Zero, But Not Quite: The Numbers Powering Modern AI

1 Introduction

The vast majority of mathematicians are often two things: lazy, and unable to accept limitation. This has led to them finding increasingly more efficient ways of doing things, and many workarounds to seemingly obvious rules. My favourite example of this is when something was created that can automatically compute derivatives while challenging usual intuition by introducing a non-zero number whose square is zero, enter dual numbers.

Like complex numbers, dual numbers are an extension to the existing number line of real numbers, but instead of being of the form $a + bi$, where i has the property $i^2 = -1$, they are expressed as $a + b\varepsilon$ where ε has the property $\varepsilon^2 = 0$ while $\varepsilon \neq 0$. While this may seem like a strange thing to create an entirely new set of numbers for, it leads to powerful applications. Not only does it have some interesting uses in approximation and optimisation, but it fuels one of the fastest growing technologies in the world: artificial intelligence. I want to look at how dual numbers can be implemented to increase speed, accuracy and simplicity of optimisation algorithms and machine learning models, specifically in training AI.

2 The Problem with Computing Derivatives

Derivatives are used everywhere when doing calculations with anything which varies; kinematics and motion, modelling rates of change, and optimisation, yet computers struggle to calculate them in the same way as people.

Take symbolic differentiation, this is the typical "bring the power down as a coefficient, then reduce the power by one" method, e.g. $x^2 \rightarrow 2x$. Although it gives the exact derivative when done correctly, there are a few problems when trying to implement it. For example, the program must be written such that it can recognise which rule to use on what sections of the expression, when to use more than one rule, e.g. using the chain rule stacked on the quotient rule and which parts of the exponent to bring down e.g. when differentiating e^{2x}, only the 2 comes down. This makes even creating a program for computing derivatives difficult, before it has even been used.

Even after the implementation challenge there's a second problem, and it lies in the scale and complexity of the expression to be differentiated. Assuming we have a program that works for any function, many useful expressions would be too complex to use a program that does uses this method and would quickly become too messy and inefficient to evaluate, thus making it less suitable for a program that should be quick and functional. To find a way around this, we turn to numerical differentiation and finding the derivative from first principles.

Numerical differentiation is a lot easier to implement but still has a major flaw. The formula from first principles:

$$f'(x) \lim_{h \to 0} = \frac{f(x+h) - f(x)}{h}$$

allows it to be easily implemented into a program using the function to be differentiated and a chosen value of h. Now comes the problem with this method – accuracy. First principles relies on using a very small h to approximate the derivative, however due to the use of floating-point binary and catastrophic cancellation when computers try to store small values, they are quite bad at representing small numbers. This means that for any derivative calculated there is a significant level of uncertainty, making it unreliable for calculating sensitive values.

So symbolic differentiation is exact but complicated, and numerical differentiation is simple but inaccurate, therefore neither provide a suitable solution. What we need is a simple, easy method to give exact derivatives – this is where dual numbers come in.

3 Automatic Differentiation and Using Dual Numbers

This is where our new set of numbers becomes powerful. The way automatic differentiation works is incredibly simple:

- First, take the function to be differentiated. I'll use this as an example, but it can be anything:

$$f(x) = 7x^2 - 2x + 4$$

- Now, sub in our dual number, $a + b\varepsilon$, and simplify:

$$f(a + b\varepsilon) = 7(a + b\varepsilon)^2 - 2(a + b\varepsilon) + 4$$

$$= 7(a^2 + 2ab\varepsilon + b\varepsilon^2) - 2a - 2b\varepsilon + 4$$

$$= 7a^2 + 14ab\varepsilon + 7b\varepsilon^2 - 2a - 2b\varepsilon + 4$$

$$= 7a^2 - 2a + 4 + (14a - 2)b\varepsilon$$

- Notice how the $7b\varepsilon^2$ term disappears, this is due to ε's property $\varepsilon^2 = 0$

Observing the fully simplified expression, you can see that it's made up of $f(a) + f'(a)b\varepsilon$, and the derivative has been exactly computed, while having applied minimal change to the original expression.

This might seem like a fluke or trick, but it is 100% mathematically accurate due to ε^2 being zero. We can show this for any polynomial using the binomial expansion:

$$(a + b\varepsilon)^n = a^n + na^{n-1}b\varepsilon + \binom{n}{2} a^{n-2}b\varepsilon^2 + \cdots$$

Now we apply ε's property and it truncates the expansion to exactly:

$$(a + b\varepsilon)^n = a^n + na^{n-1}b\varepsilon$$

This will work for any polynomial, allowing for the use of rules like the chain or quotient rule that can get messy with longer functions to be completely avoided while still giving the exact derivative.

But what about functions that aren't polynomials? For example, how could we differentiate a non-linear function like:

$$f(x) = \sin(x^2) + \ln(2x - 3)$$

To show why the same method doesn't work for these function, we can try the same thing again, so sub in the dual number:

$$f(a + b\varepsilon) = \sin((a + b\varepsilon)^2) + \ln(2(a + b\varepsilon) - 3)$$

$$= \sin(a^2 + 2ab\varepsilon + b\varepsilon^2) + \ln(2a + 2b\varepsilon - 3)$$

- $b\varepsilon^2$ can be disregarded, and using the double angle formula we get:

$$f(a + b\varepsilon) = \sin(a^2) \cdot \cos(2ab\varepsilon) + \cos(a^2) \cdot \sin(2ab\varepsilon) + \ln(2a + 2b\varepsilon - 3)$$

But there's a problem, because $b\varepsilon$ and the different a terms are trapped inside the non-linear trigonometric and natural log functions, we can't factor anything out of the expression to get it to resemble the function plus derivative multiplied by $b\varepsilon$ form anymore. Therefore, we must find another method. For functions like these, we need one more tool – the mighty Taylor series.

For any real value of h, the Taylor series is a series expansion approximation of a function around a point $x = a$. However, it can also be used with a function of $x + h$, where h is a small distance, to estimate the function's behaviour around point a. To do this, the series is written as:

$$f(x + h) = f(x) + f'(x)h + \frac{f''(\mathrm{x})}{2!}h^2 + \frac{f^{(3)}(x)}{3!}h^3 + \cdots$$

This works by using $f(x) + f(x)h$ to set up a line and after this, every increasing order of h adds curvature, therefore increasing precision of the approximation. Because it sums to infinity, there is no end to the accuracy, and allows it to replicate the curve exactly.

Using this, we can exchange $(x + h)$ for a dual number and we observe the same thing as before when using the binomial expansion:

$$f(a + b\varepsilon) = f(a) + f'(a)b\varepsilon + \frac{f''(\mathrm{a})}{2!}(b\varepsilon)^2 + \frac{f^{(3)}(a)}{3!}(b\varepsilon)^3 + \cdots$$

And again, due to ε's property, all the term containing $b\varepsilon^2$ and higher reduce to zero, collapsing the whole thing to exactly:

$$f(a + b\varepsilon) = f(a) + f'(a)b\varepsilon$$

Now, it's worth noting that in order to fully implement automatic differentiation with a computer you must first predefine the derivatives of the non-polynomial base functions (sine,

cosine, logarithms, etc.), however, since using dual numbers removes the need to program rules like the chain rule as they do it automatically (hence the name), making it still more efficient than other methods and allows the process to avoid long and heavily involved algebraic expressions.

Now that we can use the Taylor series with dual numbers to find the exact derivative of our function, we can start to implement everything together. We can simply take the original function and apply the use of the Taylor series approximation before subbing in our dual number to collapse the series and get an exact value for the derivative.

Using our original function and substituting in dual numbers, we get:

$$f(x) = sin(x^2) + \ln(2x - 3)$$

$$f(a + b\varepsilon) = \sin(a^2) + \cos(a^2) \cdot 2ab\varepsilon + \ln(2a - 3) + \frac{2}{2a - 3} b\varepsilon$$

$$= \sin(a^2) + 2a \cdot \cos(a^2)\, b\varepsilon + \ln(2a - 3) + \frac{2}{2a - 3} b\varepsilon$$

Again, we can see that the function is now separated into the two functions of *a* and their derivatives multiplied by $b\varepsilon$, as shown with our Taylor series example. It's like magic.

The way that dual numbers avoid having to explicitly use the chain rule in situations like this is by calculating the derivative separately for the inner and outer functions of each term in the expression. For example: $\sin(x^2)$ uses the inside function of x^2 as the operation to perform on the dual number, giving:

$$(a + b\varepsilon)^2 = a^2 + 2ab\varepsilon$$

$b\varepsilon^2$ tends to zero and now we use the *a* term to sub into $f(a)$ and $2ab\varepsilon$ as the coefficient to the derivative of the function in *a*. This is why we have $\sin(a^2)$ for $f(a)$ and $2a \cdot \cos(a^2) \cdot b\varepsilon$ in the derivative.

Using these rules, a simple program can be written to separate terms at arithmetic operators, before applying the inside functions to dual numbers and substituting them back into the original expression and expanding them to find their derivatives. No need for any complicated algebra and completely avoiding inaccuracy due to the truncation of higher order terms, making the expansions exact. So now that we have this tool, where is it being used?

4 Uses of Automatic Differentiation in AI

While the actual process of automatic differentiation is used in frameworks like PyTorch and JAX, where it serves a purpose in large scale operations comes mainly in optimisation. In practice, optimisation is the process of finding the most accurate or efficient output with the resources given, it is used in everything, from fundamentals of how your computer decides where to store your files all the way up to managing fuel consumption in rockets. What I want to look at is how automatic differentiation is used in machine learning and, more specifically

training artificial intelligence. To do this we first understand how normal optimisation works without dual numbers, then look at what they allow for.

Machine learning is how AI models recognise patterns in data, fuelling the predictions it makes and decisions on new data and data it is given. It allows a computer to learn and solve problems, similarly to how a person would, without each output being programmed. Usually, the way this works is through being fed to a model loads and loads of data then using a function that maps each input to an output by minimising loss. For example, Google translate was trained on using websites in different languages.

But how does it actually work? This is where optimisation comes in. An AI works off of millions of adjustable numbers called weights, and to optimise a model, we must find the best values for these weights in order to make the outputs as accurate as possible – which is an optimisation problem. To do this, the method typically used is called gradient descent and is used to minimise a loss function that measures the accuracy in a model's predictions. It works by slightly adjusting the values of weights until the loss function is as close to zero – the optimum point – as possible. To do this is uses this formula:

$$w_n - \eta \frac{\partial L}{\partial w_n} \to w_{n+1}$$

Where L is the loss and w_{n+1} is the value of the new weight, this process is repeated until the loss is minimised, meaning the accuracy is as high as possible.

To perform this adjustment of the weights, we need to find $\frac{\partial L}{\partial w_n}$, the rate of change of loss with respect to each weight. This is where automatic differentiation becomes useful.

In practice, automatic differentiation has two modes, forward and reverse mode, both computing gradients from outputs, with respect to a number of inputs. Forward mode is most efficient when there are few inputs and many outputs as, however this is less useful in training AI as there are usually millions of weights feeding into one loss value, so we tend to more often use reverse mode.

Reverse mode, which is far more commonly used for machine learning, is when a gradient is calculated from many inputs to few outputs and is more commonly known as back propagation. It uses the model's loss value to work backwards and compute the gradient with respect to all of the model's weights in a single pass, making it scalable and fast.

Together both modes rely of the same benefits of automatic differentiation: being fast and, the fact it is able it to be used without symbolic manipulation or inaccuracies of numerical differentiation.

And with that the seemingly unremarkable set of dual numbers has the ability to train one of the most powerful tools humans have yet to have created: AI. I thought that this in itself was poetic in a way and is just one of amazing corners of maths that I think makes the subject so endlessly interesting.