First Principles of Machine Learning
Notes

Jack Fraser-Govil

October 2024

Contents

I Background Mathematics **4**

1 Introduction **5**

1.1 Some Basic Notation 5

2 Calculus **6**

2.1 Functions 6

2.2 Derivatives 7

2.3 Properties of the Derivative 10

2.4 Finding Extrema 12

3 Vectors & Matrices **13**

3.1 Vector Algebra 14

3.2 Linear Operators 15

3.3 Matrices 15

3.4 Affine Transformations 18

4 Multivariable & Vector Calculus **21**

4.1 Derivatives of Vectors 22

4.2 The Vector Derivative 22

4.3 The Chain Rule 23

5 Numerical Optimisation **25**

5.1 Newton's Descent 25

5.2 Line Search 26

5.3 Momentum & ADAM 26

II Interesting Discussions **28**

6 Introduction **29**

7 Polynomial Derivatives **30**

8 The Real Definition of a Vector **31**

8.1 Some Weird Examples 32

9 Vectors And Angles **33**

10 Matrix Multiplication **35**

11 A Tiny, Tense Tangent on Tensors **38**

III Theory of Machine Learning **39**

12 Perceptrons **41**

12.1 The Perceptron Workings 42

12.2 Linearity & Perceptrons 45

13 Multi-Layer Perceptrons **47**

13.1 The Naive Perceptron 47

13.2 The Feedforward Neural Network 50

13.3 Universal Approximation 53

14 Training an MLP **54**

14.1 Setting the Scene 54

14.2 Notational Clarification 55

14.3 The Cost Function 56

14.4 Backpropagation 56

15 Training Considerations **60**

15.1 Modified Cost Functions 60

15.2 Degradation & Convergence 61

15.3 Interpolation 62

15.4 Overfitting & Regularisation 63

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:

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

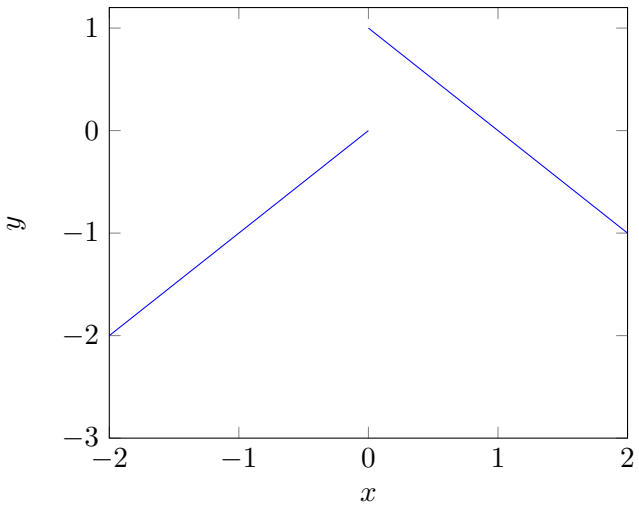
Here x and y can be anything; x might be a particular scent, and y could be the U2 album that smell most reminds my Dad of. In general, however, we will be most interested in cases where these are numerical quantities: either vectors or scalars.

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:

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

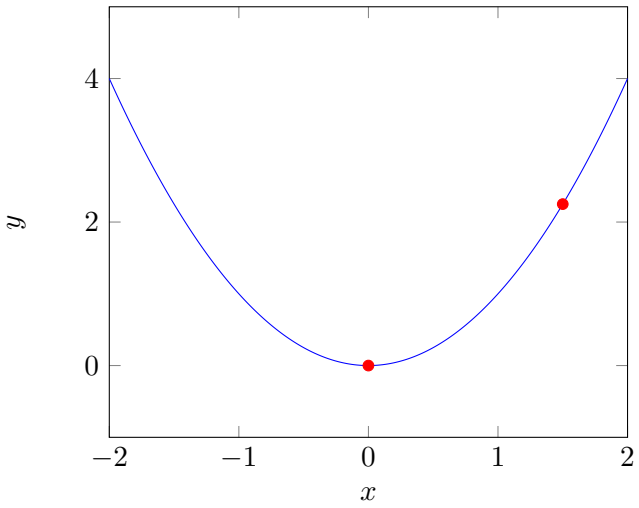
2.2 Derivatives

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.

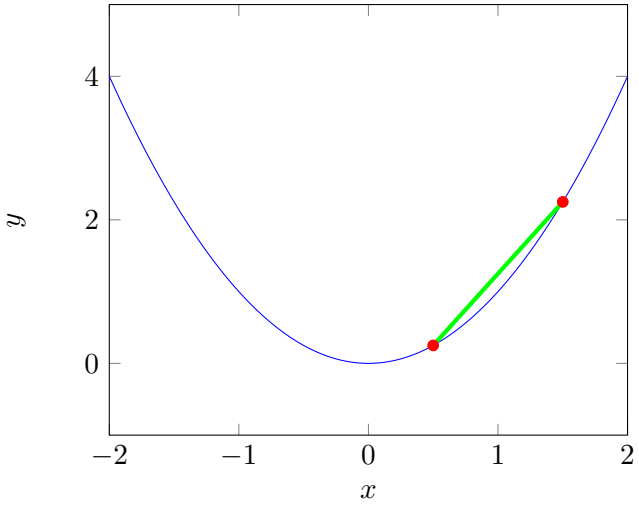


Figure 2.1: A Chord-approximation using $x_1 = 0.5$ and $x_2 = 1.5$

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:

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

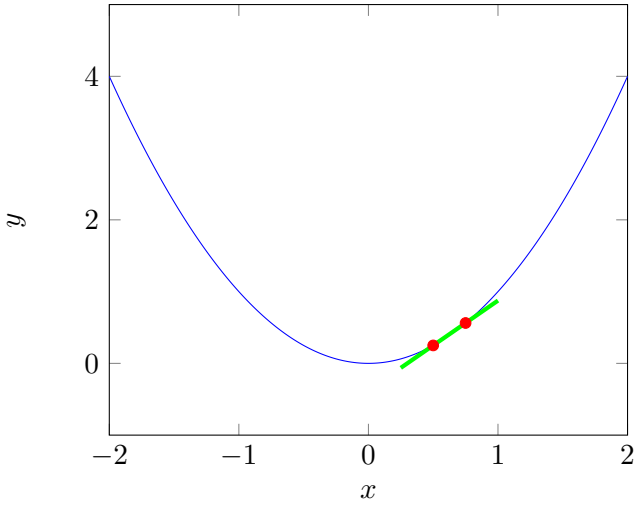


Figure 2.2: A Chord-approximation using $x_1 = 0.5$ and $x_2 = 0.75$

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]:

$$\begin{aligned} M\mathbf{x} + aM\mathbf{y} &= (\mathbf{x} + \mathbf{m}) + a\left(\mathbf{y} + \mathbf{m}\right) \\ &= M\left(\mathbf{x} + a\mathbf{y}\right) + a\mathbf{y} \\ &\neq M\left(\mathbf{x} + a\mathbf{y}\right) \text{ when } a \neq 0 \end{aligned} \tag{3.24}$$

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.

[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*

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.

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.

[6] at least!

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:

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

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:

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

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

The first development in what we would recognise as modern 'Machine Learning' was developed surprisingly early - as far back as the 1950s. This should give you a hint that a surprising amount of ML uses mathematics and theory which are (at least on the surface) relatively accessible: the difficulties which had to be overcome to make even simple Feedforward Neural Networks (FNNs) plausible were technological, not theoretical.

In this section we walk through the underlying mathematics and theory underpinning three increasingly complex forms of Machine Learning architecture: the Perceptron, the FNN and the Convolutional Neural Network (CNN).

In doing so, we will chase a few wild geese and follow some deliberately engineered dead ends: in doing so we hope to illuminate why certain features are present, rather than simply forcing you to take them for granted.

Chapter 12

Perceptrons

The Perceptron was developed in the 1950s as one of the first demonstrations of machine intelligence.

The Perceptron classification algorithm is incredibly simple: it outputs a 1 or a 0, depending on the assigned class of the input vector $\mathbf{x}$. The algorithm for assigning this class is simple:

$$\mathcal{P}_{\mathbf{w}}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix} > 0 \\ 0 & \text{else} \end{cases} \tag{12.1}$$

If $\mathbf{x}$ is an n dimensional vector, then $\mathbf{w}$ is an $n+1$ dimensional vector.

Aside

Biases vs. Augmentation

In the above notation, I treated $\mathbf{w}$ as an $n+1$ dimensional vector, and produced an 'augmented' vector with which to perform the dot product:

$$\begin{aligned} \mathbf{w} \cdot \tilde{\mathbf{x}} &= \mathbf{w} \cdot \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix} \\ &= \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x_0 \\ x_1 \\ \vdots \end{pmatrix} \\ &= w_0 + x_0 w_1 + x_1 w_2 + \cdots \end{aligned} \tag{12.2}$$

The same effect could have been achieved by letting $\mathbf{w}$ be an n dimensional vector and rewriting the equation as:

$$\begin{aligned} b + \mathbf{w} \cdot \mathbf{x} &= b + \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \end{pmatrix} \\ &= b + w_0 x_0 + w_1 x_1 + \cdots \end{aligned} \tag{12.3}$$

We can see that aside from some renaming ($w_0 \to b$, and $w_i \to w_{i-1}$) these are the same equation.

These two methods are equivalent. However, when it comes to training our neural network and taking derivatives, it is generally more helpful to be able to treat $\mathbf{w}$ as a singular object, rather than having to remember to grab the bias term separately.

Except when making a point, therefore, I will stick to using the augmentation formalism. Augmented vectors are denoted as follows:

$$\tilde{\mathbf{x}} = \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix} \tag{12.4}$$

This is deeply connected to the idea of Affine Transformations, which are discussed on page 18.

12.1 The Perceptron Workings

Those familiar with Linear Algebra know that $\mathbf{n}\cdot\mathbf{x} = \text{const}$ defines a *Plane* in n-dimensional space. The Perceptron algorithm serves to split the n-dimensional plane into two parts, and then compare whether the datapoint lies above or below the plane.

In 2 dimensions, this is simple: the 2D-splitting plane is simply a *line*. If the input data is (x, y), then the Perceptron simply serves to compute if the datapoint lies above or below the line.

12.1.1 Why $n + 1$ weights?

If the Perceptron simply serves to split the plane into two parts, it is natural to ask how many variables are needed to do that. In short: how big should our weight vector be?

As you should be familiar with from years of high school maths, a line in $2D$ space requires only 2 parameters to specify. You're most familiar with them in the following form:

$$y = mx + c \quad \implies \text{line specified by } (m, c) \tag{12.5}$$

This holds in higher dimensions: to split an N dimensional space into two parts with a plane requires N dimensions.

So why does the perceptron need $n + 1$ parameters?

The Incorrect Answer

It is incredibly disappointing to see that in many places online, an *utterly false* answer is found absolutely everywhere.

You will see many people claiming that the answer lies in the *bias* term: that this is needed to fulfill to role of c in our line equation: of permitting planes which do not go through the origin of the space.

This is false.

Why this is Wrong

It is true that if you write only $\mathbf{x} \cdot \mathbf{w}$ then you can only specify planes which go through the origin.

So....don't write it like that?

If we wanted to write a perceptron algorithm which used the minimal number of dimensions we would not use the dot product notation. Instead (in 2D), we would simply write:

$$\mathcal{P}(x, y) = \begin{cases} 1 & y > mx + c \\ 0 & \text{else} \end{cases} \tag{12.6}$$

In arbitrary n dimensions, this would be:

$$\mathcal{P}(\mathbf{x}) = \begin{cases} 1 & x_{N-1} > \sum_{i=0}^{N-2} w_i x_i + c \\ 0 & \text{else} \end{cases} \tag{12.7}$$

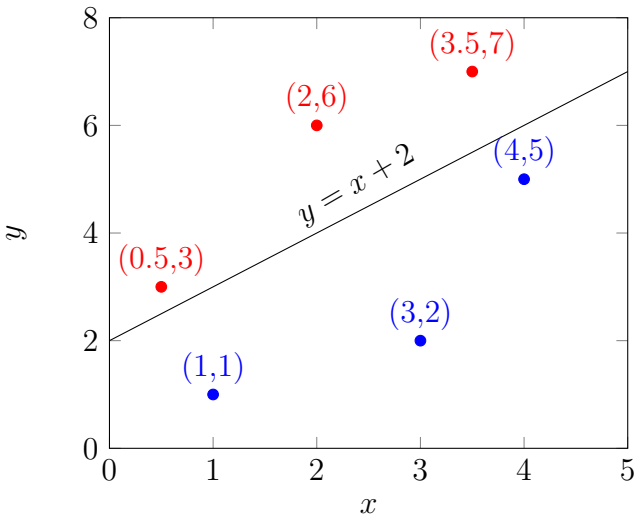
And there we have it: we have written a perceptron classifier using only n free parameters, right? The dot product notation was the problem, not the bias!

We did indeed need the bias (written as either b or c), but what we *didn't* need was the n-dimensional bit of $\mathbf{w}$. We only needed $n - 1$ of those dimensions. Pointing to the bias as the source of the extra dimensions is incorrect, as we have just proven.

The True Answer

The above classifier works if and only if you want to split the plane into two parts, **but do not care which label is given to which part**.

Consider the following classifier with $m = 1$ and $c = 2$.



The points in blue have a label of +1, and the points in red have a label of 0.

We can see that the specified line perfectly divides the points **but misclassifies every single one**. The red points lie above the line, and so satisfy $y > x + 2$ (and so have $\mathcal{P} = 1$), whilst the blue points lie below the line ($\mathcal{P} = 0$).

The issue is not the separation of the regions, it is in the **direction** that the comparison operator looks, if our classifier had tested $y < x + 2$ instead of $y > x + 2$ then it would work perfectly.

However, there is in general no way to know in advance which direction is needed, and so this failed.

This is where our extra dimension comes in handy. Consider the dot product $\mathbf{w} \cdot \tilde{\mathbf{x}}$ in two dimensions:

$$\begin{aligned} \mathbf{w} \cdot \tilde{\mathbf{x}} &> 0 \\ w_0 + w_1 x + w_2 y &> 0 \\ w_2 y &> -w_1 c - w_0 \end{aligned} \tag{12.8}$$

If we let $w_1 = -m$ and $w_0 = -c$ then:

$$w_2 y > mx + c \tag{12.9}$$

This is exactly as we expect. However, if we divide by w_2 we need to be careful, and recall how inequality operators (mis)behave when you multiply or divide by negative numbers:

$$w_2 y > mx + c \quad \longrightarrow \quad \begin{cases} y > \frac{mx+c}{w_2} & w_2 > 0 \\ y < \frac{mx+c}{w_2} & w_2 < 0 \end{cases} \tag{12.10}$$

We see that the *sign* of w_2 changes the direction of the comparison operator. If w_2 is negative then we look below the line, and if w_2 is positive then we look above it.

This is where our extra dimension comes from. Technically speaking, we could get away with simply defining (m, c) and then a direction (either ± 1), but it turns out that if we're going to the trouble of having an extra dimension anyways, then it helps the convergence if we give the line the full freedom to move w_2 around as it wants – this, for instance, allows the line to pivot around smoothly wihtout undergoing drastic changes when the direction changes.

So why do we need $n + 1$ dimensions? Because we don't just need to split the line into two parts, but also assign each part a specific label.

This is naturally tied in with the idea of the bias, but the bias itself is not the cause, since an alternative formulation which includes a bias term suffers from the same problem!

12.1.2 Training a Perceptron

Being able to compute a prediction $\mathcal{P}$ from $\mathbf{x}$ is all very well and good, but how do we determine what $\mathbf{w}$ is?

In order to do that we need to have a *training set* - a series of data points $\mathbf{x}_i$ which each have their own true label L_i.

The Perceptron training algorithm is very simple:

- Loop through the data, and make a prediction $\mathcal{P}(\mathbf{x}_i)$.
- If $\mathcal{P}_i \neq L_i$ (i.e. the prediction is wrong), update the weights according to:

$$\mathbf{w} \to \mathbf{w}' = \mathbf{w} + r \times (L_i - \mathcal{P}_i)\tilde{\mathbf{x}}$$

- Repeat until either a set number of loops has completed, or until all predictions are correct.

Here $r > 0$ is the 'learning rate' and determines by how much the weights get updated: small values mean many loops will take to converge (but it won't miss narrow optima), large values mean it will jump around quickly (but might miss the optima).

Why does this update formula work?

If the update formula is being called, then $\mathcal{P}_i \neq L_i$, and we have two options:

1. $L_i = 0$ (and so $\mathcal{P}_i = 1$). This means $\mathbf{w} \cdot \tilde{\mathbf{x}}_i > 0$, and so we want $\mathbf{w}' \cdot \tilde{\mathbf{x}}_i$ to be *smaller*.
2. $L_i = 1$ (and so $\mathcal{P}_i = 0$). This means $\mathbf{w} \cdot \tilde{\mathbf{x}}_i \leq 0$, and so we want $\mathbf{w}' \cdot \tilde{\mathbf{x}}_i$ to be *bigger*.

If we compute the new dot product after the weights have been updated:

$$\mathbf{w}' \cdot \tilde{\mathbf{x}}_i = \mathbf{w} \cdot \tilde{\mathbf{x}}_i + r\,(L_i - \mathcal{P}_i)\,|\tilde{\mathbf{x}}_i|^2 \tag{12.11}$$

By definition, both r and $|\tilde{\mathbf{x}}_i|^2$ are positive, and so:

$$\begin{aligned} \mathbf{w}' \cdot \tilde{\mathbf{x}}_i &= \mathbf{w} \cdot \tilde{\mathbf{x}}_i + \text{positive} \times (L_i - \mathcal{P}_i) \\ &= \mathbf{w} \cdot \tilde{\mathbf{x}}_i + \begin{cases} \text{positive} & L_i = 1 \\ \text{negative} & L_i = 0 \end{cases} \end{aligned} \tag{12.12}$$

We can therefore see that he new dot product gets bigger if $L_i = 1$ and gets smaller if $L_i = 0$. This pushes the dot product towards 0, and hence towards the value at which its decision will change.

This explains why $\tilde{\mathbf{x}}$ appears in the update formula: it is a vector $\mathbf{v}$ for which it is guaranteed that $\mathbf{v} \cdot \tilde{\mathbf{x}} > 0$, and hence improve the prediction – we could replace it by any linear scaling of $\tilde{\mathbf{x}}$ (for instance, $\mathbf{v} = \frac{1}{|\tilde{\mathbf{x}}|}\tilde{\mathbf{x}}$), but it has to point in the same direction.

12.2 Linearity & Perceptrons

One major limitation of Perceptrons is their inherent linearity: by definition they split the n dimensional spaces into 2 parts using a $n-1$ dimensional hyperplane. With our $n+1$ parameters we may have given them the freedom to choose which side to give which labels, but nothing yet gives them the ability to draw a curved line to disentangle data.

We *can* get around this, however – all within the framework of a simple perceptron.

We consider augmenting our input vector with *non-linear combinations of the input features.*

For instance:

$$\begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} 1 \\ x \\ y \\ x^2 \\ xy \\ y^2 \end{pmatrix} \tag{12.13}$$

In doing so we project our input vector into a higher order space (in this case, the 5-dimensional space), and because this projection ocurred in a non-linear fashion, when the Perceptron draws a line through the space and projects it back down into 2D, the result will look like a curved line.

This particular choice permits the classification of all possible hyperbolically-separable spaces, instead of just the linearly separable spaces. We can extend the dimensionality even further:

$$\begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ x \\ y \\ x^2 \\ xy \\ y^2 \\ x^3 \\ x^2y \\ xy^2 \\ y^3 \\ \vdots \\ x^{210}y^{100} \\ x^{209}y^{101} \\ \vdots \end{pmatrix} \tag{12.14}$$

In doing so we leverage the power of Taylor expansions, and can therefore use this to separate **all possible analytically separable classes**.

That's right: a single perceptron node can, with suitable modifications, work on arbitrary analytical spaces. Forget what you've been told about the XOR problem!

Aside

This isn't anything new: this is a variation on what is known as the 'kernel trick' in SVM terminology, but I rarely see it put into these terms!

The Kernel Trick varies in that it doesn't require an explicit mapping, relying instead on similarity measures; but it remains the case that for any specified 'Kernel Trick' you can decompose it into a non-linear predictor above (doing so is often intractable however, even if theoretically possible).

The problem with this is twofold: it gets complicated *really* fast. As soon as you include terms of suitably high order (I personally had trouble above 8th order), the convergence of your perceptron is absolutely terrible unless you dive in and hold its hand: the higher order terms will flip flop like crazy. Even with r tiny, each update will completely alter your decision boundaries.

This is why things like the Kernel Trick exist: they permit this kind of behaviour in a much more controlled fashion, allowing the additional dimensionality and flexibility, without the penalty of accidentally altering $\mathbf{w}$ by an amount equal to 10^{85}.

In short, whilst the Perceptron is, contrary to what you may have heard, capable of dealing with ludicrously non-linear classification tasks, the formulation is poorly suited for them.

We need something more powerful....maybe some *layers*?

Chapter 13

Multi-Layer Perceptrons

We saw that, in the case of the non-linear perceptron that, if we project our vector $\mathbf{x}$ into some high dimensional space, then it was possible for the classifier to then 'thread the needle' and draw a decision boundary which was a straight line in the high dimensional space, but a curved shape in the output space.

The problem, as we saw, was that this projection had to be manually encoded and resulted in highly unstable behaviour.

13.1 The Naive Perceptron

The basic principle of a multi-layer perceptron is that we abstract this projection process into another learnable process: rather than forming our nonlinear vector ourselves, we let the machine learn what would be a good way to manipulate the data – using a series of individual perceptrons.

The prediction algorithm then becomes:

- Take our input vector, $\mathbf{x}$ and pass it into an array of perceptrons. Each perceptron – called a 'node' or a 'neuron' then performs a dot product $\mathbf{w}_i \cdot \tilde{\mathbf{x}}$. We then collect the results into a new vector $\mathbf{x}_{\text{processed}}$
- We then repeat this an arbitrary number of times, taking our vector, passing into through an array of neurons, and then collecting the result into a vector, and passing it on to the next 'layer' of neurons.
- We can then the output of this highly dimensional process as the input to our 'decision' node which performs the usual perceptron algorithm.

A schematic of this kind of perceptron is shown in Fig. 13.1.

This algorithm is rather simple to code up: it simply requires a way to pass information through the layer. However, if one writes it up a problem will become obvious even without training.

If one randomly initialises the weights in all of the nodes and plots the values of the prediction across a range of values – as is done in figure 13.2 – it becomes very clear that something has gone awry.

Despite projecting up into our high-dimensional space, **the decision boundary is still a straight line**! This is despite our projections going up into a 5D space, down into 3D and before going into 1D.

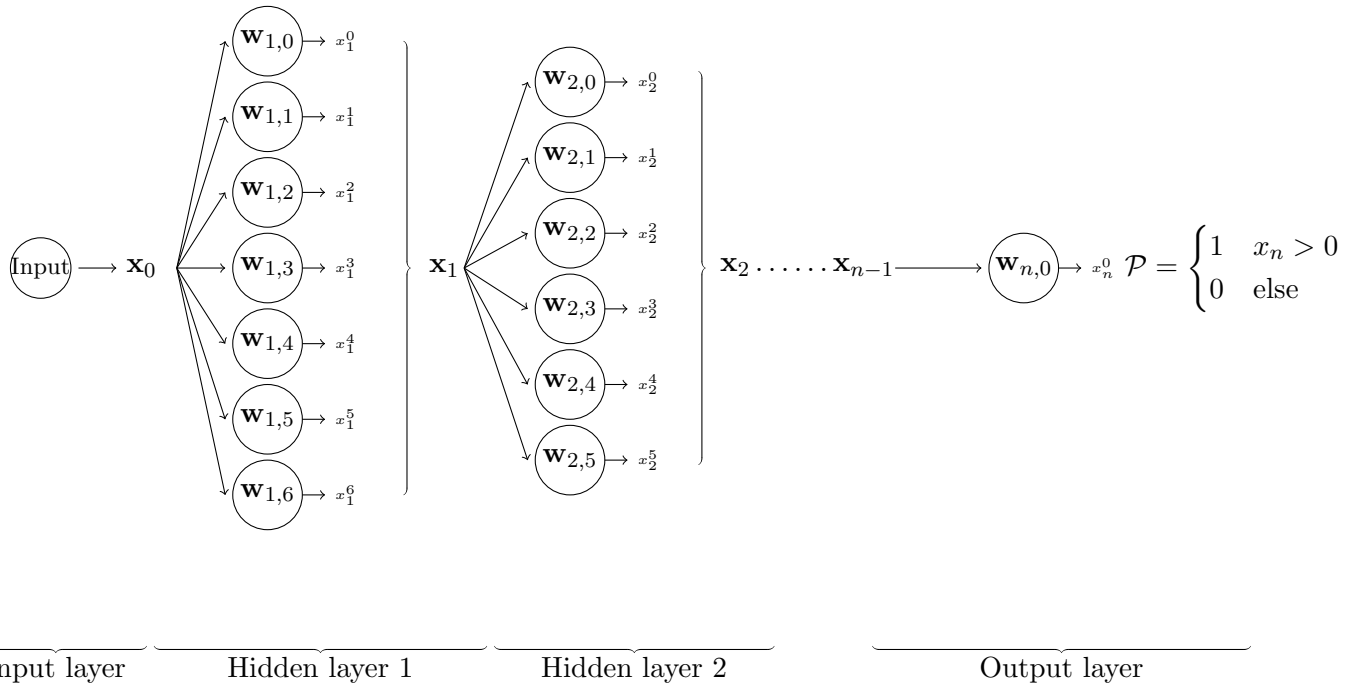


Figure 13.1: A "Naïve MLP" showing the multiple layers of dot products and collection, before a final perceptron layer.

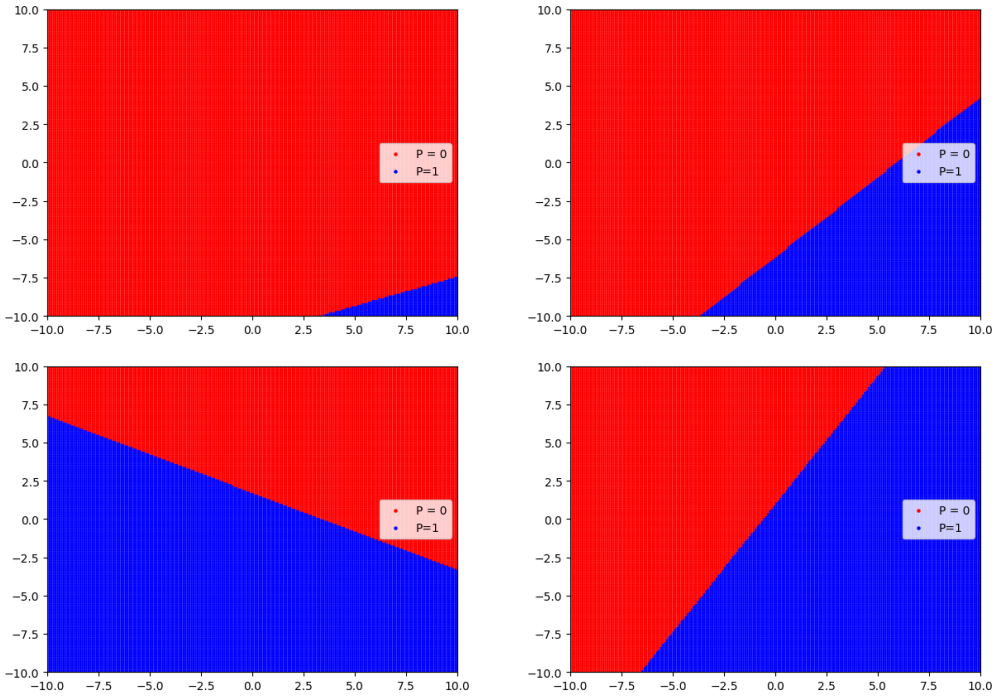


Figure 13.2: The predictions of 4 randomly initialised "naive MLP"s with dimensionality (5-3-1). The predictions for a 200x200 array of values between (-10,10) are shown. Predictions of label 0 are in red, label 1 in blue.

13.1.1 What Went Wrong

What has gone wrong here is, in fact, rather embarrassing.

If we recall chapter 3, the definition of matrix multiplication amounted to:

$$\begin{aligned} W\mathbf{x} &= \begin{bmatrix} \mathbf{w}_0 \\ \mathbf{w}_1 \\ \mathbf{w}_2 \\ \vdots \end{bmatrix} \mathbf{x} \\ &= \begin{pmatrix} \mathbf{w}_0 \cdot \mathbf{x} \\ \mathbf{w}_1 \cdot \mathbf{x} \\ \mathbf{w}_2 \cdot \mathbf{x} \\ \vdots \end{pmatrix} \end{aligned} \tag{13.1}$$

That is, matrix multiplication is equal to taking the dot product of each row of a matrix with the vector, and then stacking the results into a vector. If we observe Fig 13.1 then "taking the dot product and stacking them into a vector" is *precisely* what we are doing.

Each layer in the network is computing the matrix product with W_n, the matrix formed from stacking all of our weights in the n^{th} layer into a matrix:

$$\mathbf{x}_n = W_n \tilde{\mathbf{x}}_{n-1} \tag{13.2}$$

Or (if we recall our Affine/augmentation conversation in Chapter 3.4), this is equal to:

$$\mathbf{x}_n = \mathcal{W}_n \mathbf{x} = W'_n \mathbf{x} + \mathbf{b}_n \tag{13.3}$$

Where $W_n = [\mathbf{b}_n W'_n]$ are the weight/bias components of W_n, and $\mathcal{W}_n$ is a general Affine transformation.

The final output of our network is therefore a stack of these Affine transformations: matrix multiplications plus bias offsets:

$$\begin{aligned} \mathbf{x}_n &= \mathcal{W}_n \mathcal{W}_{n-1} \mathcal{W}_{n-2} \cdots \mathcal{W}_2 \mathcal{W}_1 \mathbf{x}_{\text{inputs}} \\ &= \mathbf{b_n} + W_n \left(\mathbf{b}_{n-1} + W_{n-1} \left(\cdots W_2 \left(\mathbf{b_1} + W_1 \mathbf{x}_{\text{inputs}} \right) \right) \right) \end{aligned} \tag{13.4}$$

Whilst this looks horrible and awful, we recall that affine transforms are associative, which means we can write **all of this as a single affine transformation** or, in other words

$$\begin{aligned} \mathbf{x}_n &= \mathcal{W}' \mathbf{x}_{\text{input}} \\ &= \mathbf{b}' + W' \mathbf{x}_{\text{input}} \end{aligned} \tag{13.5}$$

And since our final layer has dimension 1 (a scalar)...

$$x_n = b + \mathbf{w} \cdot \mathbf{x} = \mathbf{w} \cdot \tilde{\mathbf{x}} \tag{13.6}$$

All of this reduces back down into a single perceptron predictor. All of our layers and increased dimensionality **did absolutely nothing**. Diddly squat.

This network is essentially a very long, very convoluted way of computing a single dot product. Well done us.

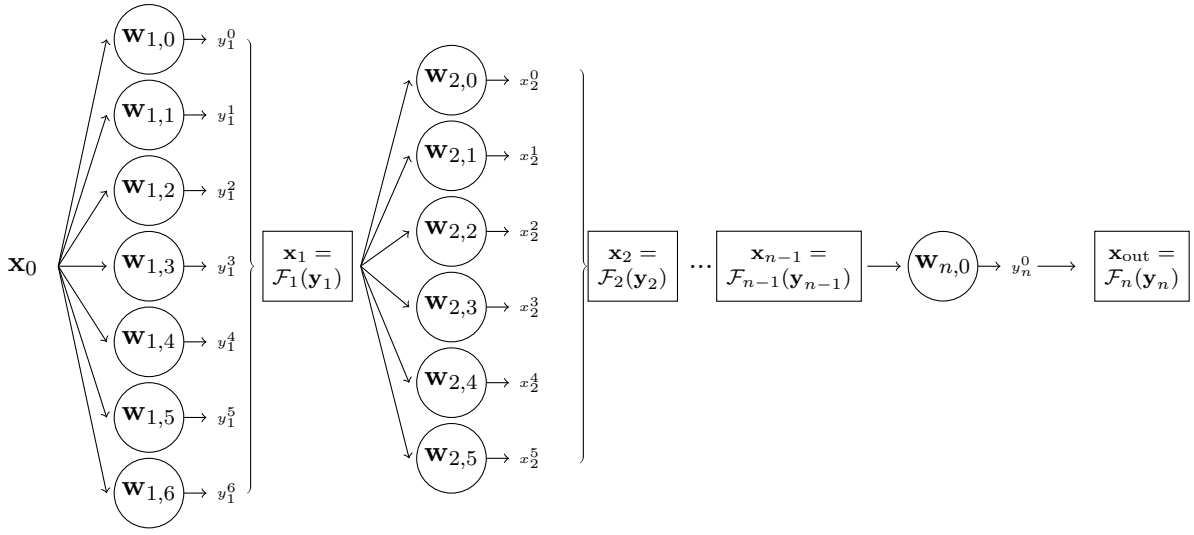


Figure 13.3: An "Activated MLP" showing the multiple layers of dot products and collection, followed by a non-linear function $\mathcal{F}$.

13.2 The Feedforward Neural Network

OK. Well that didn't go to plan, did it? Those of you familiar with neural network architecture might have thought we were onto a winner, there.

What went wrong, however, was that we were attempting to induce non-linearity[1]...using linear operations.

We projected into a higher dimensional space, hoping this would let us draw nice warped lines as straight lines – but our method of projecting was inherently linear. It was a doomed endeavour from the beginning.

Luckily, there is a simple solution to this: **do something non-linear to the vectors**.

It really is as simple as that. Instead of collecting the output of each node into $\mathbf{x}$ to be converted into $\tilde{\mathbf{x}}$ and passed onto the next layer of the network, we collect them into a temporary vector $\mathbf{y}$, and then perform some non-linear operation on that vector - the result of which is then used as $\mathbf{x}$. This function might vary between each layer, and so we denote it as $\mathcal{F}_n$:

$$\mathbf{y}_n = \mathcal{W}_n \mathbf{x}_{n-1} \qquad \mathbf{x}_n = \mathcal{F}_n(\mathbf{y}_n) \tag{13.7}$$

An example of this kind of function is shown in Fig. 13.3, and it might start to look familiar! There is practically no limit to what $\mathcal{F}$ can be: all it has to be is *not* be equivalent to an Affine transform, which is a pretty broad scope.

13.2.1 Activation Functions

Perhaps the *easiest* way to come up with a non-linear function is to come up with a nice scalar function and then *do it to each element of the vector.*

Generally speaking, so long as the function is anything other than a $f(x) = ax + b$, the

[1] or at least, non-affine-ness, which is a mathematically important distinction, if not a conceptually important one! You may assume I am using affine and linear as essentially synonymous.

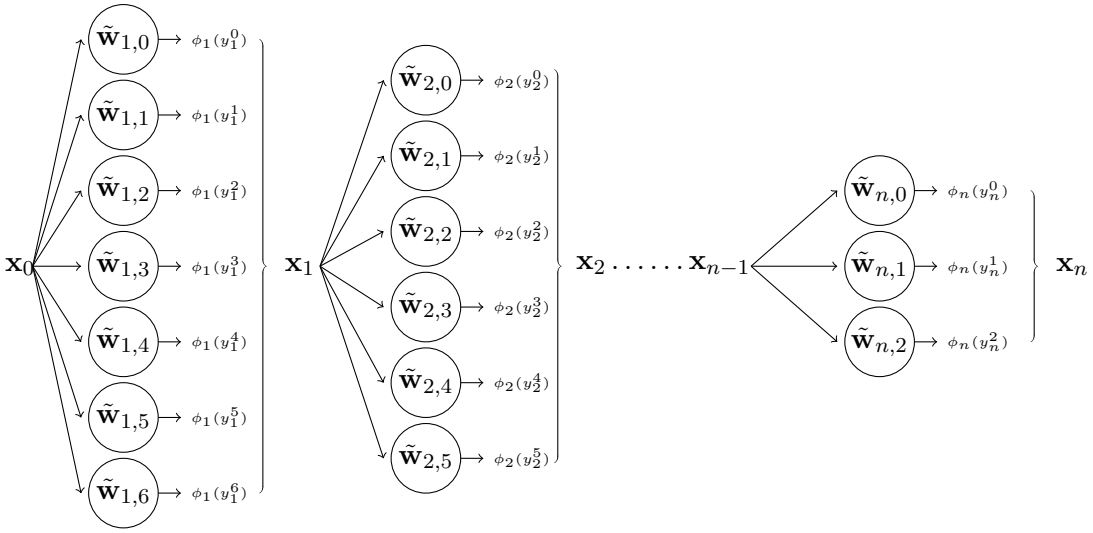


Figure 13.4: An activated MLP where the activation function is computed with the node, as a scalar function $\phi_i(x)$.

result is a nonlinear function:

$$\mathcal{F}(\mathbf{y}) = \begin{pmatrix} f(y_0) \\ f(y_1) \\ f(y_2) \\ f(y_3) \\ \vdots \\ f(y_n) \end{pmatrix} \tag{13.8}$$

When you use a function like this, it can be helpful to think of it as being computed simultaneosly with y_i that is, by the individual nodes themselves. You will therefore commonly see things such as the network depicted in Fig. 13.4, where this scalar function – called the 'activation function' – is treated as part of the dot product/node operation.

13.2.2 Non-Separable Activation Functions

Although per-node activation functions were biologically motivated by comparison with neurons, it is not necessary to consider them an inherent property of the nodes themselves.

In fact, it is often useful to consider more general functions $\mathcal{F}$, where it is proper to think of them as happening as part of the *layer*: the nodes compute y_i which the layer then gathers into $\mathbf{y}$ and the operation can only be performed once $\mathbf{y}$ is fully specified.

An example of a commonly used non-separable layer-activation function is the 'Softmax' function:

$$\text{softmax}(\mathbf{x}) = \frac{1}{\sum_i \exp(x_i)} \begin{pmatrix} \exp(x_0) \\ \exp(x_1) \\ \exp(x_2) \\ \vdots \end{pmatrix} \tag{13.9}$$

This function is used to convert arbitrary vectors $\mathbf{x}$ into meaningful probability mass arrays: the elements of $\mathbf{y} = \text{softmax}(\mathbf{x})$ sum to one and are all between 0 and 1. This layer is often used as the output for one-hot-encoding classifiers, as well as statistically motivated tasks.

It is not possible to compute this function on the nodes (without a complex series of nodes-connected-to-nodes-in-their-own-layer, which gets tiresome), because you cannot

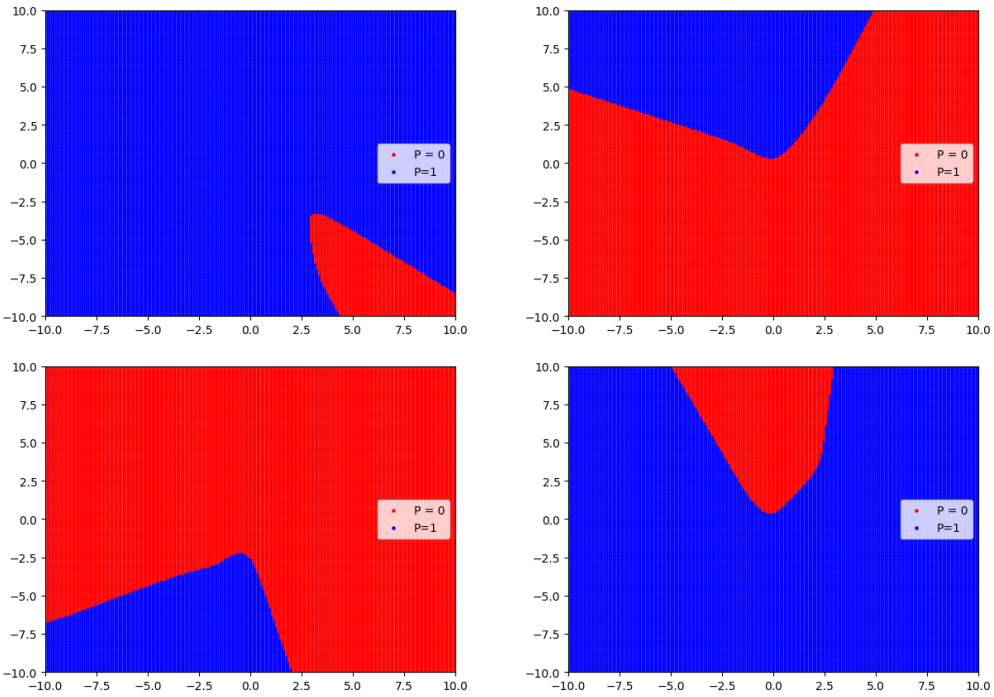


Figure 13.5: As in Fig. 13.2, but using Sigmoid activation functions instead of linear. The resulting curved lines demonstrate the effectiveness of the method.

perform the division without knowing the state of all of the other nodes in the layer.

Other common non-separable functions include the various pooling and max-ing layers common in CNNs.

For our current purposes, it suffices merely to note that, *in general* the function $\mathcal{F}$ is a per-layer function of arbitrary complexity. However, for the majority of practical purposes it is easiest to use per-node scalar activation functions.

13.2.3 The Proof is in the Pudding

We now implement a common activation function, the Sigmoid (also known as logit)

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{13.10}$$

This starts at $\sigma = 0$ at $x \ll 0$, and then increases up to $\sigma = 1$ at $x \gg 0$ with a period of linearity around $x \approx 0$.

The results of a randomly initialised (5-3-1) function with Sigmoid activation on the hidden layers and a Perceptron classifier on the final layer are shown in Fig. 13.5. Unlike the plots of the identical network with linear activation functions (Fig. 13.2) we see that our straight classification boundaries have been replaced by appropriately non-linear ones: we are ready to begin machine learning in earnest!

13.2.4 In Defence of Linear Activation Functions

It is tempting to think, given all that we have discovered, that the linear activation function $\mathcal{F}(\mathbf{x}) = \mathbf{x}$ will never see the light of day.

This is in fact wildly incorrect.

Almost all[2] neural networks have a linear activation function as their final layer.

This is because neural networks can be used to approximate *any* function, not just perform classification. If the value of this function is unbounded (i.e. it's not a probability you know should be $0 \leq p \leq 1$), then the linear activation function is needed. So long as you have non-linear activation functions within the hidden layers, the output layer being non-linear is of no consequence.

I wrote this section after I saw online[3] some people arguing that the linear activation function 'should never have been coded', which is, of course, dumb.

13.3 Universal Approximation

One of the most important properties about a FNN in the way that we have constructed them is that they are **Universal Approximators**: they are able to model arbitrary functions.

The Cybenko statement is as follows:

A FNN with a single hidden layer of infinite width and possessing the Sigmoid activation function can approximate any function.

This argument has since been generalised to networks of arbitrary depth, non-sigmoid activation functions, and non-infinite networks.

In short: for any conceivable function $f : \mathbb{R}^n \to \mathbb{R}^m$ there is *some* neural network architecture which can replicate it. It does not matter how complex the function is, or what might need to be done to compute it 'properly': a FNN of sufficient size can compute it.

The catch, naturally is twofold:

- There is very little information (*a priori*) about what 'sufficient size' is.
- These proofs are non-constructive: they tell you a network exists, but give no indication on how to go about constructing it.

In short: we know that, for every single problem, there exists an FNN which can solve it. But how to go about making that network, or how many trillions of nodes it might need, is left as an exercise to the reader.

The difficulty is not the theory, it's in the execution.

[2] In the mathematical sense, since the space of classification functions is of a lower cardinality than the space of even arbitrary scalar functions, let alone arbitrary functions on $\mathbb{R}^n$.

[3] It was reddit. Of course it was reddit

Chapter 14

Training an MLP

Training an MLP is a fiddly business: the mathematics is actually rather simple (it is merely a recurrence relation on the chain rule) – the difficulty is merely one of ensuring all the right parts end up in the right places.

14.1 Setting the Scene

The first thing we have to realise is that the simple formula of the Perceptron update formula will no longer work. We will need to tackle this as an *optimisation problem*. We therefore move away from a binary 'yes/no' output and into a continuous output.

In the case of a single-variable classifier, we might use a sigmoid as our final layer neuron (or, in the case of one-hot encoding, a series of them). If we are instead focusing on approximating an arbitrary function, then a linear activation function will suffice.

Whichever way we look at it, the output of our network is now a continuous vector $\mathbf{x}$. We might choose to post-process this into a binary answer (i.e. a classification probability > 0.5 can be converted into a 'yes' and < 0.5 into a no), but the network itself will utilise continuous outputs.

With a continuous output, we can then assign a *cost function*, $C(\mathbf{x}_{\text{output}})$. This cost function (also known as a loss function) should be a differentiable function of the prediction value and, as such, is a function of the current network state.

A well-behaved cost function $C(\mathbf{x})$ has the following behaviours:

1. $\frac{\partial C}{\partial \mathbf{x}}$ exists and is computable.
2. $C(\mathbf{x})$ is small when the network prediction is close to the 'ground truth', $\mathbf{y}_{\text{truth}}$. We may write $C = C(\mathbf{x}|\mathbf{y})$
3. It is large when the prediction is far away.

In this sense, C measures the overall performance of the network. The following score can be assigned to a 'training set' of data ($\mathbf{d}_i$) where the expected output of the network is known ($\mathbf{y}_i$) is known:

$$\mathcal{L} = \sum_{\text{data } q \text{ in set}} C\left(\mathcal{P}(\mathbf{d}_q|\mathbf{y}_q)\right) \tag{14.1}$$

This function, $\mathcal{L}$ is a function therefore of the training set and of the state of the network, which is determined by the weights:

$$\mathcal{L} = \mathcal{L}\left(\{\mathbf{w}\}\right) \tag{14.2}$$

The task of training the network is therefore equivalent to finding the set of $\{\mathbf{w}\}$ which minimise the value of $\mathcal{L}$. If we can find these values, we have found the network configuration which best approximates our input function.

We recall that the optima of a function are found whenever the derivatives are zero. If $\mathbf{w}_\ell^i$ is the weights associated with the i^{th} node in the ℓ^{th} layer, then we need to find the point where:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i} = 0 \tag{14.3}$$

We must then simultaneously solve this equation for every node in the set. **This is not analytically possible**. We must therefore resort to the methods of numerical solutions to differential equations, as discussed in Chapter 5.

In the case of Newton's gradient descent, for instance, we simply need to be able to iterate over the following equation:

$$\begin{aligned} \mathbf{w}_\ell^i \to \mathbf{w}_\ell^i - \frac{\alpha}{|\mathbf{d}_\ell^i|}\mathbf{d}_\ell^i \\ \mathbf{d}_\ell^i = \frac{\partial \mathcal{L}}{\partial \mathbf{w}_\ell^i} \end{aligned} \tag{14.4}$$

The negative sign here is because we are trying to *minimise* our cost function. We could define our cost function to be 'large when good' and 'small when bad', in which case we would use a + and be maximising.

Our goal is now clear: we must find a way to compute these gradients.

This will be a gross task, but the end result is surprisingly tractable.

14.2 Notational Clarification

The complexity in this task is one of notation and exploding dimensionality. For that reason, it is important to make sure we are clear about our notation. We recall the distinction between $\mathbf{x}$, $\tilde{\mathbf{x}}$ and $\mathbf{y}$:

$$y_i^j = \mathbf{w}_i^j \cdot \tilde{\mathbf{x}}_{i-1} \qquad \mathbf{y}_i = \begin{pmatrix} y_i^1 \\ y_i^2 \\ y_i^3 \\ y_i^4 \\ \vdots \end{pmatrix} \qquad \iff \qquad \mathbf{y}_i = W_i \tilde{\mathbf{x}}_{i-1} \tag{14.5}$$

That is, y_i^j is the *unactivated result* which comes out of node j in layer i, and which can be packaged into the vector $\mathbf{y}_i$; the *unactivated vector* associated with layer i.

Each layer[1] then performs an activation transformation, and we then augment the vector into a higher-dimension (to incorporate the bias):

$$\mathbf{x}_i = \mathcal{A}_i(\mathbf{y}_i) \qquad \tilde{\mathbf{x}}_i = \begin{bmatrix} 1 \\ \mathcal{A}_i(\mathbf{y}_i) \end{bmatrix} \tag{14.6}$$

[1]Recall that, properly speaking, it is the layer which does the activation. The nodes only do it in the case of simple separable functions

If this is a separable transform, then it can be written as:

$$\tilde{\mathbf{x}}_i = \begin{pmatrix} 1 \\ a_i\left(y_i^1\right) \\ a_i\left(y_i^2\right) \\ a_i\left(y_i^3\right) \\ a_i\left(y_i^4\right) \\ \vdots \end{pmatrix} \tag{14.7}$$

The problem is therefore simple enough to state. The issue is that our cost function is a horrible series of these functions. Taking the derivative is going to be wildly unpleasant.

14.3 The Cost Function

The Cost function associated with the network is something which is **minimised** at when the network is performing well, and **minimised** when it is performing badly.

It will be significantly easier (but not actually a requirement) if our cost function is linear in the data, that is, it can be written as:

$$\mathcal{L}(\{\mathbf{d}\}) = \sum_q C(\mathbf{d}) \qquad \Longleftrightarrow \frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \sum_q \frac{\partial C_q}{\partial \mathbf{w}} \tag{14.8}$$

If we can find the derivative as a function of a single input data, then we simply need to sum the derivatives over all the input data, and we're good to go.

We will assume that all of our cost functions will take this form, and so $\frac{\partial C}{\partial \mathbf{w}}$ is our primary goal.

A decent first guess at the cost function would be the least squares function:

$$\begin{aligned} C(\mathbf{d}|\boldsymbol{\ell}) &= (\mathcal{P}(\mathbf{d}) - \boldsymbol{\ell})^2 \\ &= (\mathcal{P}(\mathbf{d}) - \boldsymbol{\ell}) \cdot (\mathcal{P}(\mathbf{d}) - \boldsymbol{\ell}) \end{aligned} \tag{14.9}$$

Here $\mathcal{P}$ is the output of our network (the final output vector), and $\boldsymbol{\ell}$ is the label of that data. If our network outputs a scalar, then $\boldsymbol{\ell} \to \ell$ and we lose the dot product.

Other cost functions are possible – and are discussed in detail in section 15.1

14.4 Backpropagation

To compute the set of gradients $\left\{\frac{\partial C}{\partial \mathbf{w}_i^j}\right\}$ will, in general, require manual computation of every single one of these multidimensional derivatives.

Luckily, there is an easier way.

The single most important equation we will be using is this chain rule equation, which notes that:

$$\begin{aligned} \frac{\partial C_i}{\partial \mathbf{w}_i^j} &= \frac{\partial C_i}{\partial y_i^j} \frac{\partial y_i^j}{\partial \mathbf{w}_i^j} \\ &\left(\text{recall } y_i^j = \mathbf{w}_i^j \cdot \tilde{\mathbf{x}}_{i-1}\right) \\ &= \frac{\partial C_i}{\partial y_i^j} \tilde{\mathbf{x}}_{i-1} \end{aligned} \tag{14.10}$$

This is why I suggested the route of augmenting vectors rather than adding the bias manually – it makes your gradients trivially easy to compute as a whole, rather than having to do the bias separately!

14.4.1 The Output Layer

The output layer is the easiest to compute gradients for, because they do not have a knockon effect on the rest of the network in the way that the hidden layers do.

We recall that, by definition, $\mathcal{P} = \mathcal{A}(\mathbf{y_N})$, so we merely need to compute

$$\begin{aligned}\frac{\partial C_q}{\partial y_N^i} &= \frac{\partial C_q}{\partial p_i} \times \frac{\partial C_q}{\partial y_N^i} \\ &= \frac{\partial C_q}{\partial p_i} \frac{\partial}{\partial y_N^i} a(y_N^i)\end{aligned} \tag{14.11}$$

We assume that (since we chose it) if $a_N(x)$ is the activation function of the N^{th} layer, then we know that $d_N(x) = \frac{\partial a_N}{\partial x}$ is its derivative. We can then write:

$$\frac{\partial C_q}{\partial y_N^i} = \frac{\partial C_q}{\partial p_i} \times d_N(y_N^i) \tag{14.12}$$

This means that:

$$\frac{\partial C_q}{\partial \mathbf{w}_\ell^i} = d_N(y_N^i) \frac{\partial C_q}{\partial p_i} \tilde{\mathbf{x}}_{N-1} \tag{14.13}$$

If we use the least squares cost function, then:

$$\frac{\partial C_q}{\partial \mathbf{w}_\ell^i} = d_N(y_N^i) \times \left(p_q^i - \ell_q^i\right) \tilde{\mathbf{x}}_{N-1} \tag{14.14}$$

This is easy to compute & calculate. It is the product of the activation function's derivative (something we should write when we write the node's activation function itself), the difference between the i^{th} element of the prediction vector, p_q^i and it's associated 'true label' ℓ_q^i, and the output of the lower layer vector, $\tilde{\mathbf{x}}$.

One layer down, an unknown number to go....

14.4.2 The Hidden Layers

The next layer, down, however, can be trivially computed from the chain rule. Since the output from layer p, $\mathbf{y}_p$ is only directly used by the layer $p+1$, we can write:

$$\frac{\partial C_q}{\partial y_p^i} = \sum_{\text{nodes } j \text{ in } p+1 \text{ layer}} \frac{\partial y_{p+1}^j}{\partial y_p^i} \frac{\partial C_q}{\partial y_{p+1}^j} \tag{14.15}$$

That is, the derivative with respect to the y in layer p is equal to (the sum of) the way in which y_{p+1} affects C_q, multiplied by the way in which y_p affects y_{p+1}.

This seems relatively horrible to write out in mathematics, but we're almost there.

The final step is to write $\frac{\partial y_{p+1}}{\partial y_p}$ in terms of known variables. We recall that:

$$y^j_{p+1} = \mathbf{w}^j_{p+1} \cdot \begin{bmatrix} 1 \\ a_{p+1}(y^1_p) \\ a_{p+1}(y^2_p) \\ \vdots \end{bmatrix} \tag{14.16}$$

Therefore:

$$\begin{aligned} \frac{\partial y^j_{p+1}}{\partial y^i_p} &= \left[\mathbf{w}^j_{p+1}\right]_{i+1} \frac{\mathrm{d}}{\mathrm{d}y^j_p} a_{p+1}(y^i_p) \\ &= \left[\mathbf{w}^j_{p+1}\right]_{i+1} d_{p+1}(y^i_p) \end{aligned} \tag{14.17}$$

Note that the index of $[\mathbf{w}]$ is offset by one: $i+1$ not i. This is because of the bias term which sits at the first index of $\mathbf{w}$.

14.4.3 Putting it All Together

This is a lot of maths. The upside is that it is relatively trivial to actually *implement* within the neural network.

In order to compute the gradients, we have:

$$\frac{\partial C_q}{\partial y^i_\ell} = \begin{cases} \frac{\partial C_q}{\partial P^i_q} \times d_N(y^i_N) & \text{if } \ell = N-1 \\ d_\ell(y^i_\ell) \sum_j \left[\mathbf{w}^j_{\ell+1}\right]_{i+1} \frac{\partial C_q}{\partial y^j_{\ell+1}} \end{cases} \tag{14.18}$$

Where $\frac{\partial C_q}{\partial P^i_q}$ is the derivative of your chosen cost function, and $d_N(y)$ is the derivative of the N^{th} layer's activation function. Having computed all of these values, it is therefore trivial to find:

$$\frac{\partial C_q}{\partial \mathbf{w}^i_\ell} = \frac{\partial C_q}{\partial y^i_\ell} \tilde{\mathbf{x}}_{\ell-1} \tag{14.19}$$

What on Earth just happened?

This seems imposing, so it is worth reiterating what is happening.

- We first start by passing the information *forward* through the network, in order to compute our values of $\mathbf{y}$, and $\tilde{\mathbf{x}}$ since we need these to compute the gradients.
- We then move to the back of the network by first computing how the score changes if the prediction changes. This is $\frac{\partial C_q}{\partial P^i_q}$: it tells us how the score (C) of the q^{th} datapoint changes when we tweak the value of P, the prediction.
- The final layer of the network (the one which makes P) then uses this information to work out how P changes when it tweaks its unactivated values, $\mathbf{y}$. It then uses this to work out how the score C changes when y is tweaked: this is $\frac{\partial C}{\partial y_N}$.
- Th next layer down then works out how much $\mathbf{y}_N$ changes when the layer $N-1$ tweaks *its* unactivated values. The rate at which C changes in response is just the product of the higher layer's gradients times these internal tweak values, giving us $\frac{\partial C}{\partial y_{N-1}}$

- Then we repeat through the network, each time using the upper layer's gradient to compute the gradient of the next layer.
- When we reach the first layer, we are almost done.
- We then go back through the network and use $\frac{\partial C}{\partial y}$ to compute $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$ – this is a simple multiplication!

Chapter 15

Training Considerations

I won't spend too much time talking about things here because entire tomes have been written about them elsewhere, this will be a brief overview of some major themes.

15.1 Modified Cost Functions

The choice of Cost (or Loss) function is an important one, because it specifies the kinds of behaviours that you wish to encourage in the model, and – perhaps most importantly – the kinds of scales of errors that you are willing to tolerate.

For example, the least squares error does a decent enough job for fitting functions, but if your prediction value is to be interpreted as a probability, then it exhibits some undesirable behaviour.

Consider the case where a value in your training set *should* have a predicted probability of 0.25: the last squares assigns the same penalty to prediction of 0.001 as it does to a prediction of 0.5001.

However, in the realm of probability, 0.25 and 0.5 are rather 'close' – they both describe an event which is plausible; whereas 0.001 describes an event which is (almost) impossible. Although the absolute difference in probabilities is the same, the meaning behind these is very different. As a statistician I am not (particularly) worried if my model regularly predicts something as being 10% more common than it should be: I *am* worried if it regularly predicts that something common is in fact highly implausible.

My model should react far more strongly to a $0.25 \to 0.001$ error than it should to a $0.25 \to 0.5$ error, and the least squares method simply does not do that!

If we wish to encourage the model to consider the scale as important, then a cost function on the Logarithms is more appropriate. If we go back to an true/false labelled dataset, then:

$$C = \begin{cases} \log(\mathcal{P}(\mathbf{d})) & \ell = 1 \\ \log(1 - \mathcal{P}(\mathbf{d})) & \ell = 0 \end{cases} \qquad \text{and} \qquad \frac{\partial C}{\partial \mathcal{P}} = \begin{cases} \frac{1}{\mathcal{P}} & \ell = 1 \\ \frac{-1}{1-\mathcal{P}} & \ell = 0 \end{cases} \tag{15.1}$$

One popular one is (motivated by statistics) to assume a one-hot encoding, where $\boldsymbol{\ell} = (0, 0, \ldots, 1, \ldots, 0)$ is a vector of zeros, except at the index associated with its classification. This lends itself to a 'cross entropy' cost function which is (if the output vector is interpreted as a probability) merely the log-posterior of the predictor:

$$C(\mathbf{d}|\boldsymbol{\ell}) = \sum_i \ell_i \log(p_i) \qquad p_i = [\mathcal{P}(\mathbf{d})]_i \tag{15.2}$$

When writing this, one has to make special note that $0 \times \log(0) = 0$, because otherwise your computer will cry.

15.2 Degradation & Convergence

It seems intuitive that simply shouting 'MORE!' and adding more, bigger layers into a network will improve the accuracy of the prediction. After all, the Universality Theorems state that, with sufficient size, we can model any function.

Why don't I want to make my model as large as possible?

Intuitively it would seem that this could *only* make our results better. Empirically, however, it becomes obvious that this does not result in the desired improvements – in fact, making networks *larger* often makes them *worse*.

There are two slightly different ways to understand this counterintuitive result, which is known as the Degradation Problem.

15.2.1 Convergence, Gradients

A fundamental problem of numerical optimisation is that it is never possible to prove that any given location is a global maximum; and (even worse), given the finite resolution of numerical methods, it is actually impossible to prove that a given location is even an optimum, and not just an area of non-zero-but-below-computational-ϵ-gradient.

Simply put: you never know if your model has converged, and even if you did, you never know if your model has found a global solution.

These problems exist in all numerical optimisation problems (where it is often sufficient to get 'close enough'). The problem with Neural Networks is that as you add layers, you also increase the dimensionality of the optimisation space.

This induces two problems; it increase the (potential) number of local optima[1], but also makes us susceptible to the Vanishing Gradients Problem.

In a network of sigmoids where the gradient is bounded between $[-1, 1]$, the tendency of the chain rule is to cause the gradient to decrease the deeper into the model one goes; and when the model gets sufficiently large, the gradient becomes so small that the optimiser is unable to move the parameters. An opposite problem – the Exploding Gradients Problem – can also happen, in which case the optimiser moves, but in a chaotic, haphazard fashion which can cause overflows and numerical instability.

15.2.2 The Identity Problem

Consider a network of N layers, which functions perfectly, mapping input $\mathbf{x}$ to output $\mathbf{y}$. We now add this network as the 'head' of a larger model with M layers.

It would seem, intuitively, that this network should still be able to work perfectly since all the M-layer network needs to do is perform the mapping $\mathbf{x} \to \mathbf{x}$ and the perfect N-layer network will function as expected.

[1] And recall, we *know* there are multiple global optima which produce the same output, since there are multiple functions which produce the same output over the training set data – this does not even account for the number of local optima!

The problem, as it turns out, is that this mapping (the identity mapping) is actually really quite difficult for a neural network to do.

The issue is rather simple. This is a linear operation – the simplest possible linear operation, in fact.

But we went to quite some effort to grant our Neural Networks non-linear properties.

Our construction of Neural Networks was designed in such a way as to make this incredibly simple operation surprisingly difficult.

Therefore, adding layers past the point of 'being enough' can actually lead to a degradation in performance, unless you go to efforts to construct your network to permit linear operations in certain cases – ResNets are an example where the input vector $\mathbf{x}$ gets passed through the network on 'free connections'. The intermediary layers are therefore modelling the function $f(x) - x$. If the model thinks that $\mathbf{x}$ is better than a non-linear mapping then it can just set $\mathbf{w} = 0$ (for a ReLu network), such that $f(x) - x = 0 \rightarrow f(x) = x$.

These 'highway paths' are an integral part of modern architectures such as Transformers and enable networks to grow much larger than a naïve FNN can plausibly permit.

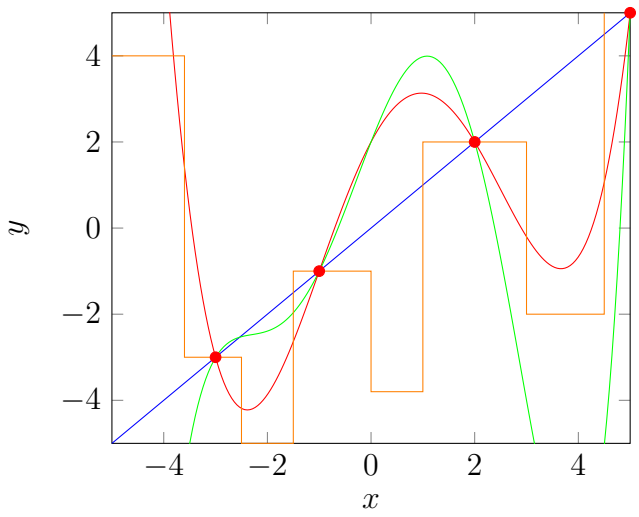
15.3 Interpolation

The Universality theorem – which, to recap, guarantees that a FNN of sufficient size is able to model any possible function – is brilliant, and is what enables modern Machine Learning as a field of study.

However, it has one major, major downside: the value of the optimal set of weights, $\{\mathbf{w}\}$ is conditioned on the input data.

That is, you do not actually train your network on a function (which usually has an infinite continuum of input and output values), but on a finite sample of points. The Universality Theorem does not guarantee that the network will fit **the** function which provides your sample, but **any** function which is capable of producing those same outputs.

For example, consider a network which is trained to predict function values based on some sample points. For the given input data, the Universality Theorem talls us absolutely nothing about which function a Neural Network will prefer – even if one seems more evidently 'correct' than the other to us.



All of these functions predict with 100% accuracy the value of the provided training points, but give wildly divergent values when we ask for predictions outside the training set.

In short, we are making the same assessment that we make whenever we perform regression and interpolation: we hope that our knowledge of the behaviour at certain points will provide insight into the behaviour of the model *in the spaces between points*.

This is by no means guaranteed: there are an infinite number of functions f which provide a finite output set $\{\mathbf{d}\}$ and whislt that's great (it's the entire basis of approximation theory!), it also means we have to be careful when making judgement calls about the behaviour of the function we observe being output from our neural network: it is only an approximation to the true function.

Trying to draw firm conclusions about f from an approximation $\tilde{f} \approx f$ is a fraught endeavour and one you must be careful with.

I loosely term this problem "The Interpolation Problem".

15.4 Overfitting & Regularisation

The 'Interpolation Problem' has a related cousin (and one that receives an awful lot more attention), namely that the following function is *also* in the set of functions which produce the provided dataset:

$$\mathcal{P}(\mathbf{d}_i) = \ell_i \tag{15.3}$$

That is, the function which outputs the label ℓ_i when given the datapoint $\mathbf{d}_i$ is an entirely valid function, and (by definition) it is also one of the infinite number of functions which minimise the Cost Function.

It is, also, the classic case of Overfitting. The model has learned absolutely nothing about 'intrinsic behaviours', or 'internal representations' and has merely learned to isolate the input data and assign it the datapoint it was supposed to.

It is the difference between a schoolchild who has learned how to perform mental maths, and one who has learned to reflexively blurt out their multiplication tables: one will be able to generalise their knowledge to new areas, the other will be stumped as soon as you ask them what 2.5×3 is, since they only know 2×3 and 3×3.

There are various ways to get around this, and I won't get into too many of them here. The split between test/validation data is an easy way to detect if this is occuring (if the model converges to 100% accuracy on the training data, but poorly on the validation data, it has overfitted), but only provides a warning system: it doesn't actually help *prevent* overfitting.

Perhaps the most common and mathematically rigorous means of preventing overfitting is to implement a Regularisation method; or (in the terminology of Bayesian statistics) a prior function.

In this case we modify our cost function to be something along the lines of:

$$\mathcal{L} = \sum_q C(\mathbf{d}_q|\boldsymbol{\ell}_\mathbf{q}) + R(\{\mathbf{w}\}) \tag{15.4}$$

Where R is a Regularisation function. A common form of Regularisation is the ℓ_2-norm

regulariser:

$$R = -\sum_{\ell}\sum_{i} \alpha |\mathbf{w}_{\ell}^{i}|^{2} \tag{15.5}$$

This value is large (i.e. close to zero) when the $\mathbf{w}$ are small, and small (i.e. far from zero) when $\mathbf{w}$ are large. The model will then preferentially choose, all else being equal, parameters where $\mathbf{w}$ is small.

This form of Regularisation is easy to handle, because it requires only minor modifications to our backpropagation:

$$\frac{\partial \mathcal{C}}{\partial \mathbf{w}_{\ell}^{i}} \rightarrow \frac{\partial \mathcal{C}}{\partial y_{\ell}^{i}} \tilde{\mathbf{x}}_{\ell-1} - \alpha \mathbf{w}_{\ell}^{i} \tag{15.6}$$

The problem is that the 'regularising strength', α is a free parameter which, by definition, makes the fit worse because we want to prevent the model from fitting perfectly. Choosing the correct α is a difficult task, and one that cannot readily be automated.