Derivatives - represent the rate of change of a function (gradient)

- **Derivative Rules**
Interaction	Overall Change
Addition	$(f+g)' = f' + g'$
Multiplication	$(f \cdot g)' = f \cdot dg + g \cdot df$
Powers	$(x^n)' = \frac{d}{dx}x^n = nx^{n-1}$
Inverse	$\left(\frac{1}{x}\right)' = -\frac{1}{x^2}$
Division	$\left(\frac{f}{g}\right)' = \left(df \cdot \frac{1}{g}\right) + \left(\frac{-1}{g^2}dg \cdot f\right)$

ln(x)	1/x
e^x	e^x

Multiplication by constant	cf	cf′
Power Rule	x^n	nx^{n-1}
Sum Rule	f + g	f′ + g′
Difference Rule	f - g	f′ − g′
Product Rule	fg	f g′ + f′ g
Quotient Rule	f/g	$(f' g - g' f)/g^2$
Reciprocal Rule	1/f	$-f'/f^2$
Chain Rule (as "Composition of Functions")	f º g	(f′ º g) × g′
Chain Rule (using ′)	f(g(x))	f′(g(x))g′(x)
Chain Rule (using $\frac{d}{dx}$)	$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$	

- **Partial Derivatives -** A derivative where we hold some variables constant
  - E.g. if we have a function with numerous variables, we can have a derivative of one variable while holding other variables constant ∂y/∂x

    f(x,y)= x^2 + y^3

    To find its partial derivative **with respect to x**
    number like 5 or something):

  - f'x = 2x + 0 = 2x

Chain Rule

- **Normal**
  - We use the chain rule for differentiating anything even remotely complex
  - We use it to be able to differentiate a function of a function
  - $$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$
  - $$\frac{d}{dx}\left[(f(x))^n\right] = n(f(x))^{n-1} \cdot f'(x)$$
    $$\frac{d}{dx}\left[f(g(x))\right] = f'(g(x))g'(x)$$

Chain Rule

If f and g are both differentiable and $F(x)$ is the composite function defined by $F(x) = f(g(x))$ then F is differentiable and F' is given by the product

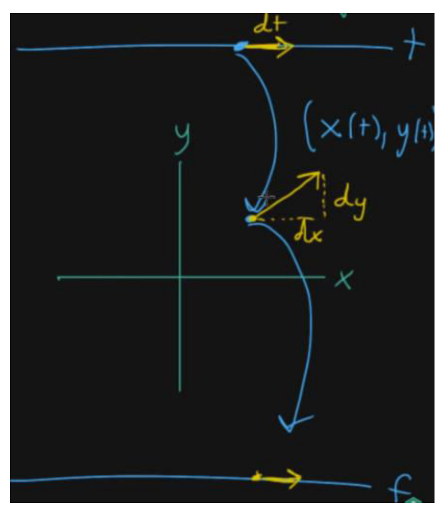
$$F'(x) = \underbrace{f'(g(x))}_{\text{Differentiate outer function}}\ \underbrace{g'(x)}_{\text{Differentiate inner function}}$$

Multivariable Chain Rule

- **Multivariable Function -** A function that takes multiple inputs and comes out with one output
- **The simplest case -** Where each input of a multivariable function are functions all of which use the single same input **t**, which leads to a simplified **multivariable chain rule**
  - *The simplest case is where you have one variable **t**, which **maps** to some **space** using numerous functions, like **x(t)** and **y(t)**, which are then composed together in a function **f**, which comes out with a single output **z***
    - ***t** -> **(x(t), y(t))** -> **z***
  - You get to this rule by substituting the functions into our multivariable function **f**, then using the chain rule on the substituted function. There you notice the below pattern (note the partial and full derivatives)

$$\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x} \cdot \frac{dx}{dt} + \frac{\partial f}{\partial y} \cdot \frac{dy}{dt}$$

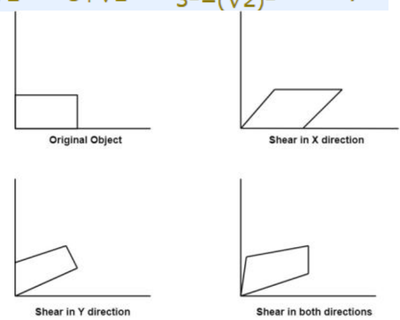
Multivariate Chain Rule Initiation - *Why would we expect something like this to happen?*



- *In above example, we are mapping from a number line **t** to a 2D **xy** plane to a real number line **f***
- A nudge in the t space will cause a nudge in the xy space will cause a nudge in f
- **dx/dt** tells us a ratio in which a tiny nudge in **t** will result in a tiny nudge in **x**
- **∂f/∂x** tells us a ratio in which a tiny nudge in **x** will result in a tiny nudge in **f**
- Therefore, **(∂f/∂x * dx/dt)** tells us that the change in **f** is equal to the change in f caused by **x**, which was in turn caused by the change in **t**
  - **df** (change in f) = **(∂f/∂x * dx/dt)** (ratio) multiplied by **dt** (extent of change) + same for y
  - *We can then rearrange for df/dt to get the gradient*

Dual Numbers https://www.youtube.com/watch?v=ZGSUrfJcXmA

- **Dual Numbers** are pairs of numbers **(a, b),** written as **a + bƐ**,
  - **Ɛ -** a free symbol that allows us to tell apart the **a** position from the **b** position
    - **Ɛ** is **nilpotent** - **Ɛ**^{2} = 0 (remember this for products of dual numbers)
    - Notation that is preferred over the tuple representation
  - *The whole idea of dual numbers is that **(a, b) * (c, d) = (ac, ad+bc)***
  - Dual number notation is used because we **can take derivatives easily**
    - $$f'(x) := \frac{f(x + \varepsilon) - f(x)}{\varepsilon}$$
  - You have to cancel out the **Ɛ**'s first, especially in times of things like chain rule, to make sure you don't 0/0
  - It is **used in automatic differentiation**
    - This is **because you can calculate the derivative while calculating the value of a function**
- If you want to find out the second derivative, you make **Ɛ**^{3} **= 0**, but not **Ɛ**^{2} **= 0**
  - You tend to do this with matrices
- **<u>Dual Numbers (Matrix) Math</u>**
- **Conjugate -** A conjugate **z*** of two terms is such that **z = (a + b) -> z* = (a - b)**
  - Used to, for example, move square roots from denominator to numerator $\frac{1}{3-\sqrt{2}} \times \frac{3+\sqrt{2}}{3+\sqrt{2}} = \frac{3+\sqrt{2}}{3^2-(\sqrt{2})^2} = \frac{3+\sqrt{2}}{7}$
  - With **dual numbers**, **z = a + bƐ** and **z = a - bƐ**
- **zz* = a**^{2}
- a = $\begin{bmatrix} a & 0 \\ 0 & a \end{bmatrix}$, bƐ = $\begin{bmatrix} 0 & b \\ 0 & 0 \end{bmatrix}$, a + bƐ = $\begin{bmatrix} a & b \\ 0 & a \end{bmatrix}$
- Dual numbers **a + bƐ** represent **scaling + shear** (tilt) transformations

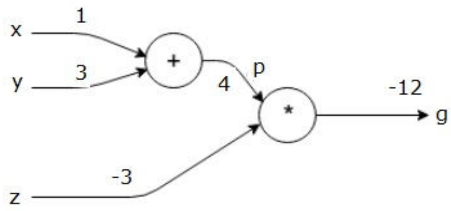


- Compare with **complex numbers a + bi** $\begin{bmatrix} a & -b \\ b & a \end{bmatrix}$, representing scaling + **rotation**
- Matrix algebra, higher order differentiation, jacobian matrix, etc

$$\begin{aligned}
\langle u, u' \rangle + \langle v, v' \rangle &= \langle u + v, u' + v' \rangle \\
\langle u, u' \rangle - \langle v, v' \rangle &= \langle u - v, u' - v' \rangle \\
\langle u, u' \rangle * \langle v, v' \rangle &= \langle uv, u'v + uv' \rangle \\
\langle u, u' \rangle \,/\, \langle v, v' \rangle &= \left\langle \frac{u}{v}, \frac{u'v - uv'}{v^2} \right\rangle \quad (v \neq 0) \\
\sin\langle u, u' \rangle &= \langle \sin(u), u' \cos(u) \rangle \\
\cos\langle u, u' \rangle &= \langle \cos(u), -u' \sin(u) \rangle \\
\exp\langle u, u' \rangle &= \langle \exp u, u' \exp u \rangle \\
\log\langle u, u' \rangle &= \langle \log(u), u'/u \rangle \quad (u > 0) \\
\langle u, u' \rangle^k &= \langle u^k, k u^{k-1} u' \rangle \quad (u \neq 0) \\
\left| \langle u, u' \rangle \right| &= \langle |u|, u' \operatorname{sign} u \rangle \quad (u \neq 0)
\end{aligned}$$

-

Computational Graphs



- A **directed** graph, where **nodes = operations**, and where **edges = numbers**
- g = (x+y) * z
- The **arguments** are the inputs on the left
- Forward Pass - evaluating the value of the mathematical expression of a graph
- Backward Pass - compute the **partial derivatives** for **each input w.r.t the output**
    - Gradients used for training the neural network (w/ gradient descent)
- For example, in the case of **g = (x+y) * z**, we want to find the gradients: $\frac{\partial x}{\partial f}, \frac{\partial y}{\partial f}, \frac{\partial z}{\partial f}$
- **Methodology**
- *We start from the end*
- First, we find the gradient of ∂**g**/∂**g**, which is 1
- Then, we find the gradients through the last node (*****), ∂**g**/∂**p** and ∂**g**/∂**z**
    - Since the operation is *, we know that g = pz, so ∂**g**/∂**p = z = -3** and ∂**g**/∂**z = p = 4**
        - We plugged in the numbers from the forward pass
- Now, we find the gradients through the second last node **w.r.t g**, in this case ∂**g**/∂**x** and ∂**g**/∂**y**
    - We do this **using chain rule** (automatic differentiation)
        - ∂g/∂x = ∂g/∂p * ∂p/∂x
        - ∂g/∂y = ∂g/∂p * ∂p/∂y
    - We know that **p = x + y**, so ∂**p**/∂**x** = ∂**p**/∂**y = 1**
        - ∂g/∂x = ∂g/∂p * 1
        - ∂g/∂y = ∂g/∂p * 1
    - We know that ∂**g**/∂**p = -3**
    - Therefore,
        - ∂g/∂x = -3 * 1 = -3
        - ∂g/∂y = -3 * 1 = -3
- *This is fast as we used the already computed values that we got from the forward pass*
- **To train a network**
- For data point x in data set, **do forward pass** with **x as input**, and calculate **cost c as output**
- After that, **do backward pass** from **c**, calculating **gradients for all nodes in graph** (which includes nodes representing **neural network weights**)
- Update weights doing W = W - LR * Deltas
- Repeat until stop criteria is met

Automatic Differentiation - numerically evaluates the derivative of a function specified by a computer program

- Uses the **chain rule** repeatedly on basic operations like addition, subtraction, multiplication, division, etc and elementary functions like exp, log, sin, cos, etc to compute the derivatives of those functions
- **Adjoint Differentiation**
- **Automatic Differentiation vs Symbolic Differentiation vs Numerical Differentiation**
    - ***All of these are methods of automatic differentiation, as opposed to manual differentiation***
    - Symbolic and numerical differentiation **both suffer with higher order derivatives**, where complexity and errors increase
    - Symbolic and numerical differentiation both **suffer with** computing partial derivatives wrt **many inputs**
    - **Automatic Differentiation -** http://114.55.97.172/guandong/algo/blob/569533b20443430842ab511ab85de3521f865699/CSC321%20Lecture%2010%EF%BC%9AAutomatic%20Differentiation.pdf
        - **Precise**
        - Operates directly on the program of interest
        - Instead of producing an expression for a derivative, it obtains its numerical value
        - It bypasses the inefficiency of symbolic differentiation by leveraging intermediate variables present in the function.
        - It is used because you can find the derivative of a function while calculating the value of the function
            - **By using .Dual Numbers.**
            - **By using .Computational Graphs.**
        - *It makes use of the fact that expressions are built up of simpler operations, the derivatives of which we know, that we can compose the simpler operations together with the **chain rule***
        - $$y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3$$

$$\frac{dy}{dx} = \frac{dy}{dw_2}\frac{dw_2}{dw_1}\frac{dw_1}{dx}$$

forward accumulation computes the recursive relation: $\frac{dw_i}{dx} = \frac{dw_i}{dw_{i-1}}\frac{dw_{i-1}}{dx}$ with $w_3 = y$,

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_{n-1}}\frac{\partial w_{n-1}}{\partial x} = \frac{\partial y}{\partial w_{n-1}}\left(\frac{\partial w_{n-1}}{\partial w_{n-2}}\frac{\partial w_{n-2}}{\partial x}\right) = \frac{\partial y}{\partial w_{n-1}}\left(\frac{\partial w_{n-1}}{\partial w_{n-2}}\left(\frac{\partial w_{n-2}}{\partial w_{n-3}}\frac{\partial w_{n-3}}{\partial x}\right)\right) = \cdots$$

reverse accumulation computes the recursive relation: $\frac{dy}{dw_i} = \frac{dy}{dw_{i+1}}\frac{dw_{i+1}}{dw_i}$ with $w_0 = x$

        - **Forward Mode -** traverses the chain rule from inside to outside, starts **dw_1/dx** and ends **dw_i/dy**
        - **Backward Mode -** traverses the chain rule from outside to inside, starts **dw_i/dy** and ends **dw_1/dx**
        - **Two versions of automatic differentiation**
            - **Forward Mode -** augmenting each intermediate variable during evaluation of a function with its derivative
                - Conceptually simpler mode
                - **Intermediate Variable -** a variable representing a simple expression of a pair of arguments or other intermediate variables in a simple operation
                - We replace intermediate variables in a function with tuples where **x -> (x, x')**, where x' is the derivative of that variable
                    - The xs are known as **primals** and they are paired with their **tangents**
                - We first take the original expression, then find all of the primals
                - We then simultaneously compute all of their tangents
                - A single pass through the function produces both the original output but the partial derivative of interest too
                - Forward mode autodiff allows us to compute partial derivatives of any output with respect to an input variable in a single pass
                - We need to run a separate forward pass for each input variable of interest

            - i.e. if we have a function $f(x_1, x_2)$ which outputs (y_1, y_2), forward diff allows us to calculate partial derivatives $\partial(x_m/y_n)$ with respect to both y_1 and y_2 in a single pass
            - However, we have to make a forward pass for each input variable x_1 and x_2 to get partials with respect to both input variables
    - **We use forward mode where the number of input variables is less than the number of output variables**
    - *We set the tangents of all input variables to 0, except for the single variable we want to calculate the derivatives of, which we set to 1*
    - **Jacobian matrix -** a matrix where each column is for partial derivatives of an input variable and each row is of a different intermediate variable
    - Forward mode computes one column of the Jacobian matrix at a time
    - **Implementations**
        - **Operator Overloading -** creating a class that holds the primal, tangent tuples, then implements functions dealing with simple operators using derivative rules
            - Abstracts away the actual derivative computation
        - **Source Code Transformations -** input source code is manipulated to produce a new function
- **Reverse Mode**
    - Used when there are a lot of inputs (parameters), and fewer outputs
    - **More suitable for machine learning models**
    - Instead of propagating forwards, the derivatives will be propagated backwards from the output
    - The gradient at each variable **n** is the sum of gradients with its children multiplied by their gradient with the whole function, **n' = Σ(c' * dc/dn)**
    - We start with a forward pass to evaluate the intermediate variables
    - Instead of simultaneously calculating the derivatives, we store the values and dependencies of intermediate variables in memory
    - After completion of the forward pass, we carry out the backward pass by computing the derivatives of the output with respect to the intermediate variables
        - These intermediate variables are known as **adjoints**
        - We compute the values of **adjoints** by looking at the children of the adjoints and calculating the **product of the children with the derivative of themselves with respect to their parent**, then summing the products
    - In the end, we get partial derivatives with respect to each input
    - *In the context of neural networks, the **backpropagation** algorithm, which is used to compute the derivatives with respect to weights, is a **special case of reverse mode autodiff**, where **adjoints** with respect to intermediate layer activations are propagated backwards through the network*
    - Reverse mode computes one row of the Jacobian at a time
    - Slightly more memory intensive

Other methods of Differentiation

- **Symbolic Differentiation -** Automatic version of manual differentiation, applies standard derivative rules
  - **Precise**
  - **Based on Trees**, it is quite impractical for something like backpropagation
  - Requires the input function to be in **closed form -** can't use control flow mechanisms like conditionals, for loops or recursion
    - It means that it is expressed using a finite number of standard operations
  - Suffers from **Expression Swell -** Derivative expressions may be exponentially longer than the original function
    - Some rules like the product rule lead to a lot of repeated computation
  - Slow - Leads to inefficient code
  - It is difficult to convert a computer program into a single expression
  - You break down each of the parts of the formula into individual problems, then apply the differentiation rules on each part and combine the results to get a single formula
- **Numerical Differentiation -** the method of finite differences -
  - **Calculates an approximate value for the derivative**
  - **Steps**
    1. You want to find the derivative x
    2. Pick a small value for a delta-x
    3. If the delta-x is a small value, then the slope between (x, f(x)) and (x+delta_x, f(x+delta_x)) is a good approximation for f'(x)
    4. You calculate the slope by doing $$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$
    - This gradient estimation is called a **difference quotient**
      - It's called a **forward difference quotient** if *h is positive*
    - We can do both, go forward a certain amount and go backward a certain amount, this is called a **symmetric difference quotient**
      - $$\lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}.$$
  - Slow and inefficient, if you have a lot of dimensions (parameters) to your gradient O(n)
  - *Used mostly when you don't know your function and can only sample it*
  - It requires a lot of computation for a high-dim function
  - Introduces round-off errors in the **discretization** process and cancellation
    - **Discretization -** Transferring continuous functions into discrete counterparts
    - **Discretization error -** Error resulting from the fact that a function of a continuous variable is represented in the computer by a finite number of evals