $\mathcal{N}, \mathcal{A}$ if we don't actually care about their value.
  - Functions will also be written in lower case italics; $y = f(x)$[1].
- Vector quantities will be written in boldface. $\mathbf{x}$, $\mathbf{y}$, $\mathbf{a}$.
  - Since *elements* of a vector are (usually) scalars, they will be written in italics. $a_i = [\mathbf{a}]_i$ is the i^{th} element of the vector $\mathbf{a}$.
- Matrices (and Linear Operators) will be written in capital italics; M, N, P and so on.
- Derivatives will usually be written in Newton's Notation ($\frac{\mathrm{d}y}{\mathrm{d}x}$), but I will sometimes use Leibniz's notation ($f'(x)$) if the text is getting a bit cramped! This is purely a stylistic choice. Don't read into it.

[1] Technically I should reserve this for 'scalar functions' (where the output is a scalar); and let vector functions be written as $\mathbf{f}(x)$, for example. For simplicity, I won't do this: all functions will be written the same.

Chapter 2

Calculus

Calculus is the mathematical study of *change*. It is needed in order to study how a function varies as its inputs are altered. For the purposes of machine learning, therefore, it is of vital importance in the process of optimization, since we are (hopefully) trying to alter the parameters of a model until the loss function reaches a maximum: this is a process only possible via calculus.

2.1 Functions

The notation and theory which underlies even the simplest of functions is vast and complex (Real Analysis), and is a field where trivial-sounding statements are often impossible to prove.

Luckily, we don't need any of that.

For our purposes, it is sufficient to note that a function is any form of **mapping** between two sets of objects. It is common for these sets to be two different kinds of numbers: for instance the function $f(x) = x^2$ maps all (real) numbers to the set of all positive numbers. However, there is no general restriction that a function 'only' has to be a relationship between numbers.

A function is a black box, into which you put an input, and you receive an output. We denote this as:

$$y_{\text{output}} = f(x_{\text{input}}) \tag{2.1}$$

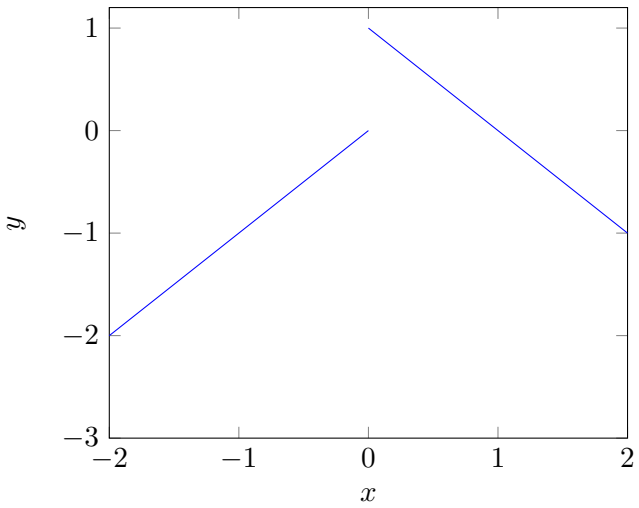
Here x and y can be anything; x might be a particular scent, and y could be the U2 album that smell most reminds my Dad of. In general, however, we will be most interested in cases where these are numerical quantities: either vectors or scalars.

2.1.1 Continuity

A function is said to be **continuous at a point** if it obeys the following relationship:

$$\lim_{x \to c} (f(x)) = f(c) \tag{2.2}$$

That is, the limit of the function is the same as the value of the function. This definition also implicitly requires that the limit be the same *no matter which direction you approach from.*



The function above is therefore non-continuous at $x = 0$ because the limit has different values depending on if you approach from the left or the right.

'Continuity' is therefore a fancy way of saying 'if you draw the function as a graph, you don't have to take your pen off the page' – however this definition generalises to arbitrary quantities where you can meaningfully define limits (such as 24-dimensional spaces where your pencil would have to curl in on itself).

2.1.2 Nested or Chained Functions

It will be exceptionally common (and very important for us!) to consider the cases where functions are nested. If f and g are both functions from some 'object space'[1] F to the same space (i.e. the output is the same type of object as the input), then it is possible to have:

$$y = f(g(x)) \quad \leftrightarrow \quad y = f \circ g \circ x \tag{2.3}$$

(This $\circ$ notation is a functional notation some mathematicians like. It cuts down on parentheses.) This should be read as "do g to x first, and then do f to the output of that".

Many more complex mathematical expressions can be written as nestings of more simple functions; for instance the Gaussian function *can* be written as:

$$y = \mathcal{N} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) \tag{2.4}$$

Or it can be written as:

$$y(x) = \mathcal{N} f(g(h(x))) \tag{2.5}$$
$$f(g) = \exp(g) \tag{2.6}$$
$$g(h) = -\frac{1}{2}h^2 \tag{2.7}$$
$$h(x) = \frac{x-\mu}{\sigma} \tag{2.8}$$

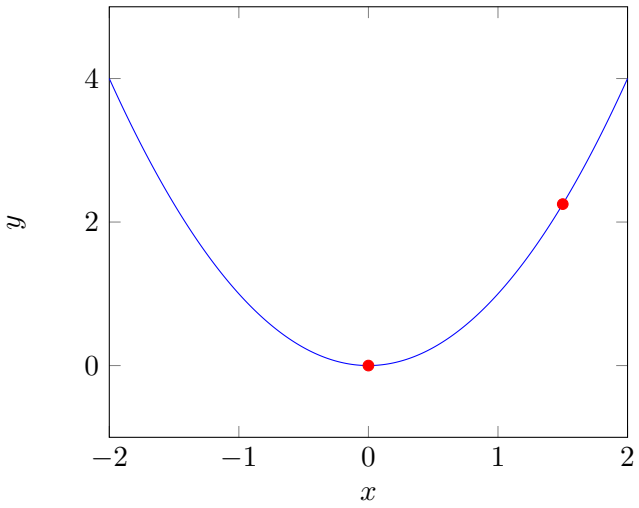
2.2 Derivatives

We will now assume that we are dealing solely with real functions; those where the inputs and outputs are both real numbers. When you have such a function, it is natural to ask "how does the output change when I change the input?".

[1] A field

This is the **gradient** of the function: for a function $y = f(x)$, it is the amount that y changes, when x changes by a small amount.

It is clear that the answer to this question varies, depending where you are looking. In the plot below, the gradient at $x = 0$ (the first red dot) is clearly very flat; the value of the function at $x = 0.1$ is 0.01; very close to zero, and so the function has not changed that much. At $x = 1.5$ (and $y = 2.25$), however, the gradient is much steeper: increasing x by 0.1 increases y by 0.31 – 31 times larger than the increase we saw at $x = 0$!



The gradient of $f(x)$ is therefore a second function; when evaluated at x it tells you how steep the original function is at that point. This gradient function is the *derivative* of the function, and is denoted:

$$\text{gradient of } f \text{ at } x = \frac{\mathrm{d}f}{\mathrm{d}x} = f'(x) \tag{2.9}$$

Isn't that a fraction?

Despite being written using familiar notation, mathematicians will generally start shouting at you if you treat $\frac{\mathrm{d}f}{\mathrm{d}x}$ like a fraction. They'll start howling about things like 'infinitesimals' and 'surreal numbers'.

The dirty secret that mathematicians don't want you to know is that it is *often perfectly fine* to do so with single-valued functions, and things behave as intuitively as you might expect:

$$\frac{\mathrm{d}x}{\mathrm{d}x} = 1 \tag{2.10}$$

$$\frac{1}{\left(\frac{\mathrm{d}y}{\mathrm{d}x}\right)} = \frac{\mathrm{d}x}{\mathrm{d}y} \tag{2.11}$$

$$\frac{\mathrm{d}f}{\mathrm{d}x}\mathrm{d}x = \mathrm{d}f \tag{2.12}$$

However, when it comes to *multivariable calculus*, and things start looking like $\frac{\partial f}{\partial x}$, then you should never treat them as fractions.

2.2.1 Formally Defining the Derivative

Even though it's not strictly necessary, it can often be helpful to recall the formal definition of the derivative.

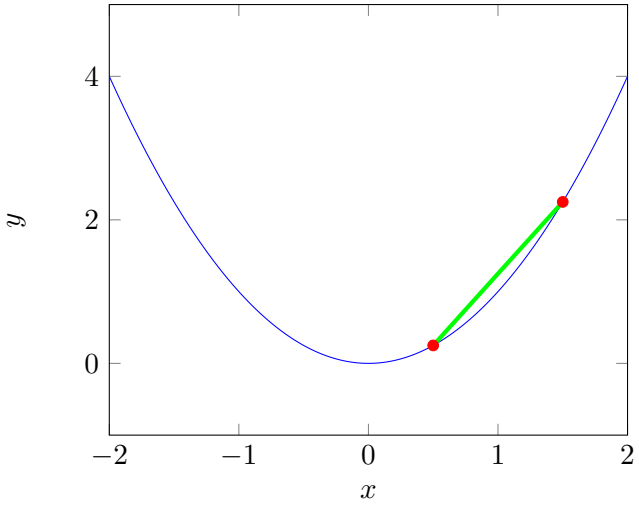


Figure 2.1: A Chord-approximation using $x_1 = 0.5$ and $x_2 = 1.5$

Consider a function $f(x)$ which you are able to calculate, but you would like to know the **gradient** of. In the absence of any other ideas, tt is reasonable to try and estimate the gradient using a **chord**.

To do this, we choose a point Δx away from x, and then draw a line between the two points. An example of this is shown in Fig. 2.1. The gradient of this choord can easily be computed as:

$$g = \frac{\Delta y}{\Delta x} = \frac{f(1.5) - f(0.5)}{(1.5) - (0.5)} \tag{2.13}$$

Since (in this case) $f(x) = x^2$, this simplifies down to:

$$g = \frac{2.25 - 0.25}{1} = 2 \tag{2.14}$$

Therefore the gradient *of the chord* is equal to 2; but we can see that this isn't quite the gradient of the function – the chord is much steeper! The chord is merely an approximation to the true gradient.

If we reduce Δx, then maybe we get a better approximation (see Fig 2.2):

$$\begin{aligned} g &= \frac{\Delta y}{\Delta x} \\ &= \frac{f(0.75) - f(0.5)}{(0.75) - (0.5)} \\ &= \frac{0.5625 - 0.25}{0.25} \\ &= 1.25 \end{aligned} \tag{2.15}$$

This is clearly much closer! We could repeat this exercise using the formula:

$$g \approx \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{2.16}$$

If we repeat this with successively smaller Δx, we would (hopefully) eventually converge on the correct final answer.

This is what differentiation is. It is the above process, taken to the infinite limit where Δx becomes infinitesimally small:

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \lim_{\delta \to 0} \left(\frac{f(x + \delta) - f(x)}{\delta} \right) \tag{2.17}$$

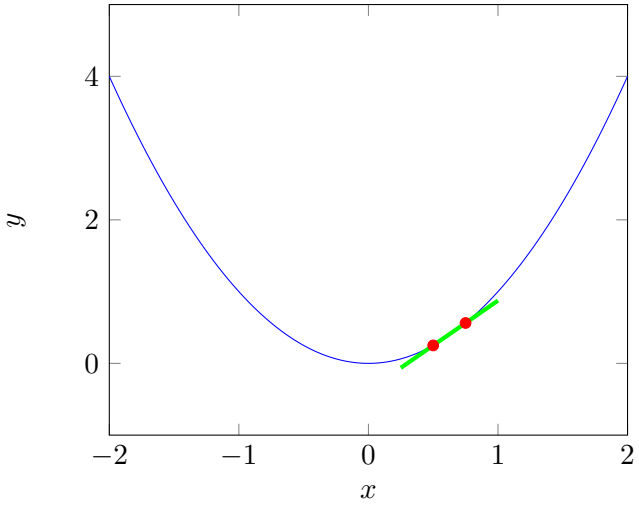


Figure 2.2: A Chord-approximation using $x_1 = 0.5$ and $x_2 = 0.75$

Of course, the problem with this is that dividing by zero is undefined, so you can't just put $\delta = 0$ in here! The process works because, as you divide by zero, the number on the top *also* becomes zero (because $f(x+\delta)$ and $f(x)$ become very similar) - so you are dividing an infinitely small value by another infinitely small value; in the hope that it cancels out, and you are left with something sane.

The joy of the above definition is that you don't **actually** need to care about it – it is the formal definition of a derivative, and can be helpful to remember what it means (especially when optimising). When dealing with an actual function, however, the derivatives are (usually) expressible in terms of other functions.

I won't prove these results now, but it can be shown that:

$$\frac{\mathrm{d}}{\mathrm{d}x}x^n = nx^{n-1} \tag{2.18}$$

$$\frac{\mathrm{d}}{\mathrm{d}x}\cos(x) = -\sin(x) \tag{2.19}$$

$$\frac{\mathrm{d}}{\mathrm{d}x}\log(x) = \frac{1}{x} \tag{2.20}$$

$$\frac{\mathrm{d}}{\mathrm{d}x}\exp(x) = \exp(x) \tag{2.21}$$

A discussion of how these values arise can be found in chapter 7.

2.3 Properties of the Derivative

This can all seem quite complex and overwhelming, but the derivative is in fact a lovely object, and has a number of desirable properties.

It is a Linear Operator

This means that if you know the derivative of f and the derivative of g, then:

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(f(x) + \alpha g(x)\right) = \frac{\mathrm{d}f}{\mathrm{d}x} + \alpha\frac{\mathrm{d}g}{\mathrm{d}x} \tag{2.22}$$

You can split apart sums, and pull constant functions out of the derivative, leaving you to focus on the nasty bits.

It Vanishes at Maxima and Minima

Maxima or a Minima are, by definition, the local extrema of a function – they are (locally) the highest or lowest values the function achieves. It must therefore, follow, therefore, that the gradient is zero at these points – or else you could increase (for maxima) or decrease (for minima) the value of the function by stepping in the direction of the non-zero gradient.

Finding the extremal points of a function can therefore be reduced to finding the solution(s) to:

$$f'(x) = \frac{\mathrm{d}f}{\mathrm{d}x} \quad \Longrightarrow \quad f'(x) = 0 \tag{2.23}$$

Products are Easy

The linear operator makes adding functions trivially easy. Multiplying functions is a bit harder, but is still a simple rule:

$$\frac{\mathrm{d}}{\mathrm{d}x}\Big(f(x) \times g(x)\Big) = f(x)\frac{\mathrm{d}g}{\mathrm{d}x} + g(x)\frac{\mathrm{d}f}{\mathrm{d}x} \tag{2.24}$$

For example:

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}x}(x\sin(x)) &= \sin(x)\frac{\mathrm{d}x}{\mathrm{d}x} + x\frac{\mathrm{d}\sin(x)}{\mathrm{d}x} \\ &= \sin(x) + x\cos(x) \end{aligned} \tag{2.25}$$

Chains are Possible

It is possible to 'chain together' derivatives, allowing you to take derivatives of functions-of-functions. For example, the function $f(x) = \log(x^2)$ can be thought of as $f(x) = \log(g(x))$ where $g(x) = x^2$.

In this case you can use the chain rule:

$$\frac{\mathrm{d}}{\mathrm{d}x}\Big(f(g(x))\Big) = \frac{\mathrm{d}f}{\mathrm{d}g}\frac{\mathrm{d}g}{\mathrm{d}x} \tag{2.26}$$

Here you treat g as a variable in the first term, and then like a function in the second. In our above example we have $f = \log(g)$ and $g(x) = x^2$ so:

$$\begin{aligned} \frac{\mathrm{d}\log(x^2)}{\mathrm{d}x} &= \left(\frac{1}{g}\right) \times (2x) \\ &= \frac{2x}{x^2} = \frac{2}{x} \end{aligned} \tag{2.27}$$

(We would have gotten the same result if we had used $\log(x^2) = 2\log(x)$ and then used the linearity condition!)

You Can Repeat Derivatives

Since $\frac{\mathrm{d}f}{\mathrm{d}x}$ is itself a function, there's nothing stopping you from taking another derivative:

$$\frac{\mathrm{d}^2 f}{\mathrm{d}x^2} = \frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{\mathrm{d}f}{\mathrm{d}x}\right) \tag{2.28}$$

This second derivative has some useful properties (it can help infer if an extrema is a maxima or a minima, for example), but in general, it is useful to know that you can take an infinite number of derivatives, if you wanted to.

2.4 Finding Extrema

If you are able to compute the derivative of a function, $\frac{\mathrm{d}f}{\mathrm{d}x} = f'(x)$, then finding the extrema is as simple as finding the zeros fo the function. Consider the polynomial $f(x) = 4x^3 - 9x^2 - 12x - 2$. This has derivatives:

$$f'(x) = 12\left(x^2 - \frac{3}{2}x - 1\right) = 12(x-2)\left(x + \frac{1}{2}\right) \tag{2.29}$$

Therefore the gradient has zeros at $x = -0.5, 2$, and so the function has extrema at these locations (since the function is dominated by a $+x^3$ term, we can also immediately infer that $x = -0.5$ is a maxima, and $x = 2$ is a minima).

This works great: **if your derivative-equation has an analytical solution**. This will not always be the case. For example:

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(x\sin(x)\right) = \sin(x) + x\cos(x) \tag{2.30}$$

This is a perfectly well formed derivative, however the solution to the maxima are of the form:

$$x + \tan(x) = 0 \tag{2.31}$$

This has infinitely many solutions, but only $x = 0$ is analytically solvable, the rest must be computed numerically.

This highlights an important problem: even if you can write out your derivative in a lovely analytical form, you might still have to resort to inelegant, brute-force methods for finding the optima. This is almost always the case in Machine Learning applications, and is why optimisation and training is such and important part of the process!

Chapter 3

Vectors & Matrices

If I were being cruel and rigorous, I would begin this section by saying that vectors are 'abstract members of a Vector Space', defining them through abstract and ethereal properties, that seem to have no tangible link to 'a vector'. In this language, vectors can be anything; functions, matrices, abstract quantum states, websites[1] and motorway junctions.

That doesn't seem the most helpful approach here, however (I have relegated this discussion to chapter 8). Instead, we will use the following definitions:

- A Vector is a structured store of numerical information
- A Matrix is a (linear) operation on that structured information

What I have done here is taken the abstract definition and performed an operation known as *projecting onto a basis* – what I am calling a Vector in this section is merely one *projection* (or *representation*) of a vector.

The most common form of Vector will be a 'simple linear vector' of dimension N, which we can write as a column of N numbers:

$$\mathbf{v} = \begin{pmatrix} a \\ b \\ c \\ \vdots \\ z \end{pmatrix} \tag{3.1}$$

When this information represents a spatial location (for example $\mathbf{x} = (x, y)$ is the xy coordinates), this correlates with the highschool definition of a vector as 'a quantity that has both magnitude and direction'.

For our purposes, however, it is often going to be the case that these objects do not have a meaningful 'direction', but are instead ordered lists of quantities, measurements and data, encodings of information of use to us. Although it is possible to interpret an input image as a 'vector with length and direction' just because we've transported it as a vector of pixel intensities, doesn't necessarily mean this is meaningful – though in Chapter 9 I do elaborate how we might meaningfully define a way to interpret angles, with a suitable choice of inner product...

[1] Yes, seriously, this is how Google's PageRank used to work!

3.1 Vector Algebra

Vectors can be trivially added together and multiplied by scalars[2]:

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} + x \begin{pmatrix} d \\ e \\ f \end{pmatrix} = \begin{pmatrix} a + xd \\ b + xe \\ c + xf \end{pmatrix} \tag{3.2}$$

Multiplying Vectors is rather more complicated, and there are multiple different ways to do it.

Elementwise Product

The simplest vector product is also amongst the least used:

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} \odot \begin{pmatrix} d \\ e \\ f \end{pmatrix} = \begin{pmatrix} ad \\ be \\ cf \end{pmatrix} \tag{3.3}$$

This is known as the element-wise or **Hadamard product**, and there is a similar definition for the Hadamard divisor ($\oslash$). This (usually) isn't used very much at all in matrix or vector algebra, though there are some places where it is useful – the definition of the ADAM optimiser, when written in full vector form defines the step vector $\mathbf{s}$ as:

$$\mathbf{m}_n = \beta_1 \mathbf{m}_{n-1} + (1 - \beta_1) \frac{\partial f}{\partial \mathbf{x}} \tag{3.4}$$

$$\mathbf{v}_n = \beta_2 \mathbf{v}_{n-1} + (1 - \beta_2) \frac{\partial f}{\partial \mathbf{x}} \odot \frac{\partial f}{\partial \mathbf{x}} \tag{3.5}$$

$$\mathbf{s}_n = \mathbf{m}_n \oslash \sqrt{\mathbf{v}_n} \tag{3.6}$$

My general experience is that the Hadamard product generally only appears in situations where vectors are being used to simplify group operations, rather than in cases where vector algebra is being used. In ADAM, for instance, each element of the vector is being updated independently, even though the gradient is technically a vector object; the update formula is using each element of the vector as its 'own thing', rather than as a valid vector component.

Cross Product

The Cross Product, like the Elementwise product, takes two vectors of dimension N, and returns a third N dimensional vector:

$$\mathbf{a} \times \mathbf{b} = \mathbf{c}$$

This has some weird properties – it is anti-commutative, so $\mathbf{b} \times \mathbf{a} = -\mathbf{a} \times \mathbf{b}$, which is a bit weird.

Importantly for our purposes, the Cross Product is only meaningful for Vectors in 3D space, where there is a reasonable geometric interpretation. There are generalisations to higher dimensions and arbitrary vector spaces (using the lovely Levi-Cevita tensor). If you are not trying to do physics or computer graphics where surface normals are of the highest concern, you never need to think about the cross product.

Dot/Inner Product

The Dot product (also called the inner product) is a meaningful operator on almost every vector we will encounter[3]. The dot product is a mapping between the vector space, and a scalar, and is usually defined as follows.

[2] This is in fact one of the 'abstract definitions' of a vector

[3] Very few interesting Vector Spaces for us cannot be turned into an Inner Product space.

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} \cdot \begin{pmatrix} d \\ e \\ f \end{pmatrix} = ad + be + cf \tag{3.7}$$

The dot product is, technically speaking, the special inner product on $\mathbb{R}^n$, normal Euclidean space: $\langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a} \cdot \mathbf{b}$. However, there is almost always a generalised Inner Product which looks and behaves like the dot product when our vectors are projected into column vector format.

Again, it is worth reiterating because dot products form a vital part of Machine Learning: **when you take the dot product of two vectors, you multiply elementwise, then take the sum**. That's all it is.

3.2 Linear Operators

Before we move on to talk about Matrices, it is first useful to first stop by and discuss Linear Operators.

A Linear Operator is an abstract mathematical object which '**does something**' to a vector. What this 'something' is, is left vague and undefined; Linear Operators are essentially special functions which obey some additional rules – and which use slightly different notation.

An operator acts on a vector space of N dimensions, but can output vectors of any dimension. All that matters is that any linear operator $\hat{M}, \hat{N}$ has to obey the following rules:

$$\hat{M}(\mathbf{x} + \alpha\mathbf{y}) = \hat{M}\mathbf{x} + \alpha\hat{M}\mathbf{y} \tag{3.8}$$

$$(\hat{M} + \hat{N})\mathbf{x} = \hat{M}\mathbf{x} + \hat{N}\mathbf{x} \tag{3.9}$$

$$\hat{M}\Big(\hat{N}\mathbf{x}\Big) = \Big(\hat{M}\hat{N}\Big)\mathbf{x} \tag{3.10}$$

However, note that it is not a requirement (and will not be true in general) that $\hat{M}\hat{N} = \hat{N}\hat{M}$, operators do not commute. This can be fairly trivially seen: the operator 'put on your shoes' and 'put on your socks' obviously matter which order you do them!

This seems obvious when expressed like this; it is, however, a different matter to remember when you have algebra on the page that you can't simply swap the order of things like you can with scalars!

3.3 Matrices

In chapter 10, I demonstrate the simple fact that *matrices are linear operators.* This is their most important definition. A matrix – in mathematical terms – is a way of representing a linear operator in the same projection space that you are representing your vectors.

That is, if you are dealing with 'abstract vectors', then you have to work with 'abstract operators'. If you have turned your vectors into column vectors, however, then you have also turned all of your operators into matrices.

When you perform this 'projection', these abstract 'mappings from an n dimensional vector space into an n dimensional vector space' become a *grid of numbers* – and the size of that grid is m rows by n columns: this is not a coincidence!

If $m = n = 2$, then:

$$M = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{3.11}$$

Where a, b, c and d are scalar values.

3.3.1 Matrix Algebra

Matrices of the same size can trivially be added and it works as you might expect: elementwise

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} + \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} a+e & b+f \\ c+e & d+h \end{pmatrix} \tag{3.12}$$

You can also multiply a matrix by a scalar with similar results:

$$\alpha \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \alpha a & \alpha b \\ \alpha c & \alpha d \end{pmatrix} \tag{3.13}$$

3.3.2 Applying A Matrix To A Vector

This is what we're all here for. Matrices are linear operators, which are defined via their 'acting on' vectors. Given a matrix M and a vector $\mathbf{v}$, what is the result? In short, if $\mathbf{u} = M\mathbf{v}$, what is $\mathbf{u}$?

Formally speaking, if M_{ab} is the element of M in row a and column b, then the i^{th} element of $\mathbf{u}$ is:

$$u_i = \sum_j M_{ij} v_j \tag{3.14}$$

This can be a little hard to understand in the abstract – but it is the definition which is used in 'ludicrously high dimensions' when visualising it is impossible. If you are in low dimensions, you can do the following:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} \\ \begin{pmatrix} d \\ e \\ f \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} \\ \begin{pmatrix} g \\ h \\ i \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} \end{pmatrix} = \begin{pmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{pmatrix} \tag{3.15}$$

That is, matrix multiplication can be though of as using each row of the matrix to perform a dot product with your 'target vector'. The dimension of your output vector is therefore obviously equal to the number of rows in your matrix; this is the number of dot products you are performing.

3.3.3 Matrix Multiplication

You can also multiply matrices by other matrices. This follows the same procedure as above, but repeated over multiple rows:

$$\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix} \begin{pmatrix} g & h \\ i & j \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \cdot \begin{pmatrix} g \\ i \end{pmatrix} & \begin{pmatrix} a \\ b \end{pmatrix} \cdot \begin{pmatrix} h \\ j \end{pmatrix} \\ \begin{pmatrix} c \\ d \end{pmatrix} \cdot \begin{pmatrix} g \\ i \end{pmatrix} & \begin{pmatrix} c \\ d \end{pmatrix} \cdot \begin{pmatrix} h \\ j \end{pmatrix} \\ \begin{pmatrix} e \\ f \end{pmatrix} \cdot \begin{pmatrix} g \\ i \end{pmatrix} & \begin{pmatrix} e \\ f \end{pmatrix} \cdot \begin{pmatrix} h \\ j \end{pmatrix} \end{pmatrix} = \begin{pmatrix} ag + bi & ah + bj \\ ag + di & ch + dj \\ eg + fi & eh + fj \end{pmatrix} \tag{3.16}$$

In formal language, the product AB produces a third matrix C, which has elements:

$$C_{ij} = \sum_k A_{ik} B_{ki} \tag{3.17}$$

If A has dimensions $\ell \times m$[4] and B has dimensions $m \times n$, then the output vector has dimensions $\ell \times n$.

Matrix multiplication is not commutative! It is not true that $AB = BA$ except in very rare circumstances. You have to be careful of the order you multiply matrices.

Recall, however, that by the definition of a linear operator, it is perfectly possible to chain any arbitrary number of matrices together, provided that they all have the correct number of dimensions.

$$A = BCDEFGHIJK \tag{3.18}$$

If B has dimensions $a \times b$, and K has dimensions $c \times d$ (and all the inner matrices have appropriate complementary dimensionality); the final dimensionality of this is $a \times d$.

That is, even if E has 10,000 $\times 10,000$ dimensionality; that is largely irrelevant for the final value. This will turn out to be of vital importance for Machine Learning...

3.3.4 Matrix Transpose

The *transpose* of a matrix is the result of pivoting it about the leading diagonal. This converts $m \times n$ matrices into $n \times m$ matrices:

$$\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix}^\intercal = \begin{pmatrix} a & c & e \\ b & d & f \end{pmatrix} \tag{3.19}$$

Transpose have the following effect on matrix products:

$$(AB)^\intercal = B^\intercal A^\intercal \tag{3.20}$$

A symmetric matrix is one which

$$A^\intercal = A \tag{3.21}$$

3.3.5 Interpreting Vectors as Matrices

It is possible to treat an m dimensional column vector as a $m \times 1$ dimensional matrix – much of the same algebra applies.

For instance, it is common write the dot product of two vectors as:

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^\intercal \mathbf{b}$$

This makes it easy to prove things such as:

$$\mathbf{a} \cdot (M\mathbf{b}) = (M^\intercal \mathbf{a}) \cdot \mathbf{b}$$

When you start treating vectors in this fashion, what you are doing is treating them as 'dot product operators' the vector $\mathbf{v}$ becomes the operator which, when applied to $\mathbf{u}$ computes $(\mathbf{v}^\intercal \cdot \mathbf{u})$.

[4] ℓ rows and m columns

Formally speaking, switching between treating vectors as $m \times 1$ matrices and back is a weird operation that you have to take care with. In practice, however, it works fine.

Weirdly, however, `numpy` does draw a strong distinction between objects of dimension `(n,)` (a vector) and `(n,1)` (a matrix) – sometimes internal functions will work fine when given either; sometimes you will have to use a `reshape` command to explicitly switch between them. It can be a bit weird!

3.4 Affine Transformations

An important thing to note is that the following is **not** possible:

$$M\mathbf{x} = \mathbf{x} + \mathbf{m} \tag{3.22}$$

You can, naturally, find an M for each $\mathbf{x}$ which performs this operation, but there is no *general* matrix which always performs this operation: you cannot write M without knowing $\mathbf{x}$. This exposes the slightly weird fact that *vector addition is not a linear operation.*

This might seem weird: addition is a linear operation, surely? However, if we return to our definition of a linear operator, we find that they must be distributive:

$$M\left(\mathbf{x} + a\mathbf{y}\right) = M\mathbf{x} + aM\mathbf{y} \tag{3.23}$$

If we tried this with our 'addition operator' we would find[5]:

$$\begin{aligned} M\mathbf{x} + aM\mathbf{y} &= (\mathbf{x} + \mathbf{m}) + a\left(\mathbf{y} + \mathbf{m}\right) \\ &= M\left(\mathbf{x} + a\mathbf{y}\right) + a\mathbf{y} \end{aligned} \tag{3.24}$$

Given that vector addition represents the *translation* operation, what are we to do? Does this mean we cannot treat addition using our lovely formalism of linear algebra?

Technically speaking, yes, that is what it means because – as we have just shown – translation and addition are **non-linear operations**, and so are beyond the scope of *linear* algebra.

Luckily, this is not the end of the road.

An **Affine Operator** is one which can be written as a combination of a linear operator and a translation operator:

$$\hat{\mathcal{M}}\mathbf{x} = \hat{M}\mathbf{x} + \mathbf{m} \tag{3.25}$$

That is, Affine Operators are the group of linear operators, with the non-linear translation operators tacked onto the end: Affine Transformations are essentially "Linear++" transforms and, although they formally sit outside the realm of Linear Algebra, the majority of the techniques still apply. Of particular interest is the composition of two general affine operators $\hat{\mathcal{M}}, \hat{\mathcal{N}}$:

$$\begin{aligned} \hat{\mathcal{M}}\hat{\mathcal{N}}\mathbf{x} &= \hat{\mathcal{M}}\left(\hat{N}\mathbf{x} + \mathbf{n}\right) \\ &= \hat{M}\hat{N}\mathbf{x} + \left(\hat{M}\mathbf{n} + \mathbf{m}\right) \\ &= \hat{P}\mathbf{x} + \mathbf{p} \\ &= \hat{\mathcal{P}}\mathbf{x} \end{aligned} \tag{3.26}$$

Which, in words, shows that **any combination of affine operators can always be written as a single affine operator**; that is, affine operators form a closed group.

This is important because the perceptron algorithm (and thus, the internal workings of nodes in a feedforward network) are perform an affine transformation on their input vector – this has important ramifications which we will discuss in the workshop.

[5] In words: there is a vast difference (in terms of sweets given out) between giving a box of sweets to a classroom, and giving a box of sweets *to each person in the class*

3.4.1 Addition vs. Augmenting

There are[6] two different ways that one may represent an Affine Operator – one of them as already been demonstrated:

$$\hat{\mathcal{M}}\mathbf{x} = \hat{M}\mathbf{x} + \mathbf{m} \tag{3.27}$$

This makes it eminently clear that our operation is a composition of a linear operator, and an addition operation.

However, in some circumstances (....such as when you're trying to elegantly write an optimisation routine for a backpropagating neural network, to give you a spoiler alert) this can be a little bit awkward to deal with, because you have two different 'types' of object to deal with.

In such cases it can be easier to treat the Affine Operator as a linear operator *on an augmented space*.

Consider the following constructions (where we have projected into using matrices in place of operators):

$$\tilde{\mathbf{x}} = \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix} \quad \tilde{M} = \begin{bmatrix} \mathbf{m} & M \end{bmatrix} = \begin{bmatrix} \begin{pmatrix} m_1 \\ m_2 \\ m_3 \\ \vdots \end{pmatrix} & M \end{bmatrix} \tag{3.28}$$

That is, $\tilde{\mathbf{x}}$ is just $\mathbf{x}$ with a '1' inserted above the first element, and $\tilde{M}$ is just M with the vector $\mathbf{m}$ inserted as the first column.

$\tilde{\mathbf{x}}$ is therefore one dimension larger than $\mathbf{x}$, and $\tilde{M}$ similarly has 1 more column than M (but the same number of rows).

When we multiply these out:

$$\begin{aligned} \tilde{M}\tilde{\mathbf{x}} &= \begin{pmatrix} m_1 + \sum_i M_{1i}x_i \\ m_2 + \sum_i M_{2i}x_i \\ m_3 + \sum_i M_{3i}x_i \\ \vdots \end{pmatrix} \\ &= \mathbf{m} + M\mathbf{x} \end{aligned} \tag{3.29}$$

This has the advantage that we do not need to do anything to treat the addition differently from the multiplication: the rules of matrix multiplication handle that for us automatically.

If (as we will be later), we are interested only in finding a suitable transform $\hat{\mathcal{M}}$ which simply works 'well enough' and has no intrinsic meaning supplied to it, then it is sufficient to treat it solely as an unknown matrix multiplication on an augmented input vector: we don't have to worry about the internal structure of $\tilde{M}$ – merely trust that the neural network will sort it out for us.

An Example

Consider the following components of an affine transformation which rotates an (x, y) vector by 90° around the x axis, and then translates it upwards by 2 units:

$$R = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \qquad \mathbf{r} = \begin{pmatrix} 0 \\ 2 \end{pmatrix} \tag{3.30}$$

[6] at least!

Applying this to the vector $\mathbf{x} = (2, 4)$ we see:

$$\begin{aligned} R\mathbf{x} + \mathbf{r} &= \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} 2 \\ 4 \end{pmatrix} + \begin{pmatrix} 0 \\ 2 \end{pmatrix} \\ &= \begin{pmatrix} 4 \\ -2 \end{pmatrix} + \begin{pmatrix} 0 \\ 2 \end{pmatrix} \\ &= \begin{pmatrix} 4 \\ 0 \end{pmatrix} \end{aligned} \tag{3.31}$$

If instead we augmented the matrix to become:

$$\tilde{R} = \begin{pmatrix} 0 & 0 & 1 \\ 2 & -1 & 0 \end{pmatrix} \quad \tilde{\mathbf{x}} = \begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix} \tag{3.32}$$

This gives the result:

$$\tilde{R}\tilde{\mathbf{x}} = \begin{pmatrix} 4 \\ 0 \end{pmatrix} \tag{3.33}$$

As expected, this is the same final result, but rather than two separate operations, we did one (larger) operation.

Chapter 4

Multivariable & Vector Calculus

The above section made the assumption that f was only ever a function of one variable, a single scalar. That is clearly a limitation; we are going to want to do calculus with many dozens, if not thousands of parameters!

What if, let's say, we go truly nuts, and have:

$$f(x, y) = x^2 + 3xy \tag{4.1}$$

How can we possibly deal with such things?

The solution is: just do what you did before with x, but pretend y is a constant. And then reverse it; pretend x is a constant and take the derivative with respect to y:

$$\left(\frac{\partial f}{\partial x}\right)_y = 2x + 3y \tag{4.2}$$

$$\left(\frac{\partial f}{\partial y}\right)_x = 3x \tag{4.3}$$

We can see that, instead of our previous notation of $\frac{\mathrm{d}f}{\mathrm{d}x}$, we have switched to this new symbol, ∂. That is because this is no longer a full derivative; it is a *partial derivative*. The brackets and the subscript are there to remind us 'what we kept constant' – formally speaking, we should always include those, because a partial derivative is only meaningful if accompanied by information about what was 'partial' about it. In practice, however, it is almost always 'obvious', and so we can omit this notation – you only really need to keep them around when you're dealing with a subset of a larger set of variables which are all interrelated in some way. The classic case of 'you will go insane if you don't use the full notation' is Thermodynamics, where pressure (P), temperature (T), entropy (S), free energy (E) and are all interrelated – so $\left(\frac{\partial E}{\partial T}\right)_P$ is a completely different beast than $\left(\frac{\partial E}{\partial T}\right)_S$.

For what we're dealing with, however, we can ignore this; it suffices merely to note that, if you have a function $f(x_1, x_2, x_3, ...)$ where x_i are all free, independent variables, then taking partial derivatives is equivalent to simply 'holding the others constant' and taking the derivative as you would in single variable calculus.

It really is that simple (for now!)

What's That About Vectors?

Because I don't have time to go into the full ins and outs of the difference between multivariable calculus, and calculus of a single variable where that variable happens to be a multidimensional vector, I am playing slightly fast and loose with the terminology and formalism.

In short, I am making the assumption that:

$$f(x,y,z) = f\left(\begin{pmatrix} x \\ y \\ z \end{pmatrix}\right)$$

This is not always the case; and this mapping is not strictly 1:1. For instance, strictly speaking, Multivariable Calculus does not require the similar relationships under addition and linear transformations that Vector Calculus implies – there are many pathological examples you can concoct where the difference between $f(x,y,z)$ and $f\left(\begin{pmatrix} x & y & z \end{pmatrix}^\intercal\right)$ actually matters.

This is not one of those times.

Broadly speaking, we can use vector calculus and multivariable calculus as synonymous and interchangeable – a simple matter of 'packaging up' and notation.

Almost all of the other rules still apply; so long as you remember to hold the other variables constant whilst you do so.

4.1 Derivatives of Vectors

The simplest case for vector calculus is where a vector $\mathbf{v}$ is a function of a single variable; the classic case is position as a function of time, $\mathbf{x}(t)$. When we differentiate this with respect to time, the result is simply:

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x}(t) = \frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \\ \vdots \end{pmatrix} = \begin{pmatrix} \frac{\mathrm{d}x_1}{\mathrm{d}t} \\ \frac{\mathrm{d}x_2}{\mathrm{d}t} \\ \frac{\mathrm{d}x_3}{\mathrm{d}t} \\ \frac{\mathrm{d}x_4}{\mathrm{d}t} \\ \vdots \end{pmatrix} \tag{4.4}$$

Note that, in this case, because there is only one variable to take the derivative of, I have switched back to using d, rather than ∂. We generally won't need this for Machine Learning Applications.

4.2 The Vector Derivative

Let us now assume that we have a function $f(\mathbf{x})$, where $\mathbf{x}$ is some list of N parameters. We have already seen how we might compute, for instance $\frac{\partial f}{\partial x_1}$ (we hold all other x_i constant, and take the derivative as normal). However, you will often see things like this:

$$\frac{\partial f}{\partial \mathbf{x}} \quad \text{or} \quad \nabla f \tag{4.5}$$

These two notations should be understood to mean:

$$\frac{\partial}{\partial \mathbf{x}} = \nabla = \begin{pmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} \\ \vdots \end{pmatrix} \tag{4.6}$$

In short, this is a shorthand way of saying 'do each individual partial derivative, and then package the results up into a vector'. For instance, if we have:

$$f(\mathbf{x}) = f(x,y,z) = x^2\sin(z) - xyz \tag{4.7}$$

Then:

$$\frac{\partial f}{\partial \mathbf{x}} = \begin{pmatrix} 2x\sin(z) - yz \\ -xz \\ x^2\cos(z) - xy \end{pmatrix} \tag{4.8}$$

Just as the single-dimensional derivative told us the gradient of the curve, this is a generalisation of the gradient to higher dimensions; not only does it's magnitude tell us the steepness of the function, but it points in the direction of the greatest increase. Following this gradient will, just as it did in the single-variable case, take us towards the nearest local peak; whilst walking in the opposite direction will take us to the local minima.

4.3 The Chain Rule

The multivariate chain rule is of critical importance for our work in machine learning. Let us first consider the simplest case; a scalar-valued function which is a function of a vector, but where the vector is itself a simple function of one variable.

For instance:

$$y = f(\mathbf{x}(t)) \tag{4.9}$$

Here $\mathbf{x}$ might be the position in 3D space, and so f requires three dimensions – if we specify a path which is followed as a function of time, then we actually only have to specify what the current time is to know y.

What, therefore, is $\frac{\mathrm{d}y}{\mathrm{d}t}$?

To find this out, we use the *total derivative* of f. When we write the total derivative of a function, you get:

$$\begin{aligned} \mathrm{d}f(\mathbf{x}) &= \frac{\partial f}{\partial x_1}\mathrm{d}x_1 + \frac{\partial f}{\partial x_2}\mathrm{d}x_2 + \dots \\ &= \sum_i \frac{\partial f}{\partial x_i}\mathrm{d}x_i \\ &= \frac{\partial f}{\partial \mathbf{x}} \cdot \mathrm{d}\mathbf{x} \end{aligned} \tag{4.10}$$

We can then do the thing which makes mathematicians cry, and 'divide by $\mathrm{d}t$', to find:

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\partial f}{\partial \mathbf{x}} \cdot \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} \tag{4.11}$$

These leads us to the final case; where $\mathbf{x}$ is not a function of a nice scalar, but of another vector:

$$y = f(\mathbf{x}(\mathbf{u})) \tag{4.12}$$

Surprisingly, however, this is surprisingly easy (because we've done the hard work of packaging things into nice notation)

$$\frac{\partial f}{\partial \mathbf{u}} = \frac{\partial \mathbf{x}}{\partial \mathbf{u}}^{\mathsf{T}} \frac{\partial f}{\partial \mathbf{x}} \tag{4.13}$$

Where

$$\frac{\partial \mathbf{x}}{\partial \mathbf{u}} = J_{\mathbf{x}} = \begin{pmatrix} \frac{\partial x_1}{\partial u_1} & \frac{\partial x_1}{\partial u_2} & \dots & \frac{\partial x_1}{\partial u_n} \\ \frac{\partial x_2}{\partial u_1} & \dots & & \\ \vdots & & & \\ \frac{\partial x_m}{\partial u_1} & \dots & & \frac{\partial x_m}{\partial u_n} \end{pmatrix} \tag{4.14}$$

Why is this written in such a way that we need to Transpose it? Why not just define it already-transposed? The answer is that this entity – the Jacobian matrix – appears in lots of

different places, most of which where the above definition is most convenient, and so the Chain Rule gets stuck with a weird transpose.

Naturally things can get *way* more complicated here, but the general gist is that multivariable calculus isn't **that** much different than normal calculus.

Chapter 5

Numerical Optimisation

The above analysis of both single variable and multivariable calculus naturally lends itself to asking, therefore, how we may identify the maxima (or minima) of a function if analytically solving the derivative is impossible.

This is a field which is simultaneously both cutting edge, and ludicrously simple: some of the biggest and most recent 'leaps forward' are trivial modifications of methods which have existed for centuries; that is to say, they are all *first order* methods, which require only the ability to compute the gradient. In fact, many of these methods require *only* that you be able to compute the gradient, and being able to compute the function is largely irrelevant.

For the following analysis, we shall assume that we are seeking the *minimum* of the function; the *maximum* can be found simply by inverting the sign on the step.

5.1 Newton's Descent

Newton's Descent finds the optima of a function by asking the very simple question: *how do you navigate to the bottom of a hill?* The answer being, of course, **you walk downhill.** Provided you don't get caught in any wells or localised dips (i.e. local minima), you can always find the lowest point in a valley by walking in the direction which is steepest downhill.

After each step, you recompute the steepest direction, and take another step; recompute, and repeat.

This, in essence, is the underpinning of all first-order methods.

Given we start at a position $\mathbf{x}_0$, the position after each step is then given by:

$$\mathbf{x}_n = \mathbf{x}_{n-1} - \alpha \mathbf{d}_{n-1} \tag{5.1}$$

Where $\mathbf{d}_n$ is the direction vector of the gradient:

$$\mathbf{d}_n = \frac{1}{\left|\frac{\partial f}{\partial \mathbf{x}}\right|} \frac{\partial f}{\partial \mathbf{x}} \tag{5.2}$$

Why is the step direction negative? The gradient points *up* the hill, and we want to walk *downhill*, in order to minimize. If we were maximising, we would use a + sign, and walk uphill.

The variable α is the step-size; it denotes how far you will move each time you compute the gradient. Naturally this has a problem because, if the optimum is at $\mathbf{x} = 0$, $\alpha = 1$ and you are at $\mathbf{x} = 0.5$, the gradient will tell you to to go backwards; to $\mathbf{x} = -0.5$, at which point the gradient will tell you to go forwards...to 0.5.

This will repeat, ad infinitum; you will never, ever reach precisely the optimum; but with a suitable choice of α you can get close enough for all practical purposes.

5.2 Line Search

Naïve Newton's descent keeps α fixed, requiring human intervention to fine tune the optimisation – if α is set too large, then the optimisation will only get 'near to' the optimum, whilst if α is too small, convergence to the optimum will take significant computer power simply recomputing gradients which tell you to move in the same direction you were already moving!

Line Search algorithms encompass those which try to automatically determine the suitable α each step. A simple Line Search algorithm would go something along the lines of:

1. At $\mathbf{x}_n$, compute $\mathbf{d}_n$ and $y_n = f(\mathbf{x}_n)$.
2. Let $\hat{\mathbf{x}} = \mathbf{x}_n - \alpha\mathbf{d}_n$ and $\ell = 1$
3. Compute $\hat{y} = f(\hat{\mathbf{x}})$.
   - If $\hat{y} < y_n$, then set $\mathbf{x}_{n+1} = \hat{\mathbf{x}}$
   - Let $\hat{\mathbf{x}} = \mathbf{x}_n - \frac{\alpha}{2^\ell}\mathbf{d}_n$ and $\ell \to \ell + 1$. Then go to step 3.

In this algorithm, rather than blindly stepping in the direction $\mathbf{d}$, we first check that this actually reduces the value of y – if it does, we move our usual step of length α. If, however, it does not improve the score, we do not make the step; we instead decrease α by a factor of 2, and recompute the score at this new proposed position: we repeat this until either a set number of iterations is reached (and we move anyway), or we find a length scale that does decrease the score.

This allows us to keep α large for moving fast near the beginning of the optimisation. However, when we get close and a large α would be detrimental, the line search will ensure that we don't jump over the optimum. The cost of this is that we must compute $f(\mathbf{x})$, potentially multiple times for each step. If f is a costly function, then this will dominate our optimisation times, making the routine take much longer.

Other line search and similar methods exist: the BFGS algorithm, for instance, attempts to estimate the correct value of α by approximating the Hessian (the second derivative matrix); essentially modelling the gradient as a quadratic, rather than straight line.

All of these methods, however, take additional computation time which scales (usually) as n^2 or n^3 in the number of parameters; when you have many parameters to optimise and your function is costly to compute, these costs often mean that it is simply easier to use a simpler, less clever method, but run it for longer! This is often the case in Machine Learning applications, where the number of parameters exceeds hundreds to thousands; using the dumb-but-quick methods are often significantly more approachable, even though their mathematical accuracy and elegance would imply the opposite.

5.3 Momentum & ADAM

One of the problems that the basic Newton Method faces is that it will readily get caught in local optima: the gradient has no way of informing you "hey if you move $x \to x + 10$" there's a way deeper valley than the one you're currently in!" The gradient, after all, merely points towards (or away from) the nearest local optima.

One way to overcome this, and also better estimate α when the optimiser is bouncing around trying to find things, is to use *momentum*. In the analogy of 'walking down a hill', this becomes more like 'rolling a ball down a hill' – it will generally roll downhill, but if it gets up enough speed, it will happily plow over smaller valleys.

In this case, we have an additional parameter; the *memory* of the optimiser,β. A simple momentum optimiser looks like, with $\mathbf{d}_0 = \mathbf{0}$:

1. At $\mathbf{x}_n$, compute $\mathbf{d}_n$.
2. $\mathbf{m}_n = \beta\mathbf{m}_{n-1} + (1-\beta)\mathbf{d}_n$
3. Let $\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha\mathbf{m}_n$

If $\beta = 0$, we can see that we recover the original Newton algorithm, otherwise we see that $\mathbf{m}$ is an exponentially weighted mean of the previously computed directions: if they all point in the same direction, then $\mathbf{m} \approx \mathbf{d}$, and it will move as expected with a step size of α. If the gradient has recently switched directions (or gone to zero), then it follows that $|\mathbf{m}_n| < 1$, and so the step size will be (approximately) in the direction of the old gradients, but with a smaller step size. If the gradient keeps pointing in this new direction, it will veer back around and forget its old route – if the gradient change was only a momentary fluctuation – a bump in the road – then it will keep on going. This also has the effect of damping out oscillations; if the optimiser is bouncing backwards and forwards either sides of the true optimum, the momentum term will ensure these 'bounces' get smaller, and so the optimiser will converge on the true optimum.

Empirically, on the kinds of problems that we are interested in solving, this algorithm often outperforms the naive Newton's method, even if it might initially seem perverse to deliberately ignore changes in the gradient!

The ADAM optimiser (of Kingma & Ba, 2012) is an update to this formula – rather than simply keeping a momentum of the *direction*, they attempt to simultaneously normalise the gradient and the second moment of the gradient. The result is that the model keeps a 'per parameter' memory of it's prior performance, rather than risking having some parameters lost in the 'direction gradient'; this can happen if your model's parameters have vastly different scales: if one changes on the scale of 10^{-1}, whilst the other changes on the scale of 10^3, the normalisation of the gradient can cause the smaller scale parameter to get lost.

1. At $\mathbf{x}_n$, compute the gradient, $\mathbf{g}_n = \frac{\mathrm{d}f}{\mathrm{d}\mathbf{x}}$.
2. Let:
   - $\mathbf{m}_n = \beta_1\mathbf{m}_{n-1} + (1-\beta_1)\mathbf{g}_n$
   - $\mathbf{v}_n = \beta_2\mathbf{v}_{n-1} + (1-\beta_2)\mathbf{g}_n \otimes \mathbf{g}_n$
   - $c_1 = \left(1.0 - \beta_1^{n+1}\right)^{-1}$
   - $c_2 = \left(1.0 - \beta_2^{n+1}\right)^{-1}$
3. Let the step be:
$$\mathbf{s}_n = \alpha c_1 \mathbf{m}_n \oslash \sqrt{c_2\mathbf{v}_n + \epsilon}$$
4. Let $\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{s}_n$

Here $\oslash$ is the Hadamard (element-wise) divisor, and the $\sqrt{\mathbf{v}}$ is assumed to the be vector where each element is the square root of the corresponding element of $\mathbf{v}$. The purpose of the c_i terms is to un-bias the optimiser away from zero in the early stages of optimisation; without these the model is 'slow off the mark' and can get biased. These corrections prevent that from altering the final value. The $\epsilon \approx 10^{-8}$ term is simply to stop division by zero should the gradient get close to the optimum, and hence $[\mathbf{v}]_i \ll 1$.

This optimiser shows superior performance over many 'more intelligent' routines; it is rather remarkable how this simple modification of Newton's algorithm from the 18th century is at the forefront of many modern ML applications!

Part II

Interesting Discussions

Chapter 6

Introduction

This section of the notes is intended to provide some deeper understanding for those who are interested, and perhaps connect some of the dots between the more abstract mathematics which was introduces in Part I, and the more familiar mathematics you might be familiar with.

You should only read this section if you are already familiar with the background mathematics, and are interested to learn a bit more.

Chapter 7

Polynomial Derivatives

In the above analysis, we jumped straight from the formal definition of the derivative, straight to the familiar polynomial derivative rule:

$$\frac{\mathrm{d}}{\mathrm{d}x}x^n = nx^{-1} \tag{7.1}$$

How does one follow from the other? This is actually surprisingly complex. It is easy to show that it is true when n is an integer since – for these cases – we can use the Binomial theorem:

$$(x+\delta)^n = x^n + nx^{n-1}\delta + \frac{n(n-1)}{2}x^{n-2}\delta^2 + a\delta^3 + b\delta^4 + \dots \tag{7.2}$$

Where $a, b, c\dots$ are functions only of n and x. If we insert this into the formal definition of the derivative, we find that:

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}x}x^n &= \lim_{\delta\to 0}\left(\frac{(x+\delta)^n - x^n}{\delta}\right) \\
&= \lim_{\delta\to 0}\left(\frac{\left[x^n + nx^{n-1}\delta + \frac{n(n-1)}{2}x^{n-2}\delta^2 + a\delta^3 + b\delta^4 + \dots\right] - x^n}{\delta}\right) \\
&= \lim_{\delta\to 0}\left(nx^{n-1} + \frac{n(n-1)}{2}x^{n-2}\delta + a\delta^2 + b\delta^3 + \dots\right)
\end{aligned} \tag{7.3}$$

We see that all of the terms except the first are multiplied by a δ, and so when we set $\delta = 0$, they will vanish. The first term, however, remains:

$$\frac{\mathrm{d}}{\mathrm{d}x}x^n = nx^{n-1} \tag{7.4}$$

This, however, only works when n is an integer, since this is the only case where the Binomial expansion holds (at least, until we can derive Taylor expansions, but we need polynomial derivatives to do that!)

This gets down to a bigger question: what does it mean to say x^n when n is not an integer? If n is a rational number $n = \frac{p}{q}$, you can say that $x^n = \sqrt[q]{x^p}$; but what about x^π, an irrational number? How can you 'multiply x by itself π times'? It turns out the easiest definition is simply the exponential function:

$$x^n = \exp\left(n\log(x)\right) \tag{7.5}$$

Using the chain rule and the knowledge that $\frac{\mathrm{d}}{\mathrm{d}x}\exp(x) = \exp(x)$ and $\frac{\mathrm{d}}{\mathrm{d}x}\log(x) = x^{-1}$, we recover our familiar polynomial derivative immediately.

Chapter 8

The Real Definition of a Vector

The real definition of a vector $\mathbf{v}$ is that it must be a member of a Vector Space, V. In order for V to be a vector space, all elements of V must obey the following rules:

1. It must have a concept of 'generalised addition', such that for members, $\mathbf{v}$ and $\mathbf{u}$ the 'generalised sum' $\mathbf{v} + \mathbf{u} = \mathbf{w}$, and $\mathbf{w}$ is a member of the space.
2. It must have a concept of 'scalar multiplication', such that for $\mathbf{v}$ a member of the space and α a real or complex number, $\mathbf{w} = \alpha \times \mathbf{v}$ is a member of the space.
3. Addition is associative:
$$\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w}$$
4. Addition is commutative
$$\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$$
5. There is an identity element for addition; a vector $\mathbf{0}$ which does nothing when added:
$$\mathbf{v} + \mathbf{0} = \mathbf{v}$$
6. Each element has an additive inverse $(-\mathbf{v})$ which is in V, such that:
$$\mathbf{v} + (-\mathbf{v}) = \mathbf{0}$$
7. Scalar multiplication has an identity, 1, which when multiplied does nothing:
$$1 \times \mathbf{v} = \mathbf{v}$$
8. Scalar multiplication is distributive with respect to vector addition
$$a \times (\mathbf{v} + \mathbf{w}) = a\mathbf{b} + a\mathbf{w}$$
9. Scalar multiplication is distributive with respect to field addition
$$(a + b)\mathbf{v} = a\mathbf{v} + b\mathbf{v}$$

Some of these seem supremely obvious, but the point is that *any* object, and *any* operation you care to call 'addition' can therefore be used to form a vector space.

8.1 Some Weird Examples

8.1.1 The Positive Numbers

It might initially seem that, by definition, the positive numbers cannot be a Vector space. After all, the additive inverse of 2 is -2 which is not in the space, and so we fail the additive inverse condition.

This, however, ignores the fact that we can redefine addition and multiplication as we like! Let us define the 'positive addition' to be:

$$\mathbf{a} \oplus \mathbf{b} = a \times b$$

The scalar multiplication is then:

$$n\mathbf{a} = a^n$$

We can see that so long as $a, b > 0$, then $a \times b > 0$, and $a^n > 0$ for any n. The additive inverse of 2 is, weirdly, $\frac{1}{2}$, and the zero vector is actually 1.

These therefore satisfy the conditions for our generalised addition and multiplication. It is possible to work through the above list and show that it does indeed hold true for all of the relationships; and so the positive numbers can be a Vector Space.

Chapter 9

Vectors And Angles

The existence of the inner product is where the definition 'a vector is a quantity with direction and magnitude' comes from since, thanks to the Cauchy-Schwarz Inequality:

$$\langle a, b\rangle^2 \leq \langle a, a\rangle \langle b, b\rangle \tag{9.1}$$

In fact, we can strengthen this further if we forbid $\mathbf{a}$ from being a scalar multiple of $\mathbf{b}$:

$$\langle a, b\rangle^2 < \langle a, a\rangle \langle b, b\rangle \text{ iff } \mathbf{a} \neq x\mathbf{b} \tag{9.2}$$

That is, $\langle a, b\rangle = \sqrt{\langle a, a\rangle \langle b, b\rangle}$ is only possible if $\mathbf{a}$ is just a rescaling of $\mathbf{b}$. I've used the general $\langle a, b\rangle$ form to emphasise that this is a general property of a 'true inner product', not just of the dot product.

It is this property that lets us define the **length** of a vector as:

$$L(\mathbf{a}) = \sqrt{\langle a, a\rangle} \tag{9.3}$$

From the Cauchy-Schwarz Inequality, we than have:

$$\langle a, b\rangle^2 \leq L((a))L(\mathbf{b}) \qquad \Longrightarrow \qquad -1 \leq \frac{\langle a, b\rangle}{\sqrt{L(\mathbf{a})L(\mathbf{b})}} \leq +1 \tag{9.4}$$

An alternative way to write this:

$$\frac{\langle a, b\rangle}{\sqrt{L(\mathbf{a})L(\mathbf{b})}} = f(\mathbf{a}, \mathbf{b}) \qquad -1 \leq f(\mathbf{a}, \mathbf{b}) \leq 1 \tag{9.5}$$

This mysterious f is:

- A function of the two vectors
- Equal to 1 when the vectors are equal
- Always between -1 and 1, no matter what values $\mathbf{a}$ and $\mathbf{b}$ are

This function looks an awful lot like a cosine! We therefore define the angle between two vectors as:

$$\theta(\mathbf{a}, \mathbf{b}) = \arccos\left(\frac{\langle a, b\rangle}{\sqrt{L(\mathbf{a})L(\mathbf{b})}}\right) \tag{9.6}$$

This is the abstract definition of the angle between two vectors – and it applies even in some wacky cases.

Consider the vector space of real functions, so $\mathbf{a} = x$ and $\mathbf{b} = 2x - 3x^3$. We can define an inner product as:

$$\langle a, b\rangle = \int_{-\infty}^{\infty} \exp(-x^2) a(x) b(x) \mathrm{d}x \tag{9.7}$$

Then:

$$\begin{aligned}
L(\mathbf{a}) &= \sqrt{\int_{-\infty}^{\infty} \exp(-x^2) x^2 \mathrm{d}x} &&= \frac{\sqrt[4]{4\pi}}{2} \\
L(\mathbf{b}) &= \sqrt{\int_{-\infty}^{\infty} \exp(-x^2)(2x - 3x^3)^2 \mathrm{d}x} &&= \frac{\sqrt[4]{24964\pi}}{4} \\
\langle a, b\rangle &= \int_{-\infty}^{\infty} \exp(-x^2) x(2x - 3x^3) \mathrm{d}x &&= -\frac{5\sqrt{\pi}}{4} \\
\theta &= \arccos\left(-\frac{5}{\sqrt{79}}\right) &&= 124.2^\circ
\end{aligned}$$

Thus we have meaningfully defined an angle between two random functions! This isn't relevant for what we're doing, it's just fun and weird.

Chapter 10

Matrix Multiplication

In section 3.3, we made the jump from 'Linear Operators have certain behaviours' to 'matrices are grids which multiply like this' with a wave of the hand and a 'trust me I know what I'm doing'. Why is this connection true? Why does matrix multiplication work the way it does? How and why is this 'the way things are done'?

This section aims to derive the matrix multiplication rules from nothing more than the behaviour of linear operators.

We first consider that, when we write our vectors in a nice column vector, what we're doing (behind the scenes) is writing the vector as a **sum of basis vectors**:

$$\mathbf{v} = \begin{pmatrix} a \\ b \\ c \end{pmatrix} = a\hat{e}_1 + b\hat{e}_2 + c\hat{e}_3 \tag{10.1}$$

The entry in each element of the vector means 'I have a amount of $\hat{e}_1$, this much of $\hat{e}_2$, and this much of $\hat{e}_3$'. Generally, we have taken care to write our basis vectors in a linearly independent fashion; that is, such that saying 'I have 3 lots of unit $\hat{e}_1$', tells me nothing about how much $\hat{e}_3$ I have.

In a 'normal' 3D vector, we'd be familiar that $[3, 2, 1]^\intercal$ would mean '3 units of x distance, 2 of y and 1 of z; but if (for some reason), we decided to make our vector also be $[3, 2, 1, d]$ where d was the distance from some point of interest, we've violated this assumption, since d could already be computed from the position, so it is not linearly independent.

Since this is a horrible state of affairs, we will assume from now on that our basis is indeed independent in the above fashion.

If this is true, then it is also true that $\hat{e}_1 \cdot \hat{e}_2 = 0$ and $\hat{e}_3 \cdot \hat{e}_1 = 0$; the basis vectors are necessarily at right angles to each other. This is important! If you do not have an orthogonal basis set, the mathematics becomes horribly complicated. Most of quantum physics is about searching for a nice orthogonal basis set, and abusing that for all it's worth.

We can therefore always write a general, N dimensional vector in terms of N orthogonal bases (this is, in fact, a good definition of 'dimension'; the minimum number of vectors in a basis set):

$$\mathbf{v} = \sum_i a_i \hat{e}_i = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \end{pmatrix} \tag{10.2}$$

We can also pull a little trick to rewrite our vector, since $\hat{e}_i \cdot \hat{e}_j = 0$ if $i \neq j$, but 1 if $i = j$,

so:

$$\mathbf{v} \cdot \hat{e}_i = a_i \qquad \Longrightarrow \qquad \mathbf{v} = \sum_i (\mathbf{v} \cdot \hat{e}_i)\hat{e}_i \tag{10.3}$$

One final trick, before we're ready to put the pieces together. We now consider the 'identity operator'. This operator is the operator which does nothing - it multiplies a vector by 1, leaving behind the original vector:

$$\hat{I}\mathbf{v} = \mathbf{v} \tag{10.4}$$

We can write I in a very strange way:

$$\hat{I} = \sum_i \hat{e}_i \, (\hat{e}_i \cdot \circ) \tag{10.5}$$

What this means is, when we multiply I by a vector, we insert the vector wherever we see the $\circ$, and therefore into the dot product:

$$\hat{I}\mathbf{v} = \sum_i \hat{e}_i (\hat{e}_i \cdot \mathbf{v}) \tag{10.6}$$

By comparison with Eq. 10.3, however, this is just exactly equal to $\mathbf{v}$. This unusual way of writing the identity operator will help us clarify things, since we can always multiply by the identity, and leave everything unchanged.

We now consider the operator $\hat{M}$, acting on a vector $\mathbf{v}$. Since matrices are linear operators:

$$\hat{M}\mathbf{v} = \hat{M}\left(\sum_i (\hat{e}_i \cdot \mathbf{v})\hat{e}_i\right) \tag{10.7}$$

$$= \sum_i (\hat{e}_i \cdot \mathbf{v})\hat{M}\hat{e}_i \tag{10.8}$$

Where we remember that $(\hat{e}_i \cdot \mathbf{v}) = a_i$ is just the element in the column vector, written in a strange form. We can now multiply by the Identity operator (but have to sum over j since i is already taken!)

$$\hat{I}\hat{M}\mathbf{v} = \hat{M}\mathbf{v} \tag{10.9}$$

$$= \sum_j \hat{e}_i \left(\hat{e}_j \cdot \left[\sum_i (\hat{e}_i \cdot \mathbf{v})\hat{M}\hat{e}_i\right]\right) \tag{10.10}$$

However, this is once again a dot product (and the dot product is still linear!), so:

$$\hat{M}\mathbf{v} = \sum_j \hat{e}_j \left(\hat{e}_i \cdot \left[\sum_i (\hat{e}_i \cdot \mathbf{v})\hat{M}\hat{e}_i\right]\right) \tag{10.11}$$

$$= \sum_j \hat{e}_j \sum_i (\hat{e}_i \cdot \mathbf{v})\hat{e}_j \cdot \hat{M}\hat{e}_i \tag{10.12}$$

However, we expect $\hat{M}\mathbf{v}$ to be an entirely new vector, $\mathbf{u}$, where $\mathbf{u}$ is defined as:

$$\hat{M}\mathbf{v} = \mathbf{u} = \sum_j b_j \hat{e}_j \tag{10.13}$$

By comparison, we see that:

$$b_j = \sum_i (\hat{e}_i \cdot \mathbf{v}) \left[\hat{e}_j \cdot \hat{M}\hat{e}_i\right] \tag{10.14}$$

$$= \sum_i a_i M_{ji} \tag{10.15}$$

Where $M_{ji} = \left[\hat{e}_j \cdot \hat{M}\hat{e}_i\right]$. If we write this out fully, we find:

$$\mathbf{u} = \begin{pmatrix} M_{1,1}a_1 + M_{1,2}a_2 + \dots \\ M_{2,1}a_1 + M_{2,2}a_2 + \dots \\ \vdots \\ M_{N,1}a_1 + M_{N,2} + \dots \end{pmatrix} \tag{10.16}$$

It is clear that $\hat{M}$ can be represented as a grid of numbers, a *matrix* M, such that:

$$M = \begin{pmatrix} M_{11} & M_{12} & M_{13} & \dots \\ M_{21} & M_{22} & M_{23} & \dots \\ & & \vdots & \\ M_{N1} & M_{N2} & M_{N3} & \dots \end{pmatrix} \tag{10.17}$$

If matrix multiplication is carried out using the rules described in 3.3, then we recover Eq. 10.16, and, by definition, all of the behaviours of linear operators. This is 'why matrix multiplication is the way that it is'; we have derived that general linear operators can be represented as a 2D grid of numbers, with a specific pattern of multiplication; using only the fact that they must obey certain linear operator rules.

Chapter 11

A Tiny, Tense Tangent on Tensors

"Tensors" are a huge thing in machine learning – Google have gone all in on this, producing `TensorFlow` and their TPUs, 'Tensor Processing Units' which are now being mass produced.

So, what is a Tensor?

The theoretical physicist inside of me wants to tell you a dirty secret; something *they* don't want you to know. **Tensors are lies!**

That is, what Machine Learning calls a tensor is in fact merely a series of data structures - an M-array. A common example is a colour image - each image is a $N \times N$ grid[1], but there are 3 channels (one each for RGB). We could just unwind this into a $3N^2$ long vector of values, but to preserve the relative information within, it is more useful to preserve it as a multidimensional data structure.

This is not a Tensor.

In mathematics, a Tensor is a generalisation of the concepts of Linear Operators into 'multilinear operators'. Whilst it is true that they can be represented as multilinear arrays, they are representations of high-dimensional operations, and, crucially, must obey certain transformation laws – it is these transformation laws which make them 'tensors' and not just 'a series of numbers'. A multidimensional array T is only a tensor (of type (p, q)) if it follows the following transformation law:

$$T^{i_1,i_2,\ldots,i_p}_{j_1,j_2,\ldots,j_q}\mathbf{f} = P^{i_1}_{i'_1}P^{i_2}_{i'_2}\ldots P^{i_p}_{i'_p}T^{i'_1,\ldots i'_p}_{j'_1,\ldots,j'_q}\mathbf{f}'(P^{-1})^{j'_1}_{j_1}\ldots P^{j'_q}_{j_q} \tag{11.1}$$

Which is, quite frankly, horrible. These transformation properties allow you to make general statements about the object, without having to specify a given projection or coordinate system – a very powerful ability which the Machine Learning definition of a tensor completely ignores and obfuscates. The Einstein Field Equations are Tensor Equations, because they hold true on arbitrary Riemannian manifolds:

$$R_{\mu\nu} - \frac{1}{2}Rg_{\mu\nu} + \Lambda g_{\mu\nu} = \frac{8\pi G}{c^4}T_{\mu\nu} \tag{11.2}$$

Is this the end of the world?

No, of course not. It doesn't really matter, so long as you remember that 'tensor' when being used in Machine Learning is being used due to a *structural* similarity to Tensors, rather than due to a *conceptual* similarity.

[1]Which some people call matrices – but you know better now, because a matrix is a Linear Operator, and an image is not an operator (at least, not until you redefine your Vector Space!)

Part III

Theory of Machine Learning

Introduction

This part of the notes will (hopefully!) contain the information presented during the workshop, for you to review afterwards at your leisure.

At the current time, however, it is empty. You may ignore this section.