

Backpropagation

Ryan Cotterell and Clara Meister



Administrivia

Changes in the Teaching Staff

- Clara Meister (Head TA)
 - BSc/MSc from Stanford University
 - Despite the last name, my German ist sehr schlecht
- Niklas Stoehr
 - Germany → China → UK → Switzerland
 - I like interdisciplinarity: NLP meets political and social science
- Pinjia He
 - PhD from The Chinese University of Hong Kong
 - Focus: robust NLP, NLP meets software engineering
- New TA: **Rita Kuznetsova**
 - PhD from Moscow Institute of Physics and Technology
 - Postdoc in the BMI Lab

Course Assignment / Project Update

- About 60% of you want to do a long problem set that will also involve some coding
 - The teaching staff is preparing the assignment
 - We will update you as things become clearer!
- About 40% of you want to write a research paper
 - You should form groups of 2 to 4 people
 - Feel free to use Piazza to reach out to other students in the course
 - We will require you to write a 1-page project proposal where we will give you feedback on the idea
 - Expect to turn this in before the end of October; date will be given soon

Why Front-load Backpropagation?

NLP is Mathematical Modeling

- Natural language processing is a mathematical modeling field
- We have problems (tasks) and models
- Our models are almost exclusively data driven
 - When statistical, we have to estimate parameters from data
 - How do we estimate the parameters?
- Typically parameter estimation is posed as an optimization problem
- We almost always use ***gradient-based*** optimization
 - This lecture teaches you how to compute the gradient of virtually any model efficiently

Why front-load backpropagation?

- We are front-loading a very useful technique: backpropagation
 - Many of you may find it irksome, but we are teaching backpropagation **out of the context** of NLP
- Why did we make this choice?
 - Backpropagation is the 21th century's algorithm: You need to know it
 - At many places in this course, I am going to say: You can compute X with backpropagation and move on to cover more interesting things
 - Many NLP algorithms come in duals where one is the “backpropagation version” of the other
 - Forward → Forward–Backward (by backpropagation)
 - Inside → Inside–Outside (by backpropagation)
 - Computing a normalizer → computing marginals

Warning: This lecture is very technical

- At subsequent moments in this course, we will need gradients
 - To optimize functions
 - To compute marginals
- Optimization is well taught in other courses
 - Convex Opt for ML at ETHZ (401-3905-68L)
- Automatic differentiation (backpropagation) is rarely taught at all
- Endure this lecture now, but then go back to it at later points in the class!

Structure of this Lecture

- ① Backpropagation ② Calculus Review ③ Computation Graphs ④ Reverse-Mode AD
- 

Supplementary Material

Chris Olah's Blog, Justin Domke's Notes, Tim Vieira's Blog, Moritz Hardt's Notes, Baur and Strassen (1983), Griewank and Walter (2008), Eisner (2016)

Backpropagation

Backpropagation: What is it really?

- Backpropagation is the single most important algorithm in modern machine learning
- Despites its importance, most people don't understand it very well! (Or, at all)
- This lecture aims to fill that technical lacuna

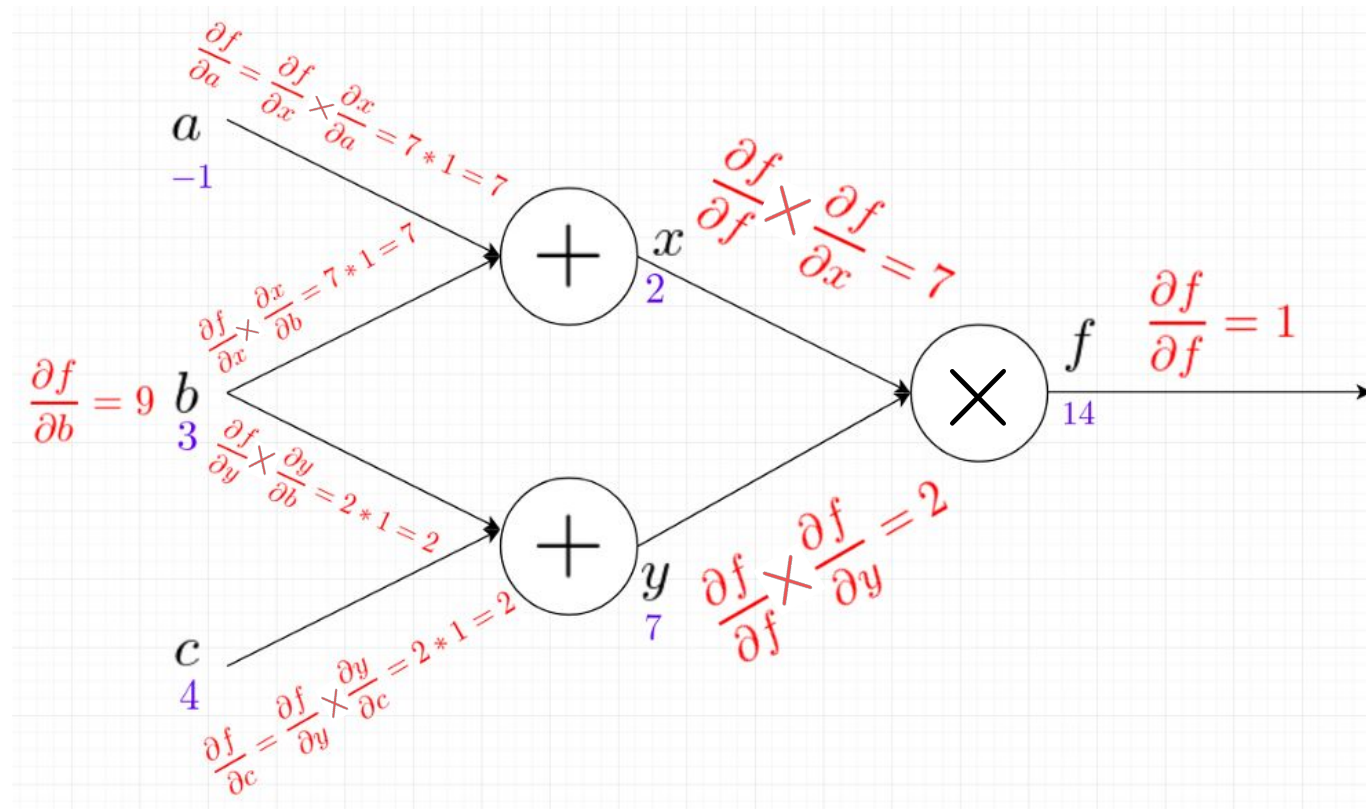
What people think backpropagation is...

The Chain Rule

$$\frac{\partial}{\partial x} [f(g(x))] = f'(g(x))g'(x)$$

What backpropagation actually is...

A linear-time dynamic program for computing derivatives



Backpropagation – a Brief History



- Building blocks of backpropagation go back a long time
 - The chain rule (Leibniz, 1676; L'Hôpital, 1696)
 - Dynamic Programming (DP, Bellman, 1957)
 - Minimisation of errors through gradient descent (Cauchy 1847, Hadamard, 1908)
 - in the parameter space of complex, nonlinear, differentiable, multi-stage, NN-related systems (Kelley, 1960; Bryson, 1961; Bryson and Denham, 1961; Pontryagin et al., 1961, ...)
- Explicit, efficient error backpropagation (BP) in arbitrary, discrete, possibly sparsely connected, NN-like networks apparently was first described in 1970 by Finnish master student Seppo Linnainmaa
- One of the first NN-specific applications of efficient BP was described by Werbos (1982)
- Rumelhart, Hinton and William, 1986 significantly contributed to the popularization of BP for NNs as computers became faster

Backpropagation – a Brief History



- Building blocks of backpropagation go back a long time
 - The chain rule (Leibniz, 1676; L'Hôpital, 1696)
 - Dynamic Programming (DP, Bellman, 1957)
 - Minimisation of errors through gradient descent (Cauchy 1847, Hadamard, 1908)
 - in the parameter space of complex, nonlinear, differentiable, multi-stage, NN-related systems (Kelley, 1960; Bryson, 1961; Bryson and Denham, 1961; Pontryagin et al., 1961, ...)
- Explicit, efficient error backpropagation (BP) in arbitrary, discrete, fully connected, NN-like networks apparently was first described in 1973 by master student Seppo Linnainmaa
- One of the first NN-specific applications of efficient BP was described by Rumelhart and McClelland (1982)
- Rumelhart, Hinton and Williams, 1986 significantly contributed to the popularization of BP for NNs as computers became faster

See this [critique](#) for some CS drama!!

Why study backpropagation?

Function Approximation

- Given inputs \mathbf{x} and outputs \mathbf{y} from a set of data $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^N$, we want to fit some function $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})$ (using parameters $\boldsymbol{\theta}$) such that it predicts \mathbf{y} well
- I.e., for a loss function L we want to minimize

$$\sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y})$$

Why study backpropagation?

Function Approximation

- Given inputs \mathbf{x} and outputs \mathbf{y} from a set of data $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^N$, we want to fit some function $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})$ (using parameters $\boldsymbol{\theta}$) such that it predicts \mathbf{y} well
- I.e., for a loss function L we want to minimize

$$\sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y})$$

(unconstrained) optimization problem!

Why study backpropagation?

- Parameter estimation in a statistical model is optimization

$$\min_{\boldsymbol{\theta}} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y})$$

- Many tools for solving such problems, e.g. gradient descent, require that you have access to the gradient of a function
 - This is about computing that gradient

Why study backpropagation?

- Parameter estimation in a statistical model is optimization

$$\min_{\boldsymbol{\theta}} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y})$$

- Consider gradient descent

init $\boldsymbol{\theta}^{(0)}$; $m = 0$

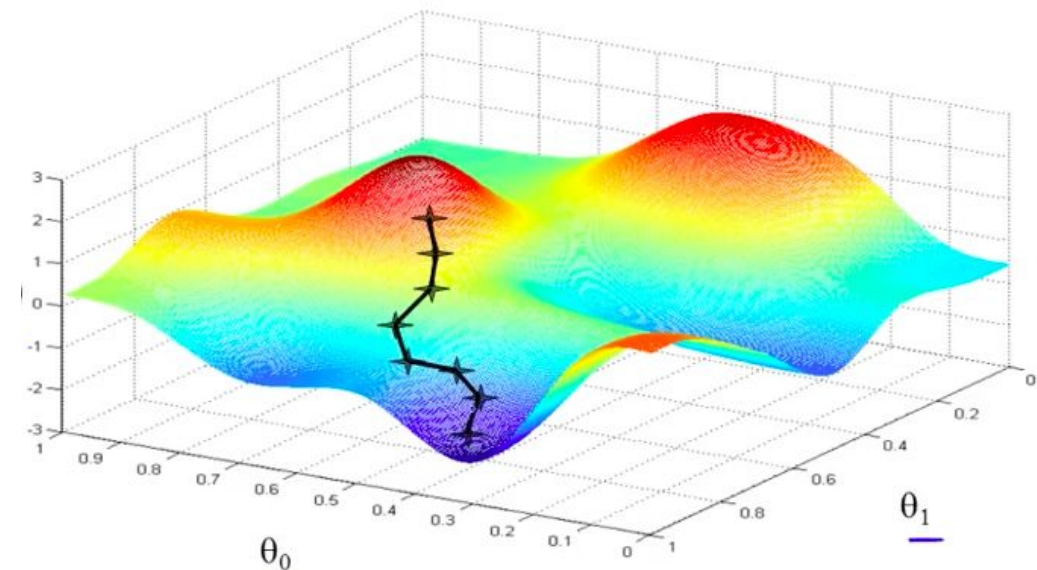
while $m < \text{max_iters}$:

$$\boldsymbol{\theta}^{(m+1)} = \boldsymbol{\theta}^{(m)} - \frac{\partial}{\partial \boldsymbol{\theta}} L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}^{(m)}), \mathbf{y})$$

$m += 1$

$$\Delta = L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}^{(m)}), \mathbf{y}) - L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}^{(m-1)}), \mathbf{y})$$

if $\Delta < \epsilon$: break



Why study backpropagation?

- Parameter estimation in a statistical model is optimization

$$\min_{\boldsymbol{\theta}} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y})$$

- Consider gradient descent

init $\boldsymbol{\theta}^{(0)}$; $m = 0$

while $m < \text{max_iters}$:

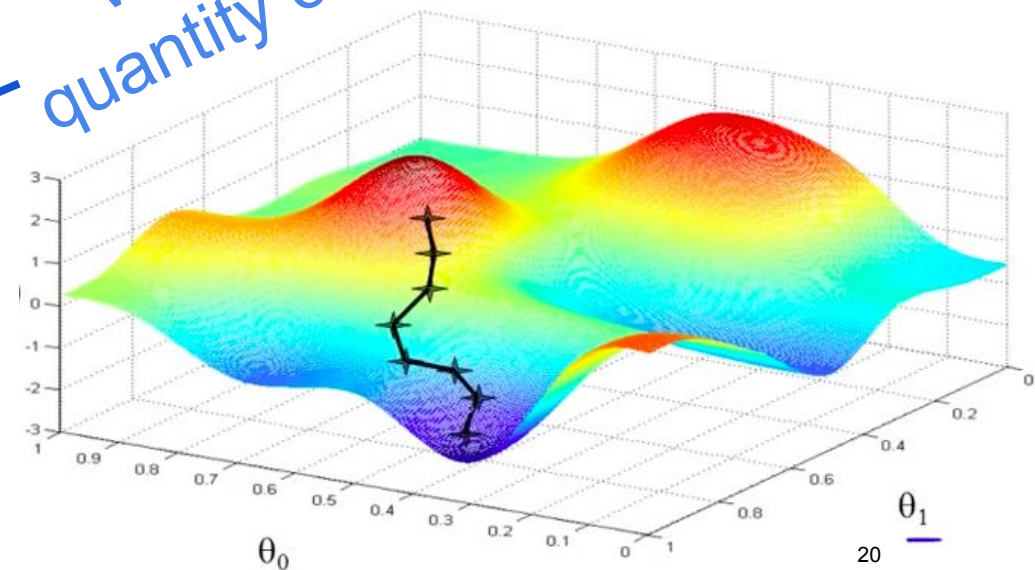
$$\boldsymbol{\theta}^{(m+1)} = \boldsymbol{\theta}^{(m)} - \frac{\partial}{\partial \boldsymbol{\theta}} L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}^{(m)}), \mathbf{y})$$

$m += 1$

$$\Delta = L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}^{(m)}), \mathbf{y}) - L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}^{(m-1)}), \mathbf{y})$$

if $\Delta < \epsilon$: break

Where did this
quantity come from?



Why study backpropagation?

- For a composite function \mathbf{f} , e.g., a neural network, $\frac{\partial}{\partial \theta} L(\mathbf{f}(\mathbf{x}; \theta), \mathbf{y})$ might be time-consuming to derive by hand
- Backpropagation is an all-purpose algorithm to the rescue!

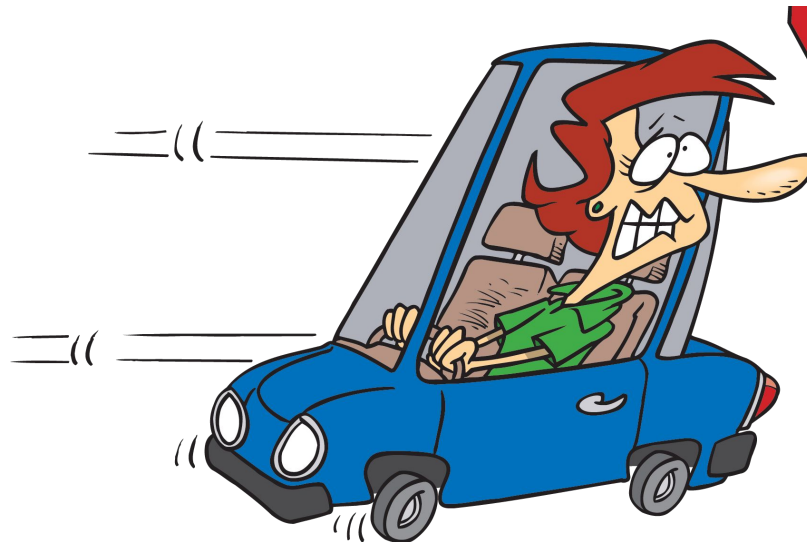


Backpropagation: What is it *really*?

Automatic Differentiation

Backpropagation: What is it *really*?

Reverse-Mode Automatic Differentiation



Backpropagation: What is it *really*?

Big Picture:

- Backpropagation (a.k.a. reverse-mode AD) is a popular technique that exploits the composite nature of complex functions to compute $\frac{\partial}{\partial \theta} L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y})$ efficiently

More Detail:

- Backpropagation is another name for reverse-mode automatic differentiation (“autodiff”).
- It recursively applies the chain rule along a computation graph to calculate the gradients of all inputs and intermediate variables efficiently using dynamic programming

Backpropagation: What is it *really*?

Big Picture:

- Backpropagation (a.k.a. reverse-mode AD) is a popular technique that exploits the composite nature of complex functions to compute $\frac{\partial}{\partial \theta} L(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}), \mathbf{y})$ efficiently

More Detail:

Theorem: *Reverse-mode automatic differentiation can compute the gradient in the same time complexity as computing \mathbf{f} !*

Calculus Background

Derivatives: Scalar Case

- Derivatives measures *change* in a function over values of a variable. Specifically, the *instantaneous rate of change*.
- In the scalar case, given a differentiable function $f: \mathbb{R} \rightarrow \mathbb{R}$, the derivative of f at a point $x \in \mathbb{R}$ is defined as:

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

where f is said to be differentiable at x if such a limit exists. Generally, this simply requires that f be smooth and continuous at x .

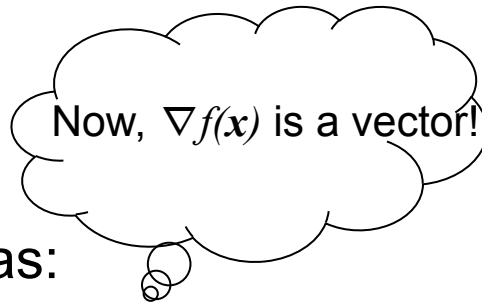
- For notational ease, the derivative of $y = f(x)$ with respect to x is commonly written as $\frac{\partial y}{\partial x}$

Derivatives: Scalar Case

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

- Hand-wavey: if x were to change by ε then y (where $y = f(x)$) would change by approximately $\varepsilon \cdot f'(x)$
- More Rigorously: $f'(x)$ is the slope of the tangent line to the graph of f at x . The tangent line is the **best linear approximation** of the function near x .
 - We can then use $f(x) \approx f(x_0) + f'(x_0)(x - x_0)$ as a locally linear approximation of f at x for some x_0

Gradients: Multivariate Case



Now, $\nabla f(\mathbf{x})$ is a vector!

- Given a function $f: \mathbb{R}^n \rightarrow \mathbb{R}$, the derivative of f at a point $\mathbf{x} \in \mathbb{R}^n$ is defined as:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left(\frac{\partial}{\partial x_1} f(\mathbf{x}), \dots, \frac{\partial}{\partial x_n} f(\mathbf{x}) \right)$$

where $\frac{\partial}{\partial x_i} f(\mathbf{x})$ is the (partial) derivative of f with respect to x_i

- This partial derivative tells us the approximate amount by which $f(\mathbf{x})$ will change if we move \mathbf{x} along the i th coordinate axis.
- For notational ease, we can again take $y = f(\mathbf{x})$ and similarly we have

$$\frac{\partial y}{\partial \mathbf{x}} = \left(\frac{\partial y}{\partial x_1}, \dots, \frac{\partial y}{\partial x_n} \right)$$

Jacobians: Multivariate Case

- Given a function $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$, for input $\mathbf{x} \in \mathbb{R}^n$ and output $\mathbf{y} = f(\mathbf{x}) \in \mathbb{R}^m$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \dots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \dots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

- The above $m \times n$ matrix (known as the Jacobian) reflects the relationship between each element of \mathbf{x} and each element of \mathbf{y} . I.e., the (i, j) -th element of $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ tells us the amount by which y_i will change if x_j is changed by a small amount.

The Multivariate Chain Rule

- Given variables x, y, z : knowing the instantaneous rate of change of z relative to y and that of y relative to x allows one to calculate the instantaneous rate of change of z relative to x :

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

- This relationship holds in the multivariate case (i.e., when $\mathbf{x}, \mathbf{y}, \mathbf{z}$ are vectors and $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ and $\frac{\partial \mathbf{z}}{\partial \mathbf{y}}$ are Jacobians). Consequently, we can form the Jacobian $\frac{\partial \mathbf{z}}{\partial \mathbf{x}}$ where

$$\frac{\partial z_k}{\partial x_i} = \sum_{j=1}^m \frac{\partial z_k}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

Computation Graphs and Slow Gradients

Composite Functions

- An ordered series of (non-linear) equations.
- Each is only a function of the preceding equations

Ex:

$$f(x, y, z) = z + \sin(x^2 + y \times \exp(z))$$

- We can represent the above equation using intermediate variables:

$$a = x^2$$

$$b = \exp(z)$$

$$c = y \times b$$

$$d = a + c$$

$$e = \sin(d)$$

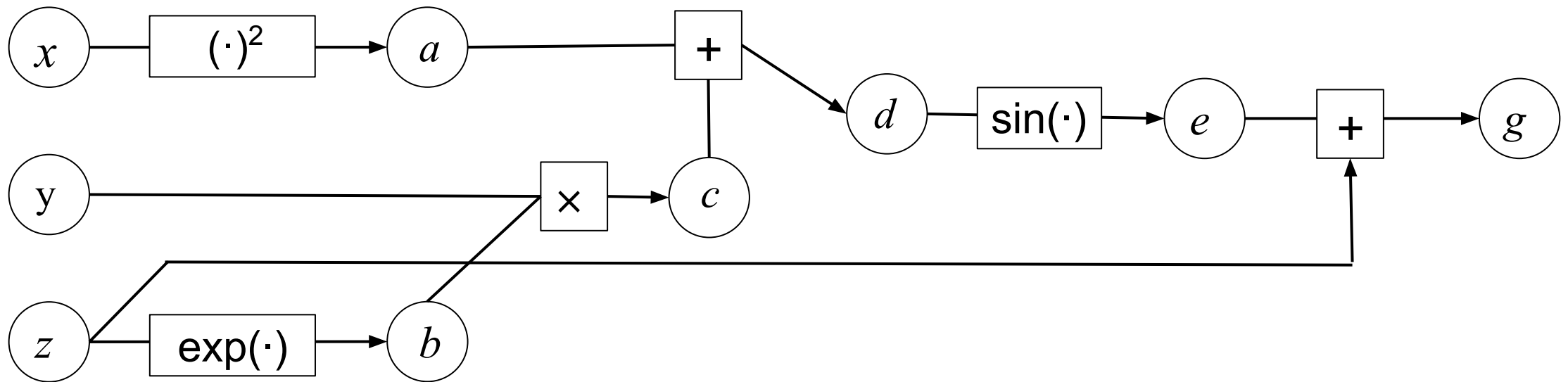
$$g = e + z$$

where $f(x, y, z) = g$

Functions as Computation Graphs

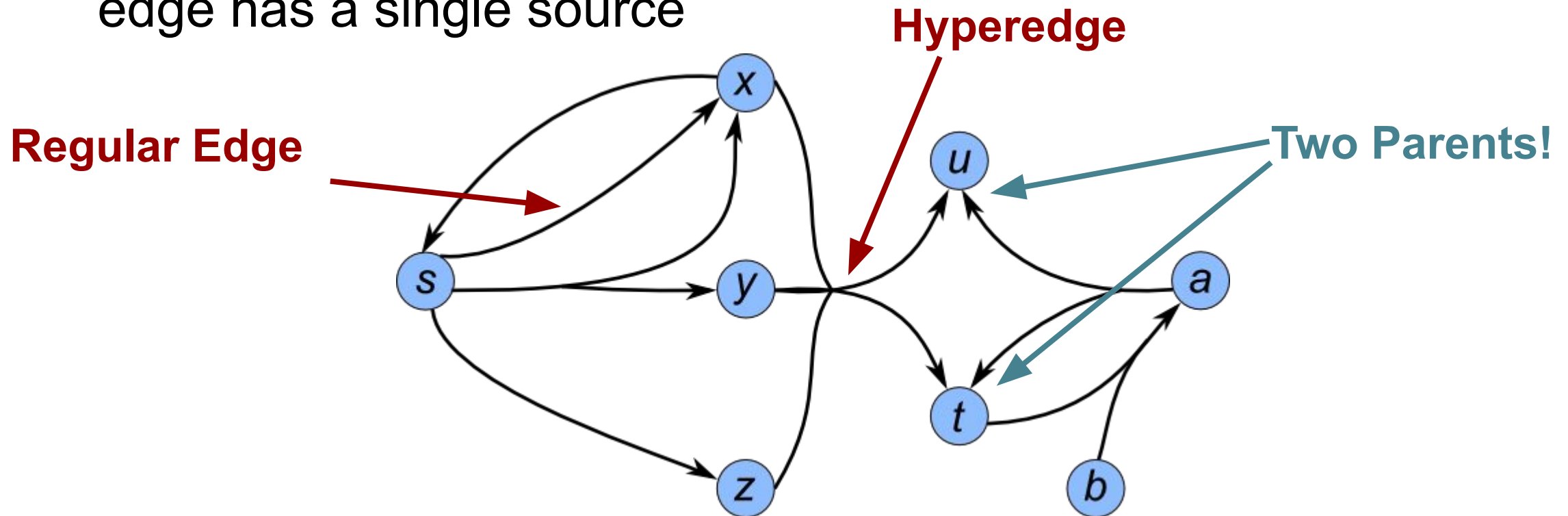
- Any composite function can be described in terms of its computation graph.
- Formally, a computation graph is a **labeled, directed acyclic hypergraph** $G = (V, E)$ where each node is a variable and each hyperedge is labeled with a function.

$$f(x, y, z) = z + \sin(x^2 + y \times \exp(z))$$



What is a hypergraph?

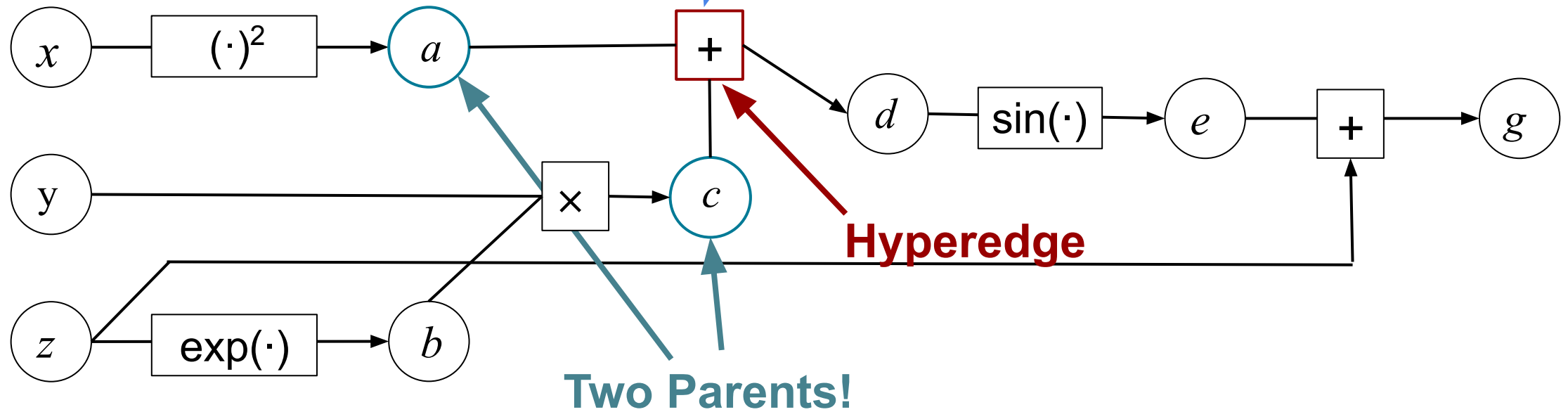
- Sounds fancy, eh? It's really simple!
- A hypergraph relaxes the assumption in a graph that every edge has a single source



Why do we need a labeled hypergraph?

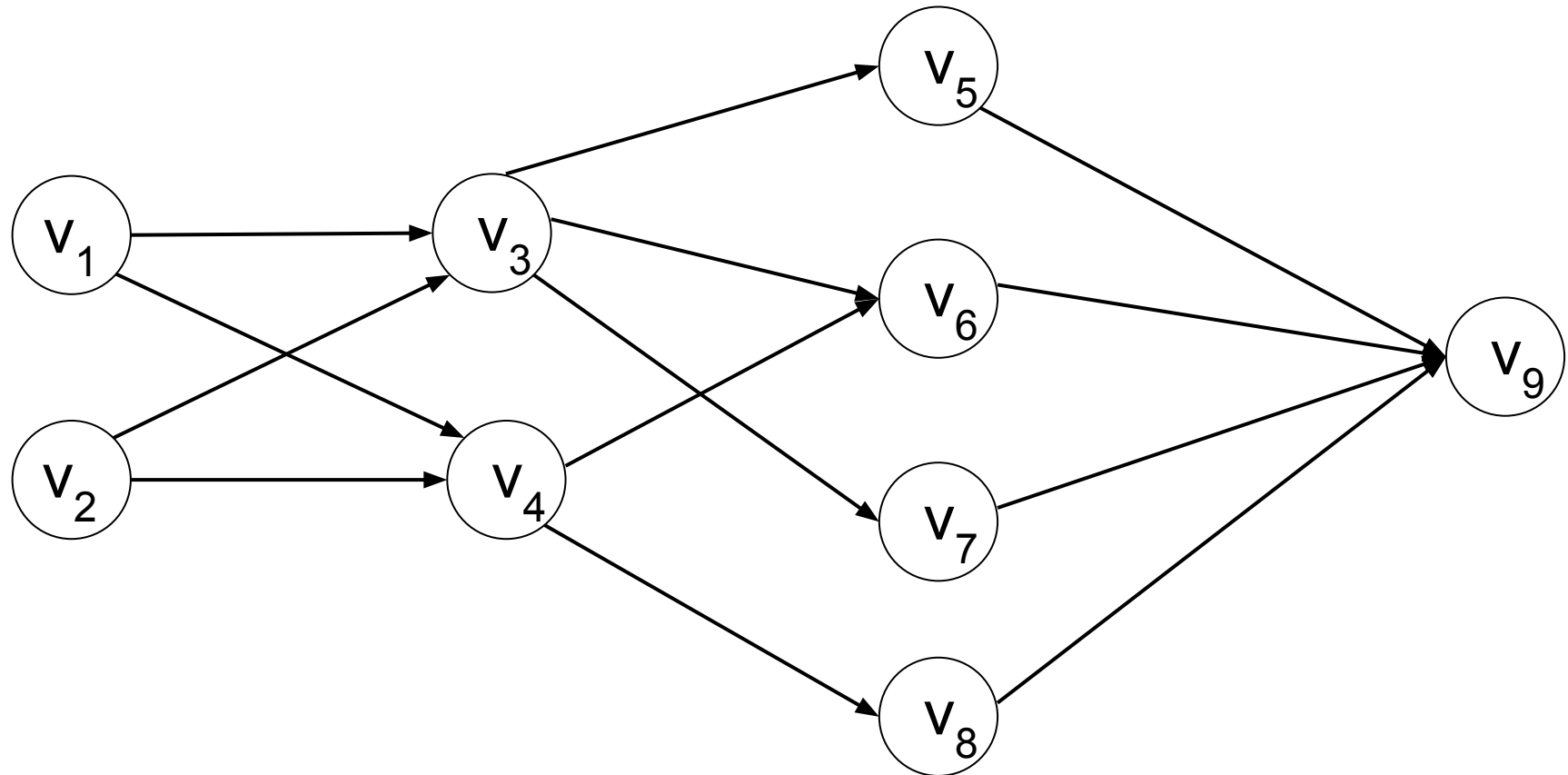
The hyperedge's label is
an arithmetic operation

$$f(x, y, z) = z + \sin(x^2 + y \times \exp(z))$$



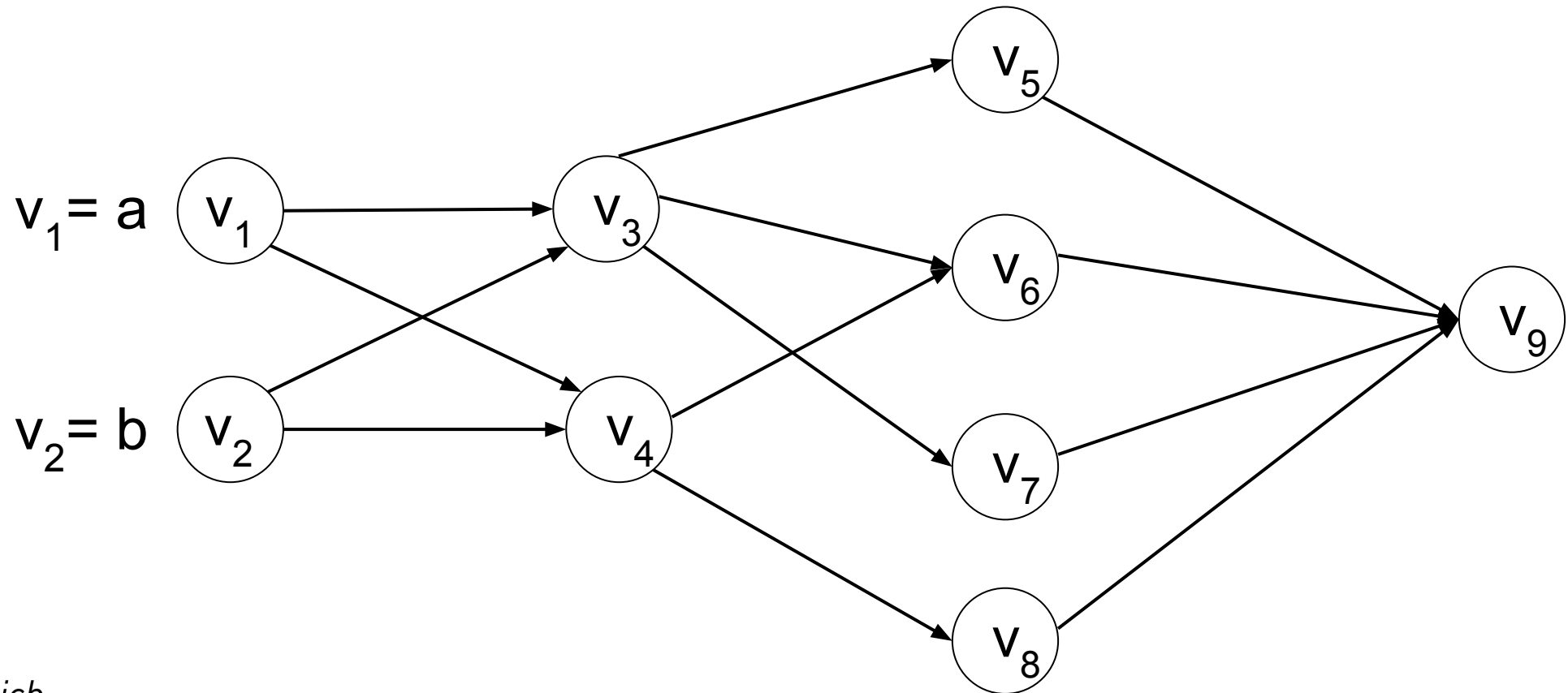
Paths of Influence

- In a composite function there are many “paths of influence”



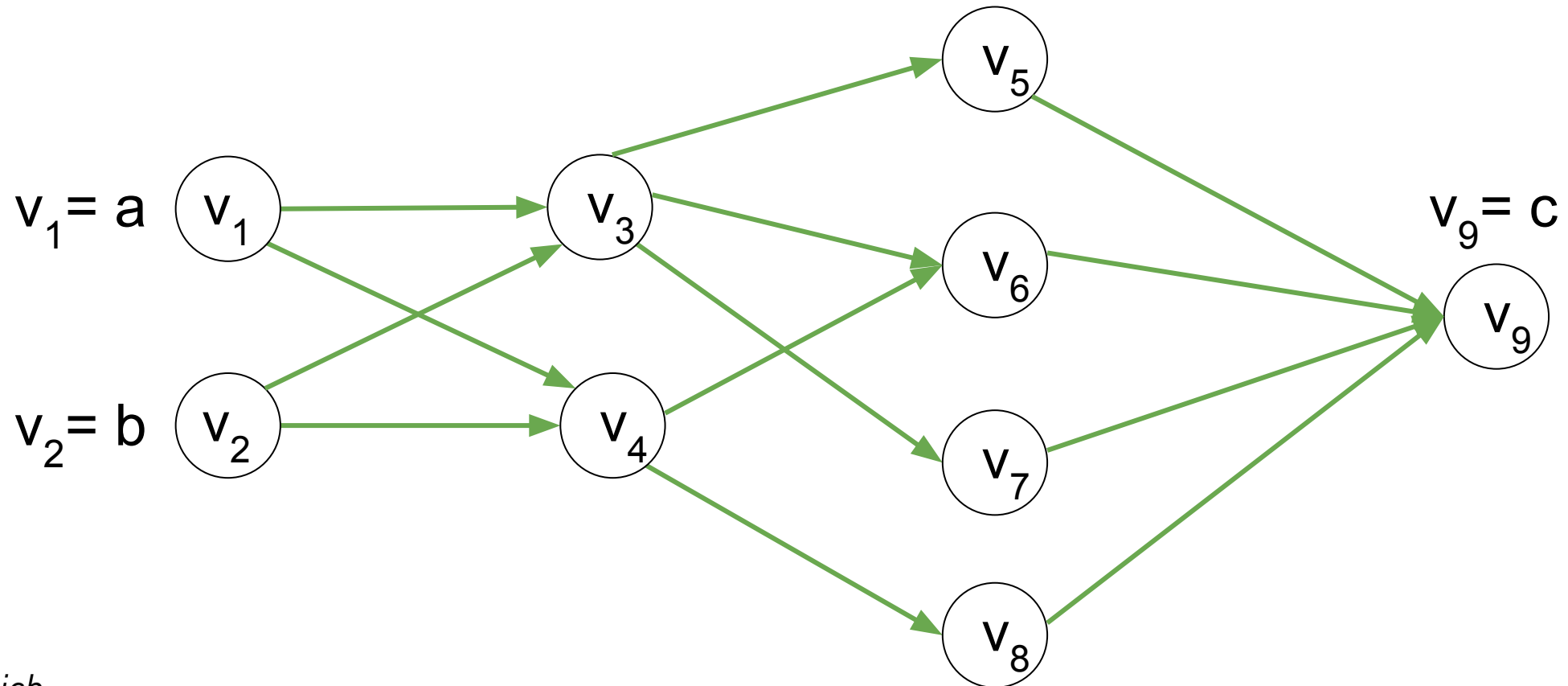
Paths of Influence

- In a composite function there are many “paths of influence”



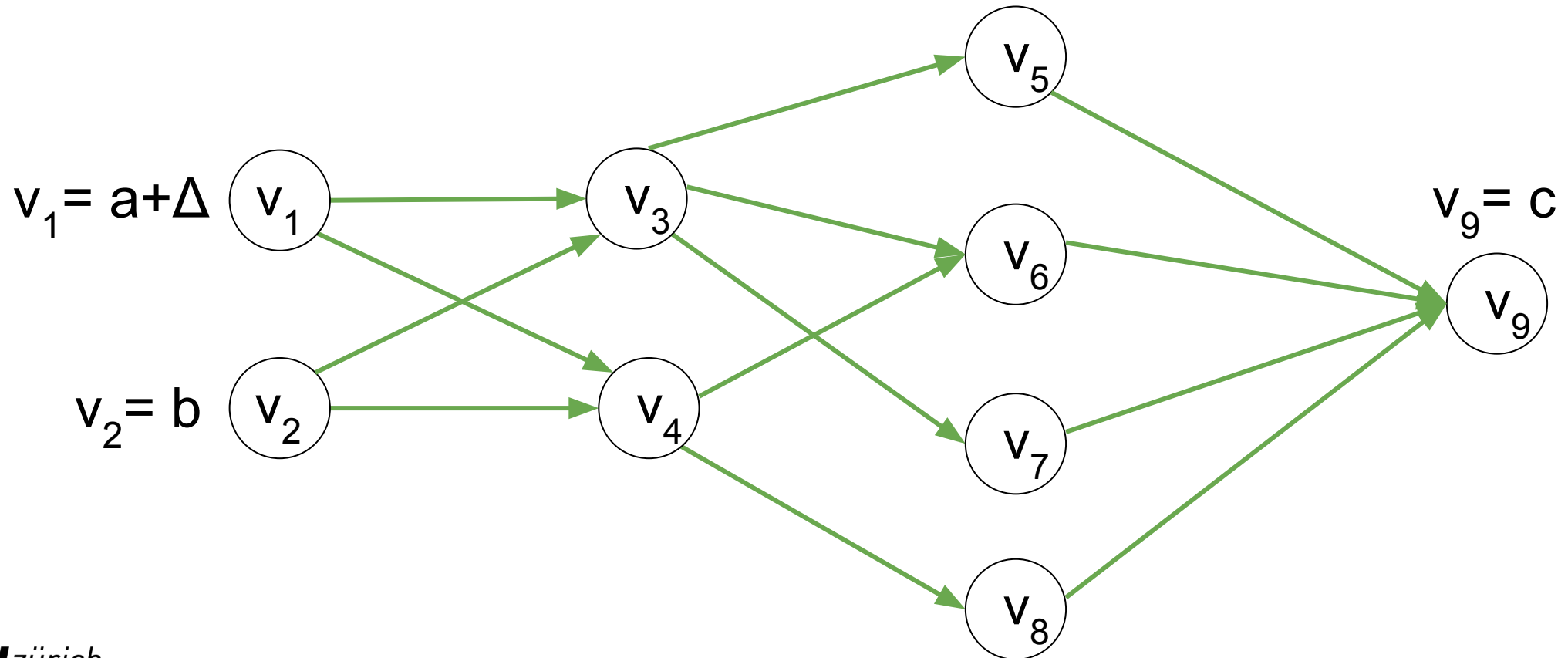
Paths of Influence

- In a composite function there are many “paths of influence”



Paths of Influence

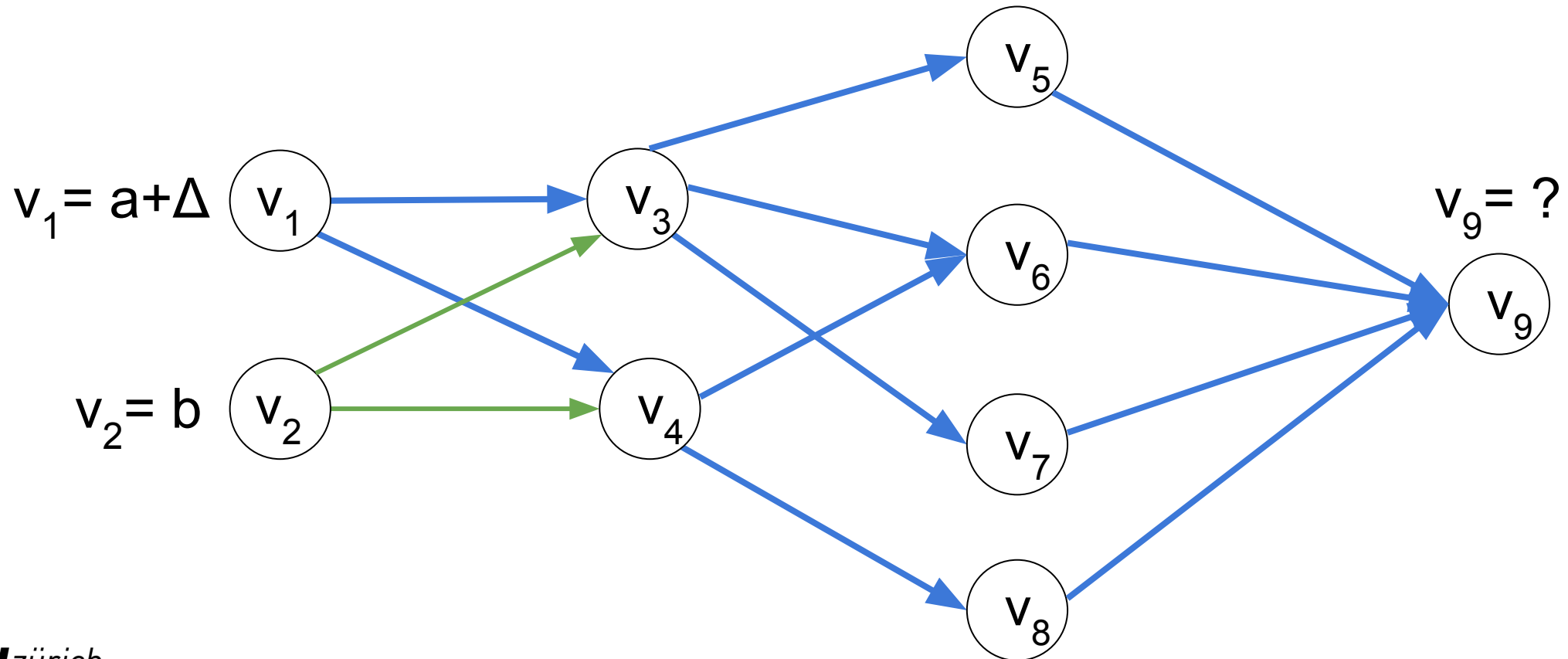
- In a composite function there are many “paths of influence”



Paths of Influence

- The derivative is a sum over all of the paths:

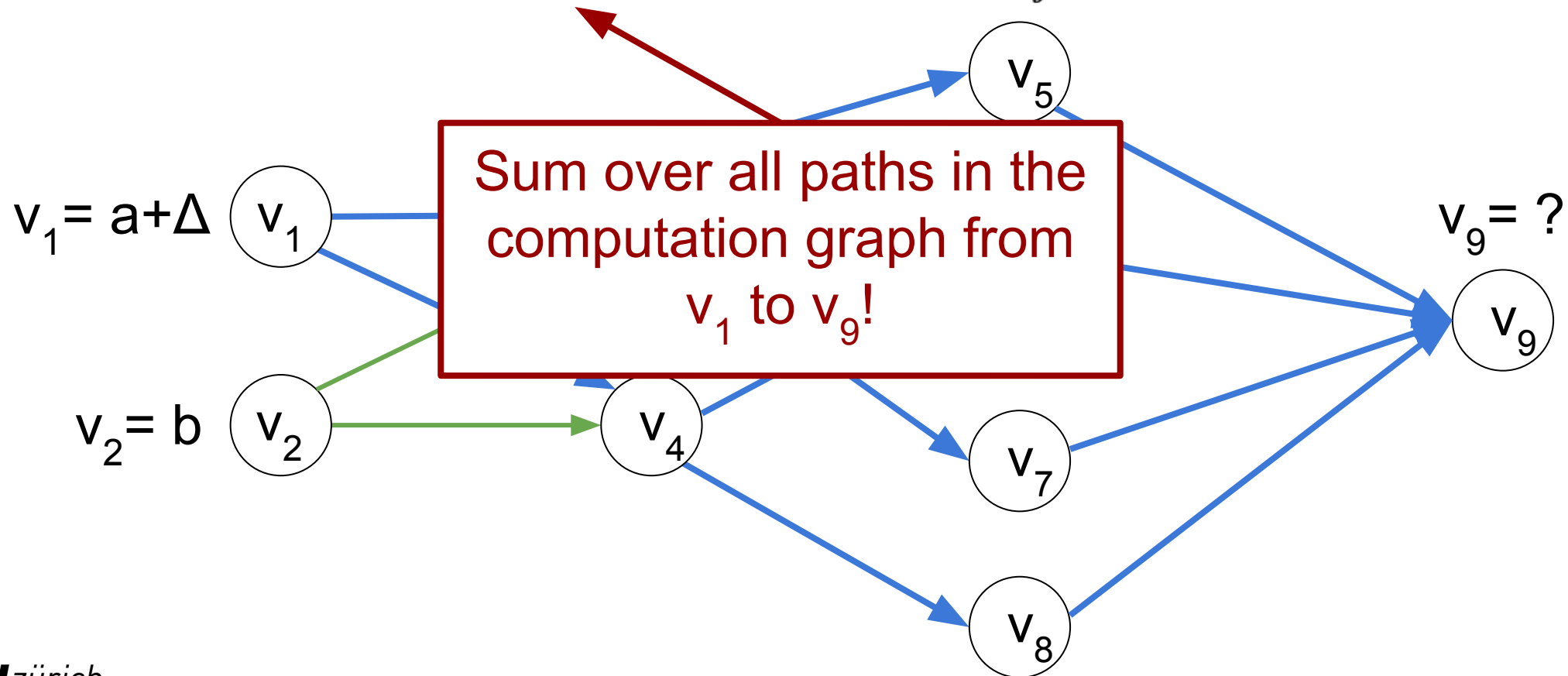
$$\frac{\partial v_9}{\partial v_1} = \sum_{v_1, v_i, \dots, v_j, v_9} \frac{\partial v_i}{\partial v_1} \dots \frac{\partial v_9}{\partial v_j}$$



Paths of Influence

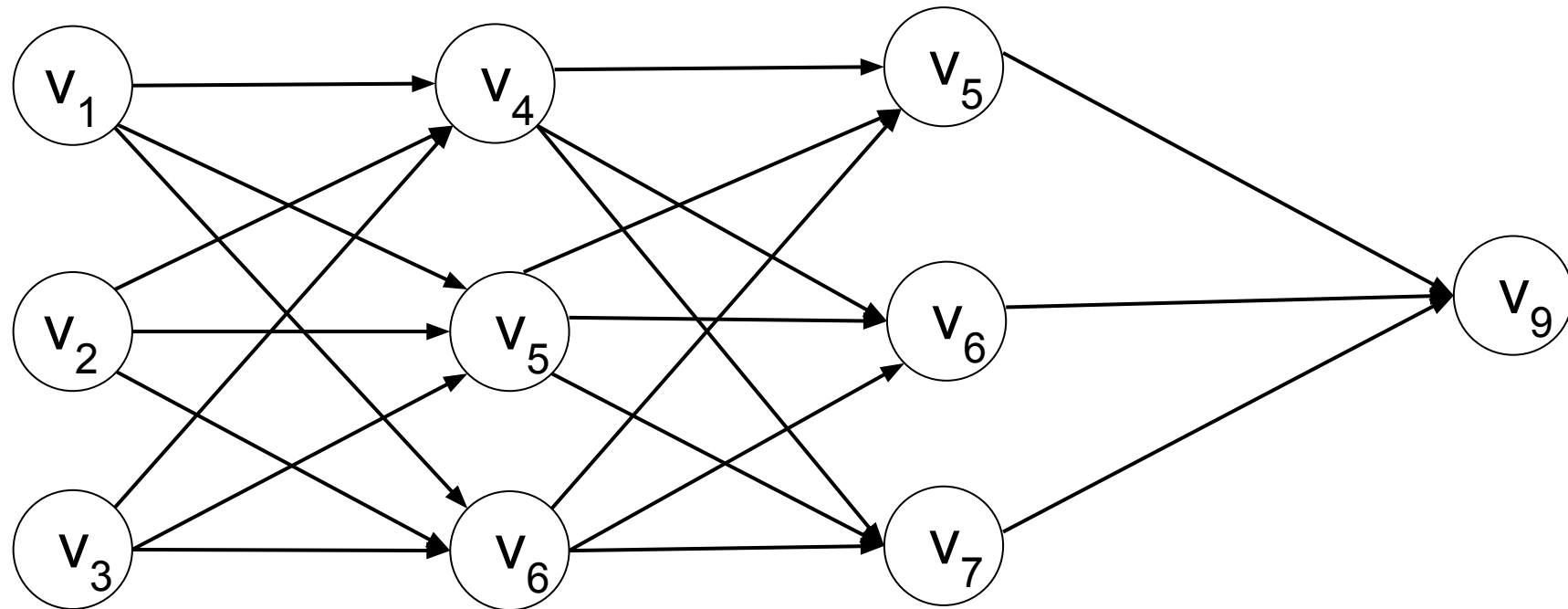
- The derivative is a sum over all of the paths of influence:

$$\frac{\partial v_9}{\partial v_1} = \sum_{v_1, v_i, \dots, v_j, v_9} \frac{\partial v_i}{\partial v_1} \dots \frac{\partial v_9}{\partial v_j}$$



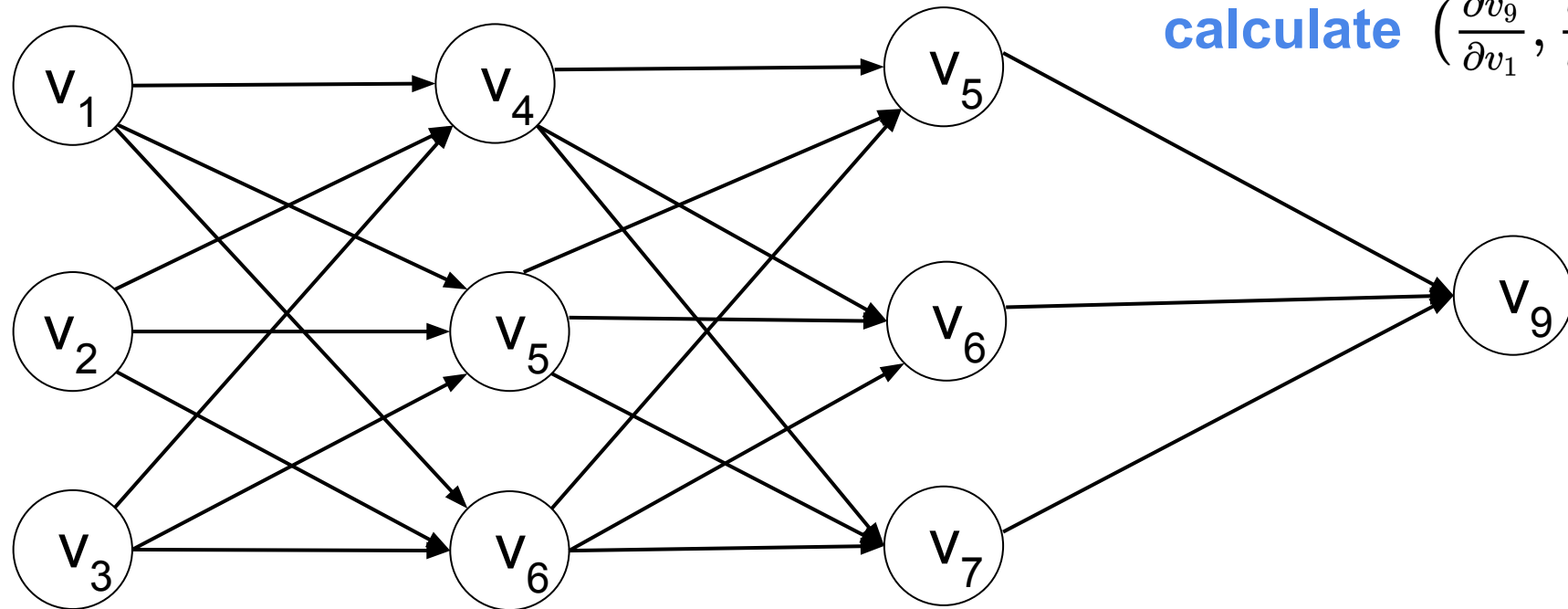
How many paths could there be?

- How bad is this naive gradient computation algorithm?
- Consider a “fully connected” computation graph:



How many paths could there be?

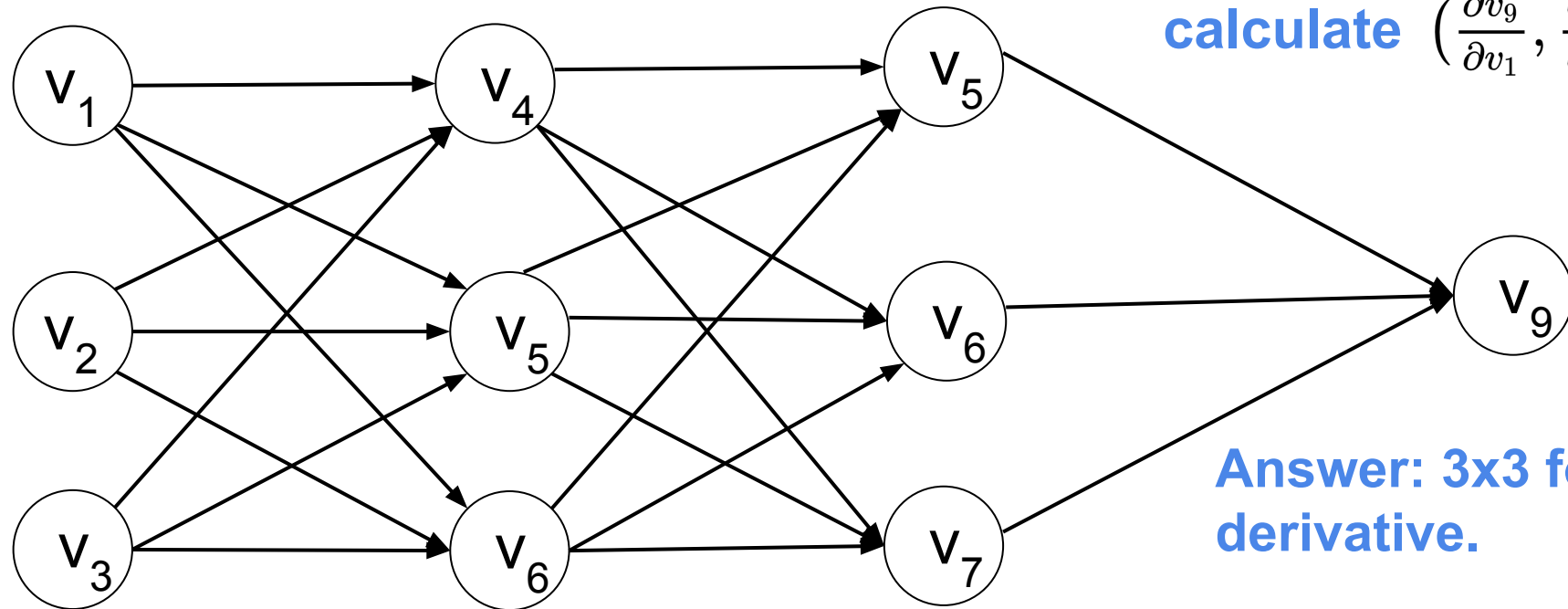
- How bad is this naive gradient computation algorithm?
- Consider a “fully connected” computation graph:



How many paths must we sum over in order to calculate $\left(\frac{\partial v_9}{\partial v_1}, \frac{\partial v_9}{\partial v_2}, \frac{\partial v_9}{\partial v_3}\right)$?

How many paths could there be?

- How bad is this naive gradient computation algorithm?
- Consider a “fully connected” computation graph:



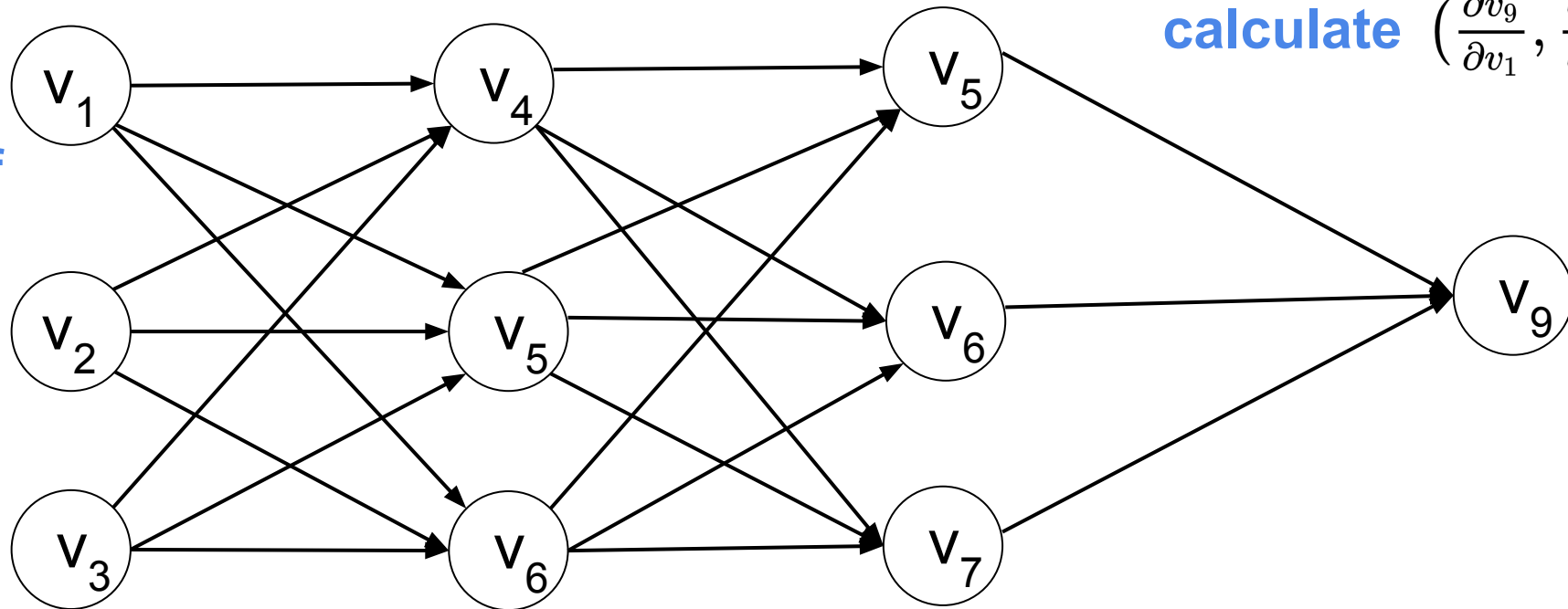
How many paths must we sum over in order to calculate $\left(\frac{\partial v_9}{\partial v_1}, \frac{\partial v_9}{\partial v_2}, \frac{\partial v_9}{\partial v_3}\right)$?

Answer: 3x3 for each derivative.

How many paths could there be?

- How bad is this naive gradient computation algorithm?
- Consider a “fully connected” computation graph:

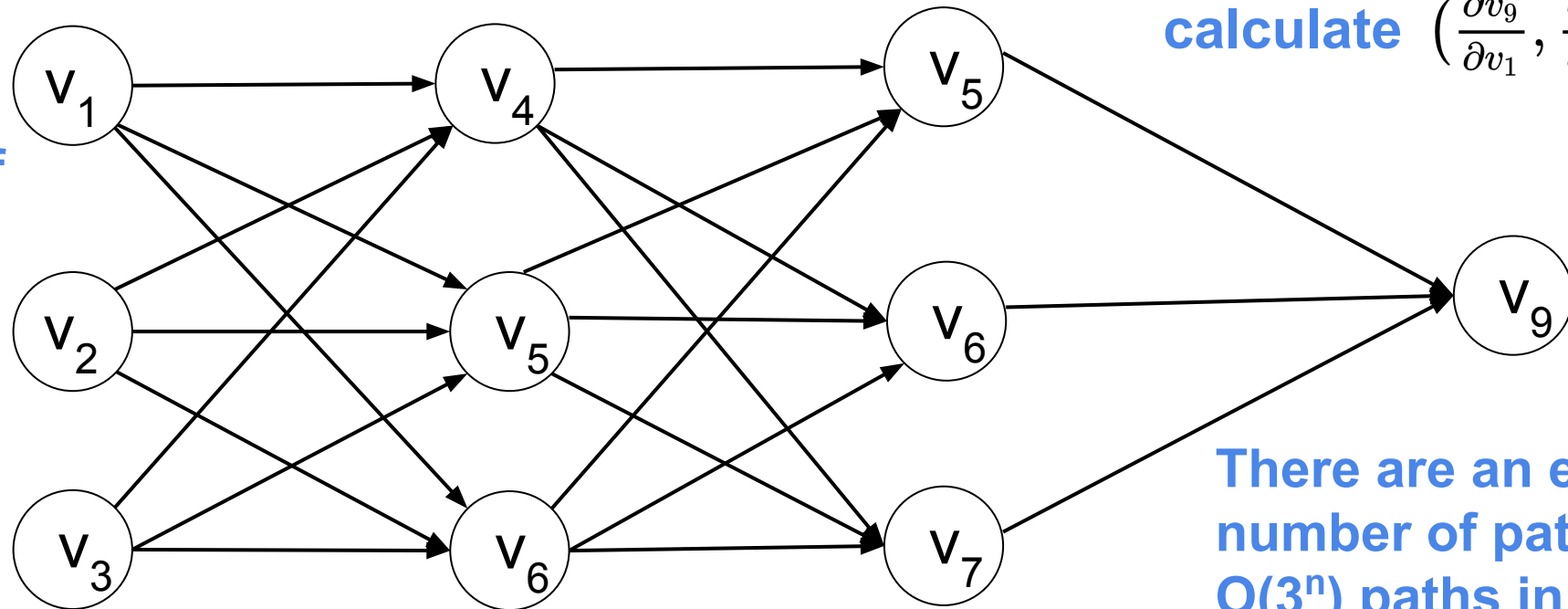
How many paths must we sum over in order to calculate $\left(\frac{\partial v_9}{\partial v_1}, \frac{\partial v_9}{\partial v_2}, \frac{\partial v_9}{\partial v_3}\right)$?



What happens if we add another “layer?”

How many paths could there be?

- How bad is this naive gradient computation algorithm?
- Consider a “fully connected” computation graph:



How many paths must we sum over in order to calculate $\left(\frac{\partial v_9}{\partial v_1}, \frac{\partial v_9}{\partial v_2}, \frac{\partial v_9}{\partial v_3}\right)$?

What happens if we add another “layer?”

There are an exponential number of paths! We have $O(3^n)$ paths in this case

How many paths could there be?

- If you apply the chain rule naively, your algorithm will run in exponential time!
- This problem has the same structure as finding the shortest path!
- So, if you wanted to find the shortest path in a graph, would you
 - (a) Enumerate all of the exponentially many paths and select the shortest one?
 - (b) Run a linear-time dynamic-programming algorithm?

How many paths could there be?

- If you apply the chain rule naively, your algorithm will run in exponential time!
- This problem has the same structure as the shortest-path!
- So, if you wanted to find the shortest path in a graph, would you
 - (a) Enumerate all of the exponentially many paths and select the shortest one?
 - (b) Run a linear-time dynamic-programming algorithm?

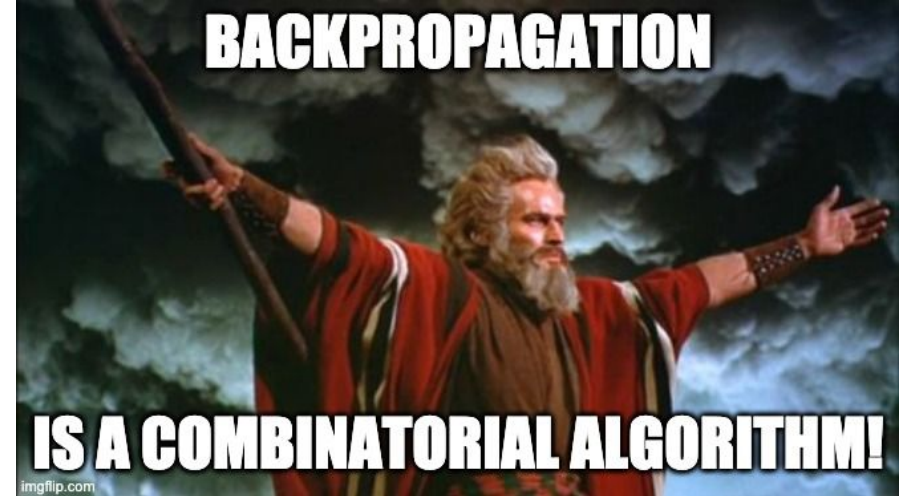
The Magic of Backpropagation

- Backpropagation also runs fast even though it explores an exponentially large space!
- Just as with the shortest path problem, backpropagation is also a dynamic program
- Other relatives you will see in this course
 - minimum edit distance
 - Cocke-Kasami-Younger



The Magic of Backpropagation

- Backpropagation also runs fast even though it explores an exponentially large space!
- Just as with the shortest path problem, backpropagation is also a **linear-time** dynamic program
- Other relatives you will see in this course
 - minimum edit distance
 - Cocke-Kasami-Younger



The Nitty-Gritty of Backpropagation (a.k.a. Reverse-Mode Automatic Differentiation)

Automatic Differentiation

- Main idea behind AD: as long as we have access to the derivatives of a set of primitives, e.g. the derivative of $\cos(x)$ is $-\sin(x)$, then we can stitch these together to get the derivative of any composite function
- Saving the values of intermediate variables (dynamic programming!) allows for low computational complexity. Indeed, we go from exponential down to linear
- The one drawback is that we require knowledge of how the function was built out of primitives and cannot treat it as a true black box

General Automatic Differentiation Framework

Set of primitives

$$f(y, z) = y + z$$

$$f(y, z) = y \times z$$

$$f(y) = y^3$$

$$f(y) = \sin(y)$$

$$f(y) = \exp(y)$$

$$f(y) = \log(y)$$

And their derivatives

$$\frac{\partial}{\partial x} f(y, z) = \frac{\partial y}{\partial x} + \frac{\partial z}{\partial x}$$

$$\frac{\partial}{\partial x} f(y, z) = y \frac{\partial z}{\partial x} + z \frac{\partial y}{\partial x}$$

$$\frac{\partial}{\partial x} f(y) = 3y^2 \frac{\partial y}{\partial x}$$

$$\frac{\partial}{\partial x} f(y) = \cos(y) \frac{\partial y}{\partial x}$$

$$\frac{\partial}{\partial x} f(y) = \exp(y) \frac{\partial y}{\partial x}$$

$$\frac{\partial}{\partial x} f(y) = \frac{1}{y} \frac{\partial y}{\partial x}$$

General Automatic Differentiation Framework

Step 1: Write down a composite function as a hypergraph with intermediate variables as nodes and hyperedges are labeled with the primitives

*Again, why a hypergraph? An intermediate variable may be a function of **more than one** preceding intermediate variable.*

Step 2: Given a set of inputs, perform a forward pass through the graph to compute the function's value; this is called forward propagation

Step 3: Run backpropagation on the graph using the stored forward values. This computes the derivative

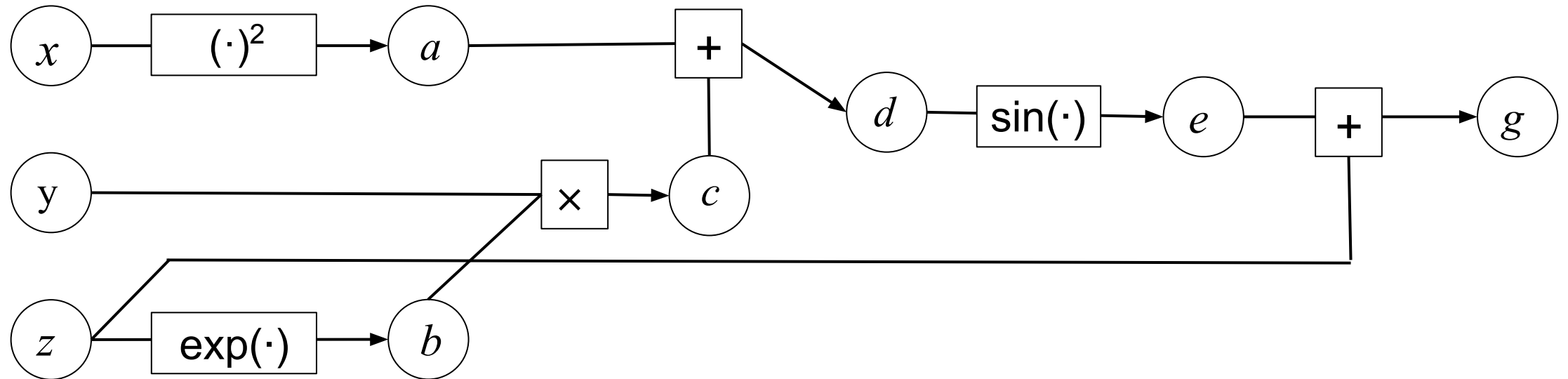
Automatic Differentiation: Forward Propagation

- Perform a “forward pass” through the computation graph where the value of each node is calculated based on its ancestors, which computes the value of $f(\cdot)$
- Not to be confused with “forward-mode differentiation”
 - Backpropagation is a synonym for reverse-mode differentiation
 - You can also do one-pass AD, but it’s generally slower for the functions we care about

Automatic Differentiation: Forward Propagation

Example:

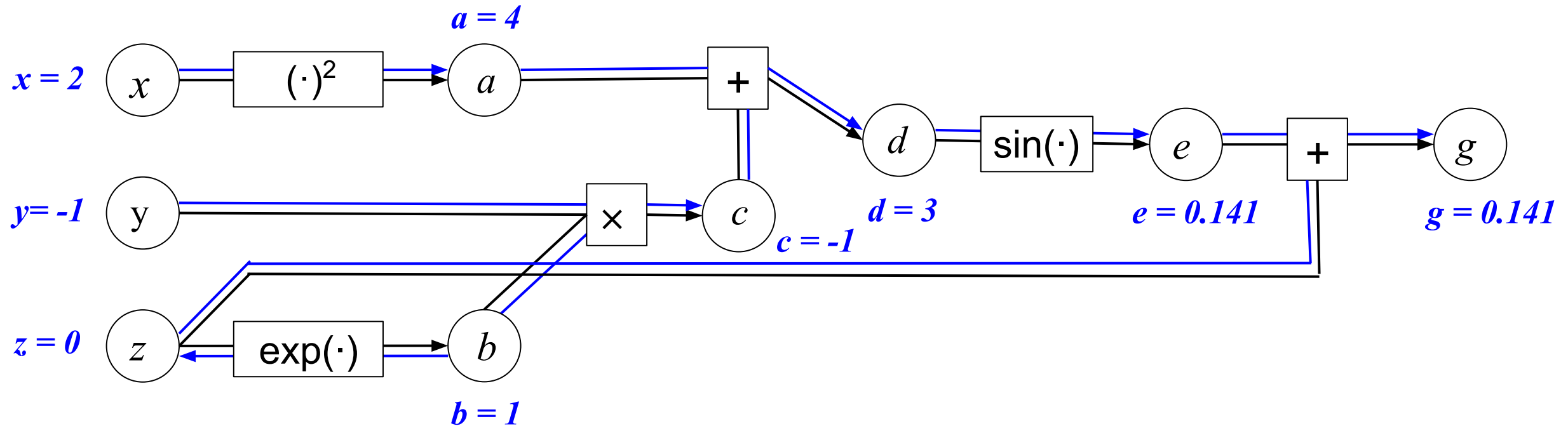
$$f(x, y, z) = z + \sin(x^2 + y \times \exp(z))$$



Automatic Differentiation: Forward Propagation

Example:

$$f(x, y, z) = z + \sin(x^2 + y \times \exp(z))$$



$$f(2, -1, 0) = g = 0.141$$

Automatic Differentiation: Forward Propagation

- Input a function f encoded as a labeled, directed acyclic hypergraph with N edges and labels p_i (for primitives) on the hyperarcs
- Assume the edges are topologically sorted so $i < j$ implies v_i is before v_j
- We assume the first n nodes are input nodes and set to \mathbf{x}
- We use bracket notation $\langle \rangle$ to represent an ordered set

forward-propagate($f, \mathbf{x} \in \mathbb{R}^n$)

$$v_i \leftarrow \begin{cases} x_i & \text{if } i \leq n \\ 0 & \text{otherwise} \end{cases}$$

for $i = N - n, \dots, N$:

$$v_i \leftarrow p_i(\langle v_{\text{Pa}(i)} \rangle)$$

return $[v_1, \dots, v_N]$

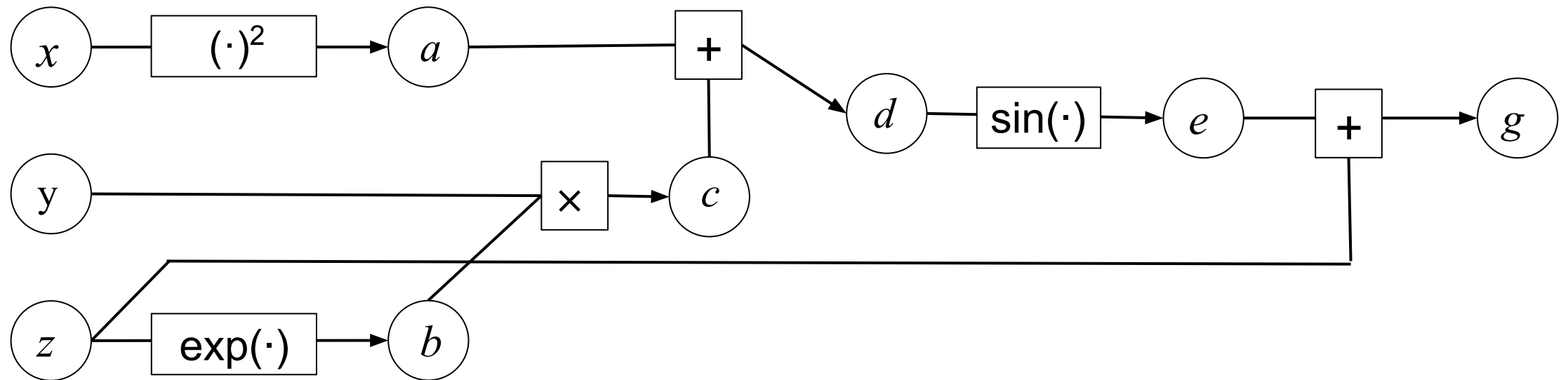
Automatic Differentiation: Backward Propagation

- The “differentiation” component of our framework
- Computing for derivative of the output with respect to intermediate variables including the input
- This is also known as reverse-mode differentiation

Automatic Differentiation: Backward Propagation

Example:

$$f(x, y, z) = z + \sin(x^2 + y \times \exp(z))$$

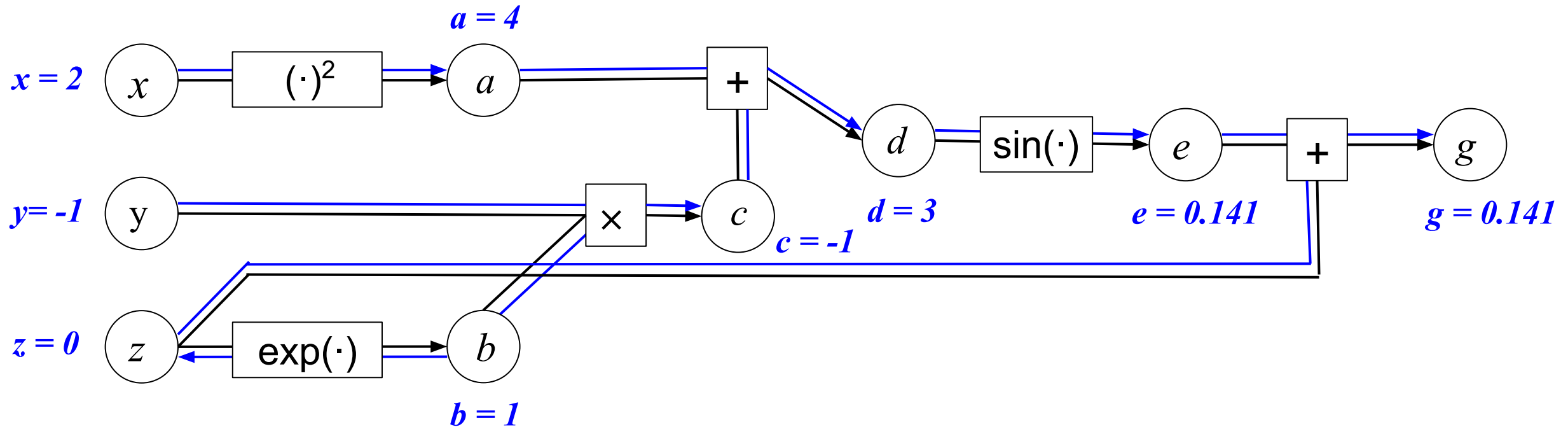


Automatic Differentiation: Backward Propagation

Example:

Perform forward propagation!

$$f(x, y, z) = z + \sin(x^2 + y \times \exp(z))$$



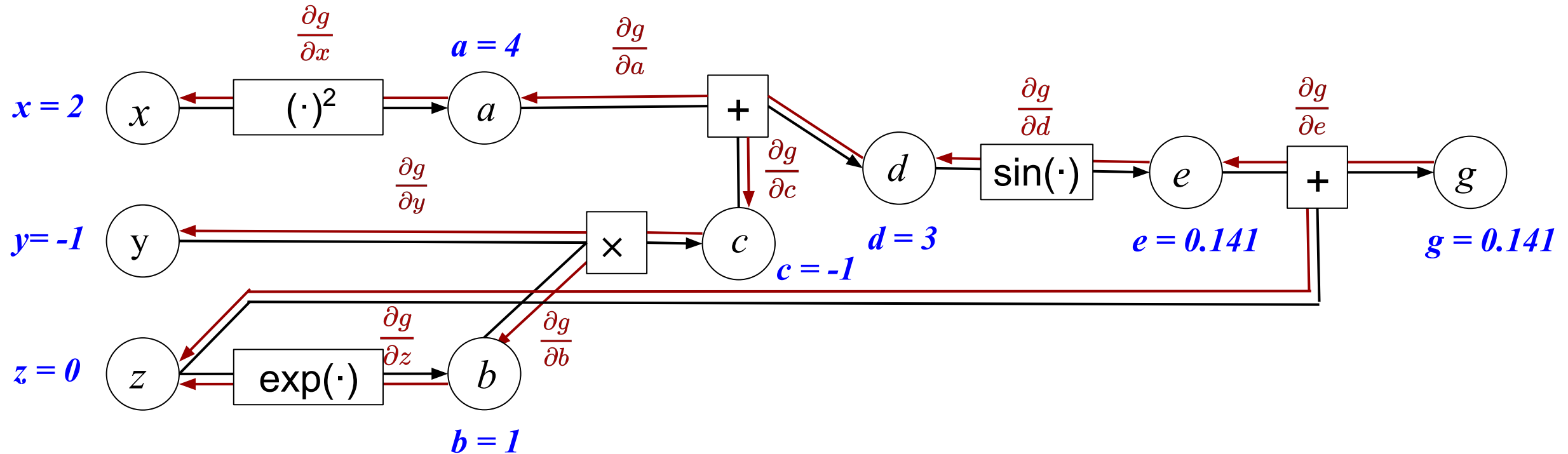
$$f(2, -1, 0) = g = 0.141$$

Automatic Differentiation: Backward Propagation

Example:

Compute values of intermediate derivatives!

$$f(x, y, z) = z + \sin(x^2 + y \times \exp(z))$$

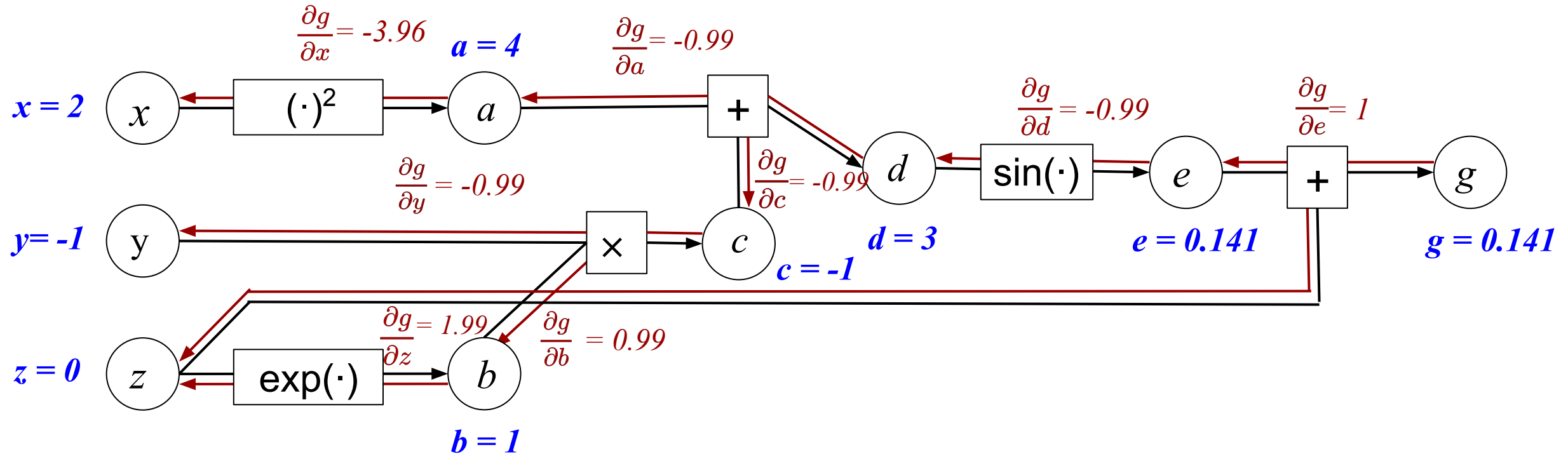


Automatic Differentiation: Backward Propagation

Example:

Compute values of intermediate derivatives!

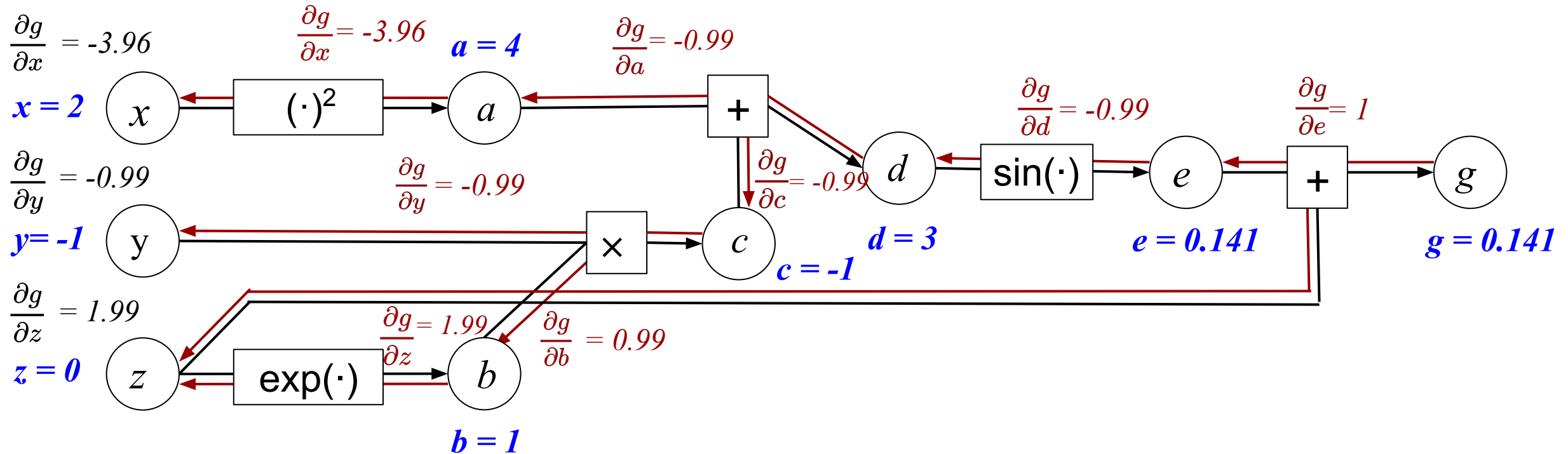
$$f(x, y, z) = z + \sin(x^2 + y \times \exp(z))$$



Automatic Differentiation: Backward Propagation

Example:

$$f(x, y, z) = z + \sin(x^2 + y \times \exp(z))$$



$$\nabla f(2, -1, 0) = \left\langle \frac{\partial g}{\partial x}, \frac{\partial g}{\partial y}, \frac{\partial g}{\partial z} \right\rangle = \langle -3.96, -0.99, 1.99 \rangle$$

Calculating Gradients

$$f(x, y, z) = z + \sin(x^2 + y \times \exp(z))$$

Example: $\frac{\partial g}{\partial e} = 1$

$$\frac{\partial g}{\partial d} = \frac{\partial g}{\partial e} \frac{\partial e}{\partial d} = \frac{\partial g}{\partial e} \cos(d)$$

$$\frac{\partial g}{\partial c} = \frac{\partial g}{\partial d} \frac{\partial d}{\partial c} = \frac{\partial g}{\partial d} 1$$

$$\frac{\partial g}{\partial b} = \frac{\partial g}{\partial c} \frac{\partial c}{\partial b} = \frac{\partial g}{\partial c} y$$

$$\frac{\partial g}{\partial z} = \frac{\partial g}{\partial b} \frac{\partial b}{\partial z} + 1 = \frac{\partial g}{\partial b} \exp(z) + 1$$

- We can easily write down the derivatives of individual terms in the graph
- Given all these, we can work backwards to compute the derivative of $g = f(x, y, z)$ with respect to each variable: a simple application of the chain rule!

Automatic Differentiation: Backward Propagation

- Input a function f encoded as a labeled, directed acyclic hypergraph with N edges and labels p_i (for primitives) on the hyperarcs
- Assume the edges are topologically sorted so $i < j$ implies v_i is before v_j
- We assume the first n nodes are input nodes and set to \mathbf{x}
- We use bracket notation $\langle \rangle$ to represent an ordered set

forward-propagate($f, \mathbf{x} \in \mathbb{R}^n$)

$$v_i \leftarrow \begin{cases} x_i & \text{if } i \leq n \\ 0 & \text{otherwise} \end{cases}$$

for $i = N - n, \dots, N$:

$$v_i \leftarrow p_i(\langle v_{\text{Pa}(i)} \rangle)$$

return $[v_1, \dots, v_N]$

back-propagate($f, \mathbf{x} \in \mathbb{R}^n$)

$\mathbf{v} \leftarrow \text{forward-propagate}(f, \mathbf{x})$

$$\frac{\partial f}{\partial v_i} \leftarrow 0, \quad \forall i \in \{1, \dots, N\}$$

for $i = N, \dots, 1$:

$$\frac{\partial f}{\partial v_i} \leftarrow \sum_{j: i \in \text{Pa}(j)} \frac{\partial f}{\partial v_j} \frac{\partial}{\partial v_i} p_j(\langle v_{\text{Pa}(j)} \rangle)$$

return $\left[\frac{\partial f}{\partial v_1}, \dots, \frac{\partial f}{\partial v_N} \right]$

$$\frac{\partial}{\partial v_i} p_j(\langle v_{\text{Pa}(j)} \rangle) = \frac{\partial v_j}{\partial v_i}$$

Automatic Differentiation: Backward Propagation

- Input a function f encoded as a labeled, directed acyclic hypergraph with N edges and labels p_i (for primitives) on the hyperarcs
- Assume the edges are topologically sorted so $i < j$ implies v_i is before v_j
- We assume the first n nodes are input nodes and set to \mathbf{x}
- We use bracket notation $\langle \rangle$ to represent an ordered set

forward-propagate($f, \mathbf{x} \in \mathbb{R}^n$)

$$v_i \leftarrow \begin{cases} x_i & \text{if } i \leq n \\ 0 & \text{otherwise} \end{cases}$$

for $i = N - n, \dots, N$:

$$v_i \leftarrow p_i(\langle v_{\text{Pa}(i)} \rangle)$$

return $[v_1, \dots, v_N]$

back-propagate($f, \mathbf{x} \in \mathbb{R}^n$)

$\mathbf{v} \leftarrow \text{forward-propagate}(f, \mathbf{x})$

$$\frac{\partial f}{\partial v_i} \leftarrow 0, \quad \forall i \in \{1, \dots, N\}$$

for $i = N, \dots, 1$:

$$\frac{\partial f}{\partial v_i} \leftarrow \sum_{j: i \in \text{Pa}(j)} \frac{\partial f}{\partial v_j} \frac{\partial}{\partial v_i} p_j(\langle v_{\text{Pa}(j)} \rangle)$$

return $\left[\frac{\partial f}{\partial v_1}, \dots, \frac{\partial f}{\partial v_N} \right]$

$$\frac{\partial}{\partial v_i} p_j(\langle v_{\text{Pa}(j)} \rangle) = \frac{\partial v_j}{\partial v_i}$$

base case for output node:

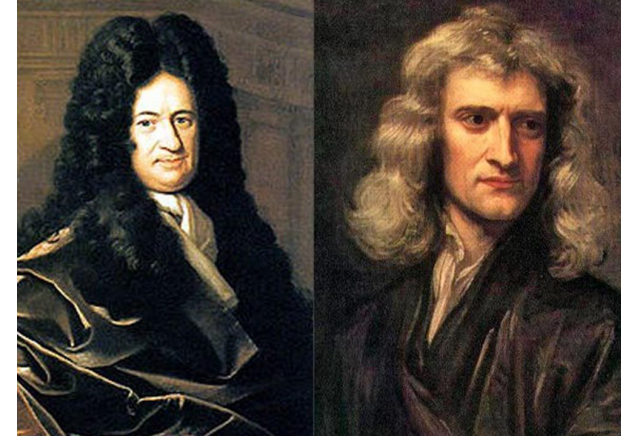
$$\frac{\partial f}{\partial v_N} = 1$$

So, why isn't backprop just the chain rule?

- Automatic differentiation works *because* of the chain rule
 - It is part of the proof of correctness of the algorithm
- Evaluating $\frac{\partial y}{\partial x}$ is provably as fast as evaluating $y = f(x)$
 - Use of intermediate variables (i.e., a, b, c, \dots) means resulting computation for the gradient has the same structure as the original function
 - Not necessarily (or usually) the case when the chain rule is used in symbolic differentiation!
- Autodiff can differentiate algorithms, not just expressions
 - Code for $\frac{\partial y}{\partial x}$ can be derived by a rote program transformation, even if the code has control flow structures like loops and intermediate variables

Analyzing Runtime of Backprop

- Enumerating all paths of influence takes $O(2^n)$ time where n is the number of nodes
- With dynamic programming, we can speed this up to $O(n)$
 - The same analysis as the shortest-path problem
- This is why backprop is **computer science** and not just calculus
 - Neither Newton nor Leibniz talked about runtime!
- Next time your friend says backprop is just the chain rule, you can retort:
 - Actually, it's an algorithm that propagates the chain rule through complex expressions efficiently by using dynamic programming



Three Types of Differentiation on your Computer

- **Symbolic Differentiation:**

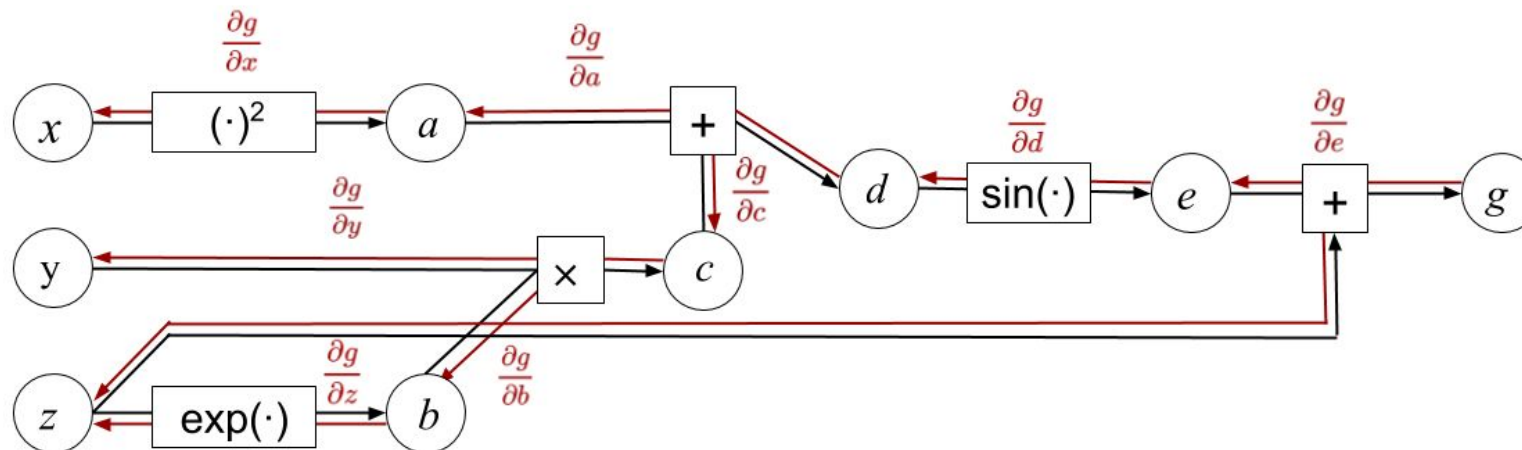
$$\frac{\partial f(x,y,z)}{\partial z} = 1 + \cos(x^2 + y \times \exp(z)) \times y \times \exp(z)$$

- **Numerical Differentiation:**

- The finite-difference approximation

$$\frac{\partial f(x,y,z)}{\partial z} \approx \frac{f(x,y,z+h) - f(x,y,z)}{h}$$

- **Automatic Differentiation (backpropagation falls under here):**



Three Types of Differentiation on your Computer

- **Symbolic Differentiation:**

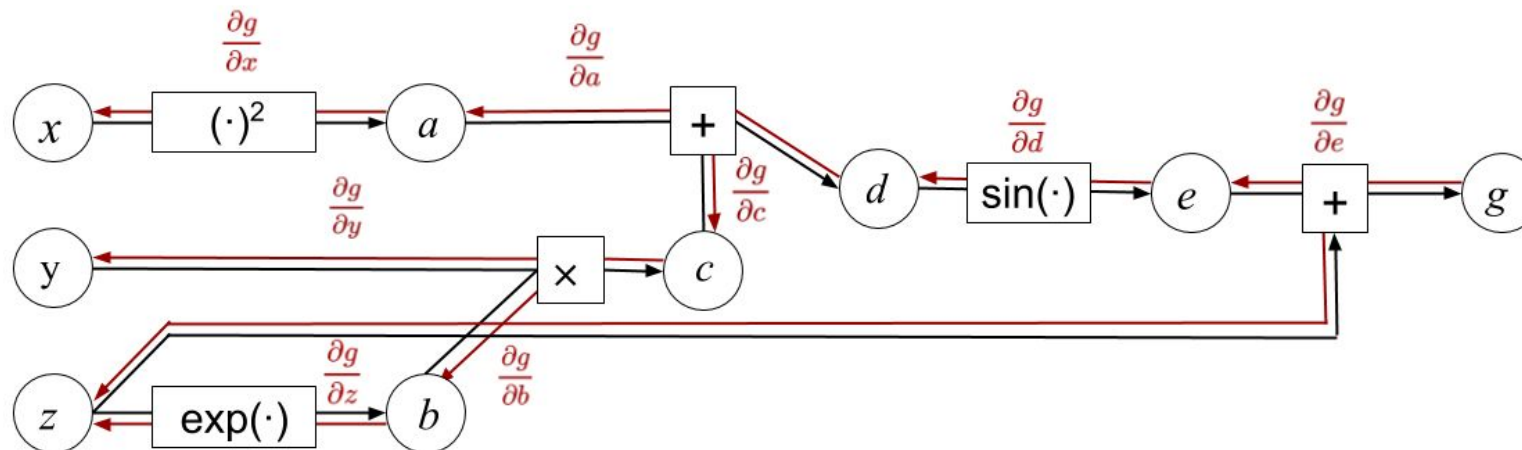
$$\frac{\partial f(x,y,z)}{\partial z} = 1 + \cos(x^2 + y \times \exp(z)) \times y \times \exp(z)$$

- **Numerical Differentiation:**

- The finite-difference approximation

$$\frac{\partial f(x,y,z)}{\partial z} \approx \frac{f(x,y,z+h) - f(x,y,z)}{h}$$

- **Automatic Differentiation (backpropagation falls under here):**



Three Types of Differentiation on your Computer

- **Symbolic Differentiation:**

$$\frac{\partial f(x,y,z)}{\partial z} = 1 + \cos(x^2 + y \times \exp(z)) \times y \times \exp(z)$$

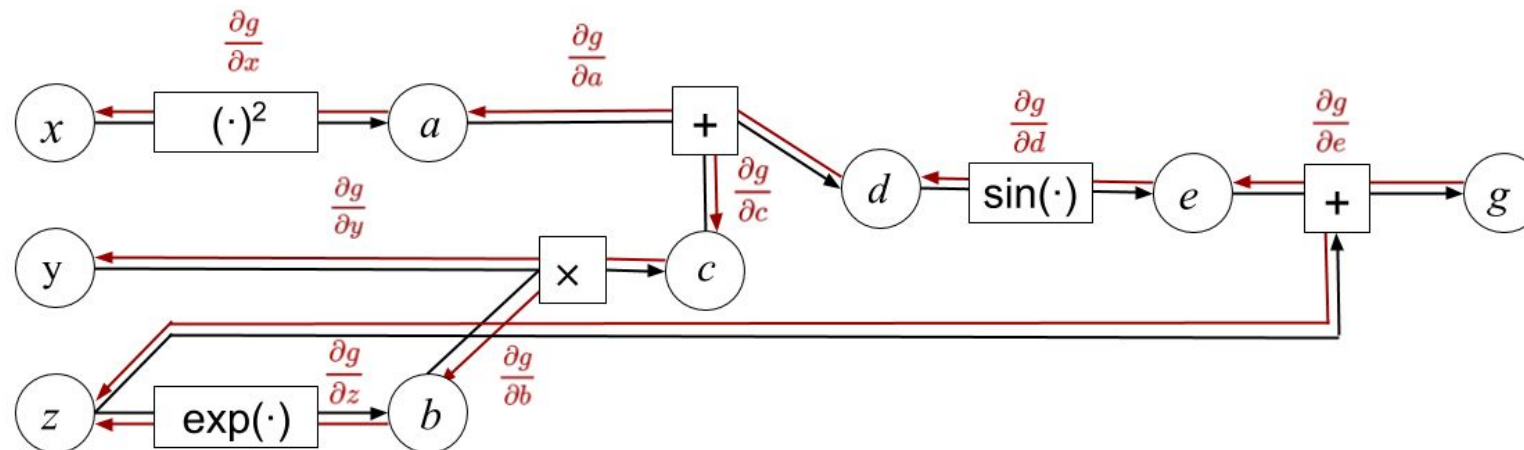
- **Numerical Differentiation:**

- The finite-difference approximation

$$\frac{\partial f(x,y,z)}{\partial z} \approx \frac{f(x,y,z+h) - f(x,y,z)}{h}$$

← Much, much slower in general

- **Automatic Differentiation (backpropagation falls under here):**



Three Types of Differentiation on your Computer

- **Symbolic Differentiation:**

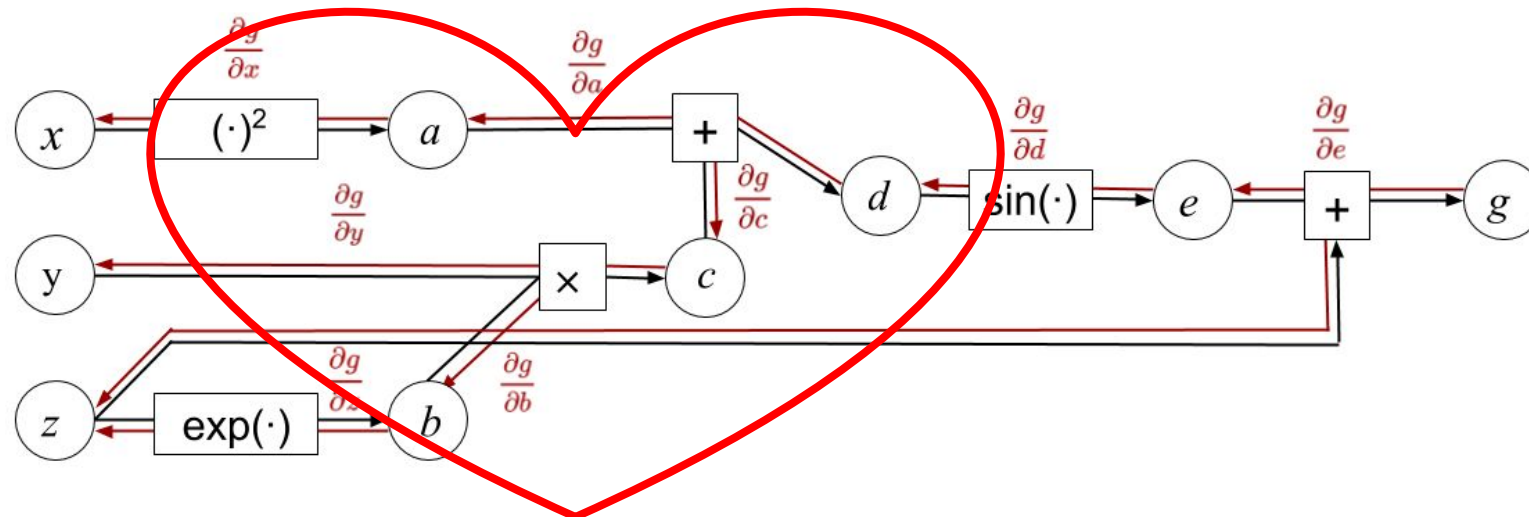
$$\frac{\partial f(x,y,z)}{\partial z} = 1 + \cos(x^2 + y \times \exp(z)) \times y \times \exp(z)$$

- **Numerical Differentiation:**

- The finite-difference approximation

$$\frac{\partial f(x,y,z)}{\partial z} \approx \frac{f(x,y,z+h) - f(x,y,z)}{h}$$

- **Automatic Differentiation (backpropagation falls under here):**



A Fun Interpretation of Backprop as Optimization (Optional Bonus Section)

Interpretation as Optimization Problem

- Take the intermediate variables in our computational graph (v_1, \dots, v_N) as simple equality constraints for a constrained optimization problem.

Example:

$$f(x, y, z) = z + \sin(x^2 + y \times \exp(z)) \longrightarrow$$

$$\operatorname{argmax}_x g$$

$$\text{s.t. } a = x^2$$

$$b = \exp(z)$$

$$c = y \times b$$

$$d = a + c$$

$$e = \sin(d)$$

$$g = e + z$$

Interpretation as Optimization Problem

- Take the intermediate variables in our computational graph (v_I, \dots, v_N) as simple equality constraints for a constrained optimization problem.

General Case:

- Input a function f encoded as a labeled, directed acyclic hypergraph with N edges and labels p_i (for primitives) on the hyperarcs
- Assume the edges are topologically sorted so $i < j$ implies v_i is before v_j
- We assume the first n nodes are input nodes and set to \mathbf{x}

$$\begin{array}{ll} \text{argmax}_{\mathbf{x}} v_N \\ \text{s.t.} & v_i = x_i \quad \text{for } 1 \leq i \leq n \\ & v_i = p_i(\langle v_{\text{Pa}(i)} \rangle) \quad \text{for } n < i \leq N \end{array}$$

Interpretation as Optimization Problem

- Using the standard method for solving constrained optimization problems—with Lagrange multipliers—we can exactly recover the intermediate derivatives in the backprop algorithm.

Derivation:

Interpretation as Optimization Problem

- Using the standard method for solving constrained optimization problems—with Lagrange multipliers—we can exactly recover the intermediate derivatives in the backprop algorithm.

Derivation:

- Input a function f encoded as a labeled, directed acyclic hypergraph with N edges and labels p_i (for primitives) on the hyperarcs
- Assume the edges are topologically sorted \rightarrow so $i < j$ implies v_i is before v_j
- We assume the first n nodes are input nodes and set to \mathbf{x}

$$\operatorname{argmax}_{\mathbf{x}} v_N$$

$$\text{s.t.} \quad \begin{aligned} v_i &= x_i && \text{for } 1 \leq i \leq n \\ v_i &= p_i(\langle v_{\text{Pa}(i)} \rangle) && \text{for } n < i \leq N \end{aligned}$$

Interpretation as Optimization Problem

- Using the standard method for solving constrained optimization problems—with Lagrange multipliers—we can exactly recover the intermediate derivatives in the backprop algorithm.

Derivation:

Optimality Condition (setting Lagrangian equal to zero):

$$\mathcal{L}(\mathbf{x}, \mathbf{v}, \boldsymbol{\lambda}) = v_N - \sum_{i=1}^N \lambda_i \cdot \left(v_i - p_i(\langle v_{\text{Pa}(i)} \rangle) \right)$$

$$\nabla \mathcal{L}(\mathbf{x}, \mathbf{v}, \boldsymbol{\lambda}) = 0$$

$$\nabla_{\lambda_i} \mathcal{L} = v_i - p_i(\langle v_{\text{Pa}(i)} \rangle) = 0 \quad \Leftrightarrow \quad v_i = p_i(\langle z_{\text{Pa}(i)} \rangle)$$

Interpretation as Optimization Problem

- Using the standard method for solving constrained optimization problems—with Lagrange multipliers—we can exactly recover the intermediate derivatives in the backprop algorithm.

Derivation:

Solving the equations

$$\begin{aligned} 0 &= \nabla_{v_j} \mathcal{L} \\ &= \nabla_{v_j} \left[v_N - \sum_{i=1}^N \lambda_i \cdot (v_i - p_i(\langle v_{\text{Pa}(i)} \rangle)) \right] \\ &= - \sum_{i=1}^N \lambda_i \nabla_{v_j} \left[(v_i - p_i(\langle v_{\text{Pa}(i)} \rangle)) \right] \\ &= - \left(\sum_{i=1}^n \lambda_i \nabla_{v_j} [v_i] \right) + \left(\sum_{i=1}^n \lambda_i \nabla_{v_j} [p_i(\langle v_{\text{Pa}(i)} \rangle)] \right) \\ &= -\lambda_j + \sum_{i:j \in \text{Pa}(i)} \lambda_i \frac{\partial p_i(\langle v_{\text{Pa}(i)} \rangle)}{\partial v_j} \\ &\iff \lambda_j = \sum_{i:j \in \text{Pa}(i)} \lambda_i \frac{\partial p_i(\langle v_{\text{Pa}(i)} \rangle)}{\partial v_j} \end{aligned}$$

Interpretation as Optimization Problem

- Using the standard method for solving constrained optimization problems—with Lagrange multipliers—we can exactly recover the intermediate derivatives in the backprop algorithm.

Derivation:

Solving the equations

$$\begin{aligned} 0 &= \nabla_{v_j} \mathcal{L} \\ &= \nabla_{v_j} \left[v_N - \sum_{i=1}^N \lambda_i \cdot (v_i - p_i(\langle v_{\text{Pa}(i)} \rangle)) \right] \\ &= - \sum_{i=1}^N \lambda_i \nabla_{v_j} \left[(v_i - p_i(\langle v_{\text{Pa}(i)} \rangle)) \right] \\ &= - \left(\sum_{i=1}^n \lambda_i \nabla_{v_j} [v_i] \right) + \left(\sum_{i=1}^n \lambda_i \nabla_{v_j} [p_i(\langle v_{\text{Pa}(i)} \rangle)] \right) \\ &= -\lambda_j + \sum_{i:j \in \text{Pa}(i)} \lambda_i \frac{\partial p_i(\langle v_{\text{Pa}(i)} \rangle)}{\partial v_j} \\ &\iff \lambda_j = \sum_{i:j \in \text{Pa}(i)} \lambda_i \frac{\partial p_i(\langle v_{\text{Pa}(i)} \rangle)}{\partial v_j} \end{aligned}$$

look familiar?

Interpretation as Optimization Problem

Recall our backprop algorithm:

back-propagate($f, \mathbf{x} \in \mathbb{R}^n$)

$\mathbf{v} \leftarrow \text{forward-propagate}(f, \mathbf{x})$

$\frac{\partial f}{\partial v_i} \leftarrow 0, \forall i \in \{1, \dots, N\}$

for $i = N, \dots, 1$:

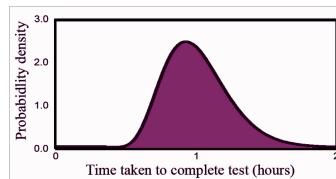
$$\frac{\partial f}{\partial v_i} \leftarrow \sum_{j: i \in \text{Pa}(j)} \frac{\partial f}{\partial v_j} \frac{\partial}{\partial v_i} p_j(\langle v_{\text{Pa}(j)} \rangle)$$

return $\left[\frac{\partial f}{\partial v_1}, \dots, \frac{\partial f}{\partial v_N} \right]$

Sneak Preview

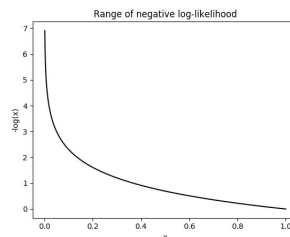
Stay tuned for more NLP (and ML) essentials

① Probability Refresher



$$p(y|x) = \frac{p(x|y)p(y)}{\int p(x|y)p(y)dy}$$

② Log-Linear Models



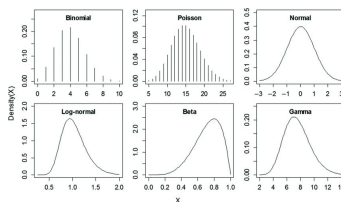
$$p(y | \mathbf{x}, \boldsymbol{\theta}) = \frac{\exp(\boldsymbol{\theta}^\top \mathbf{f}(\mathbf{x}, y))}{\sum_{y' \in \mathcal{Y}} \exp(\boldsymbol{\theta}^\top \mathbf{f}(\mathbf{x}, y'))}$$

③ Softmax Function



$$\text{Softmax}(\mathbf{h}, y, T) = \frac{\exp(h_y/T)}{\sum_{y' \in \mathcal{Y}} \exp(h_{y'}/T)}$$

④ The Exponential Family



$$p(\mathbf{x} | \boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} h(\mathbf{x}) \exp(\boldsymbol{\theta}^\top \boldsymbol{\phi}(\mathbf{x}))$$

➡ **Afterwards:** we are finally ready to do some NLP together! 😊

Conclusion

Backpropagation

- Backpropagation is a fun dynamic program that is ubiquitous in machine learning
- Most people treat backprop as a blackbox (PyTorch) ***without*** understanding how it works
 - **Life lesson:** You should understand the tools you are using!
- Backpropagation is also a constructive theorem about the computational complexity of computing the derivative of a function
 - Same asymptotic complexity as the original function!
 - Many inefficient algorithms were published because the authors did not fully understand backpropagation

Backpropagation in a Meme

backprop is
just chain rule



backprop is just
the chain rule
+ memoization



backprop is an instance the
method of Lagrange multipliers,
which uses efficient block-
coordinate steps on the multipliers
& intermediate variables!



That's why it's correct, linear time
and linear space.

backprop is an instance the method of Lagrange
multipliers, [...] which gives me the freedom to optimize &
compute it with a million other methods!

One such method is reverse-mode (backprop), another is
forward-mode, but actually there is a huge spectrum of
methods that fall out of this elegant formulation; each
give interesting algs with different time-space tradeoffs
for gradient computation.

I can run optimization directly on the Lagrange dual -
moving away from the de facto block-coordinate scheme.

I can even support cyclic computations thanks to this
wonder unified view!



Fin