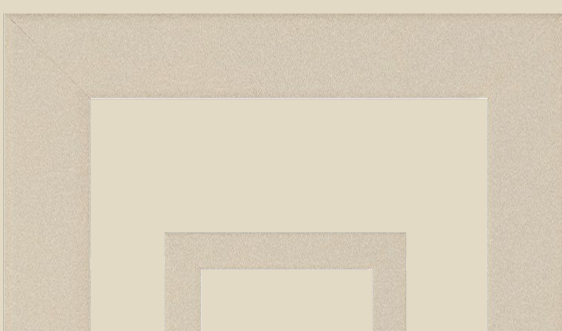
Papers We Love, Zürich

Papers We Love, Zürich

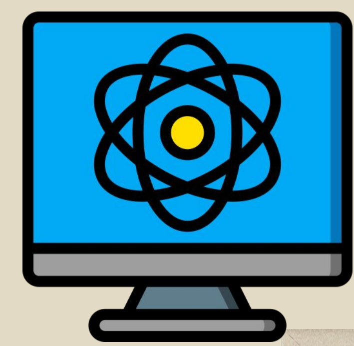


Papers We Love, Zürich

Community Driven

Papers We Love, Zürich

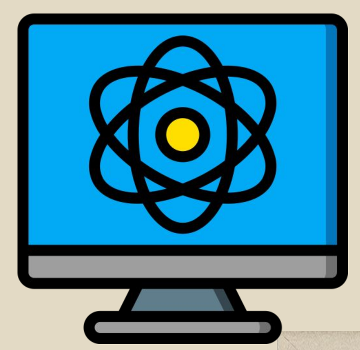
Community Driven



Computing Sciences

Papers We Love, Zürich

Community Driven



Computing Sciences

Informal

Papers We Love is a repository of academic computer science papers and a community who loves reading them.

Seattle San Francisco Columbus New York Chattanooga Vienna London Montreal St. Louis

Washington, DC San Diego Bangalore Toronto Singapore Brasilia Los Angeles Pune Boston

Rio de Janeiro Denver Berlin Buenos Aires Belfast Bucharest Winnipeg Athens Madrid Porto Chicago

Amsterdam Hyderabad Gothenburg Munich Barcelona Utrecht Philadelphia Zürich Milano

Raleigh-Durham Portland Teresina Lebanon Kyiv Hamburg Budapest Reykjavik Kansas City Beijing

Guadalajara Cairo Mumbai Kathmandu Seoul

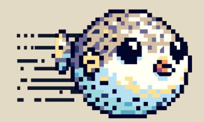
Abhiroop Postdoc

Dhruv PhD

Andrea

InfoSec Group

Organizers

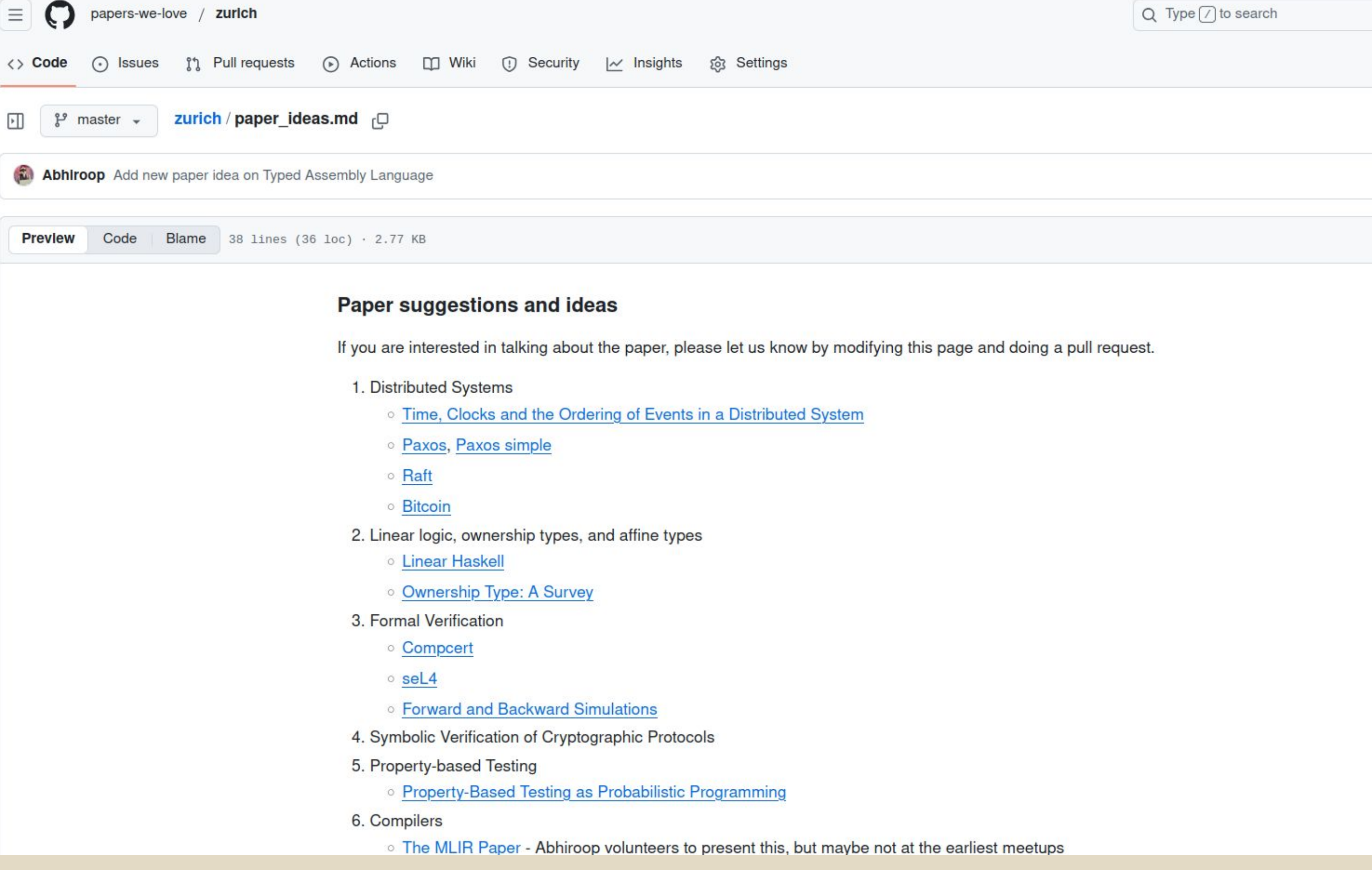


turbopuffer

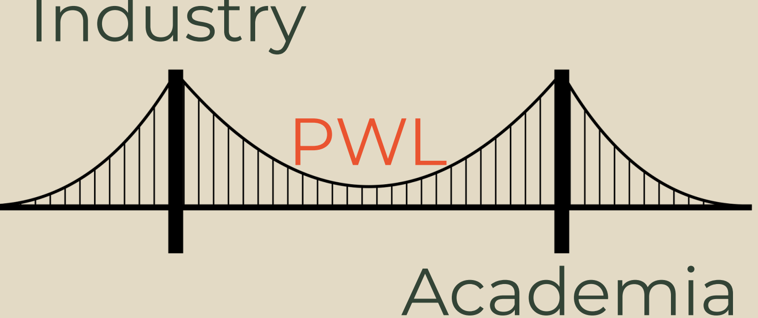
andrea@turbopuffer.com

Andrea

https://github.com/papers-we-love/zurich



https://stackoverflow.blog/2022/12/30/you-should-be-reading-academic-computer-science-papers/



DECEMBER 30, 2022

You should be reading academic computer science papers

You read documentation and tutorials to become a better programmer, but if you really want to be cutting-edge, academic research is where it's at.

Programming Language

Security

Distributed Systems

Operating Systems

Computer Networks

Computer Architecture

Formal Methods

Machine Learning

Not restricted to...

Computing Machinery and Intelligence - Turing, 1950 (PwL NYC)

Organisms ≠ Machines - Nicholson, 2013 (PwL LA)

Gödel's Incompleteness Theorems (PwL Boston)

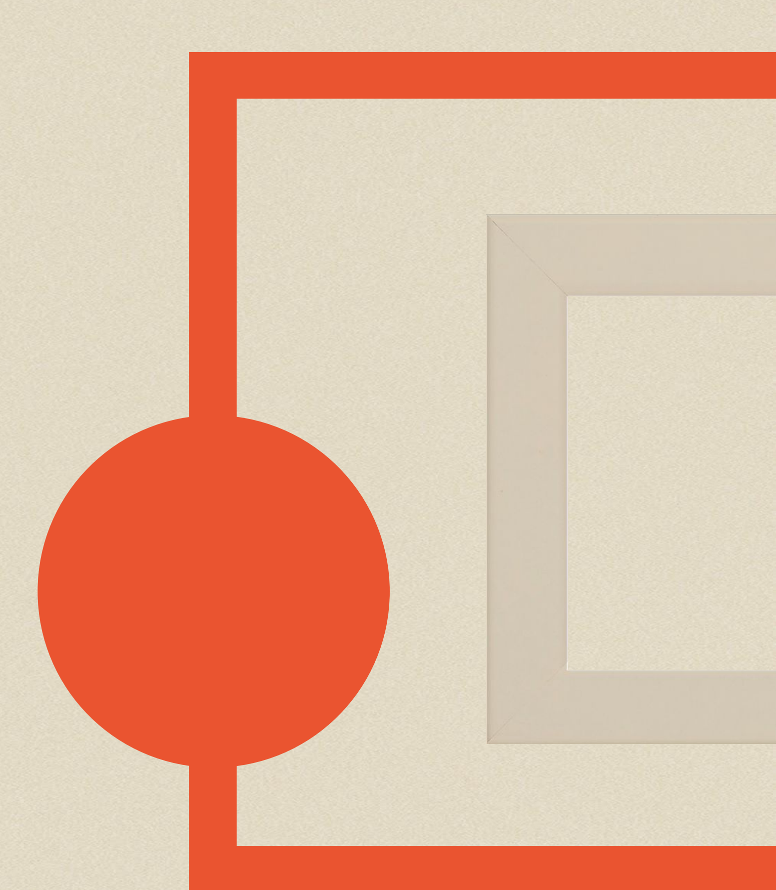
The Moral Character of Cryptographic Work - Rogaway, 2015 (PwL Winnipeg)

Onward to the paper…

Automatic Differentiation

A Programmer's Perspective

Abhiroop

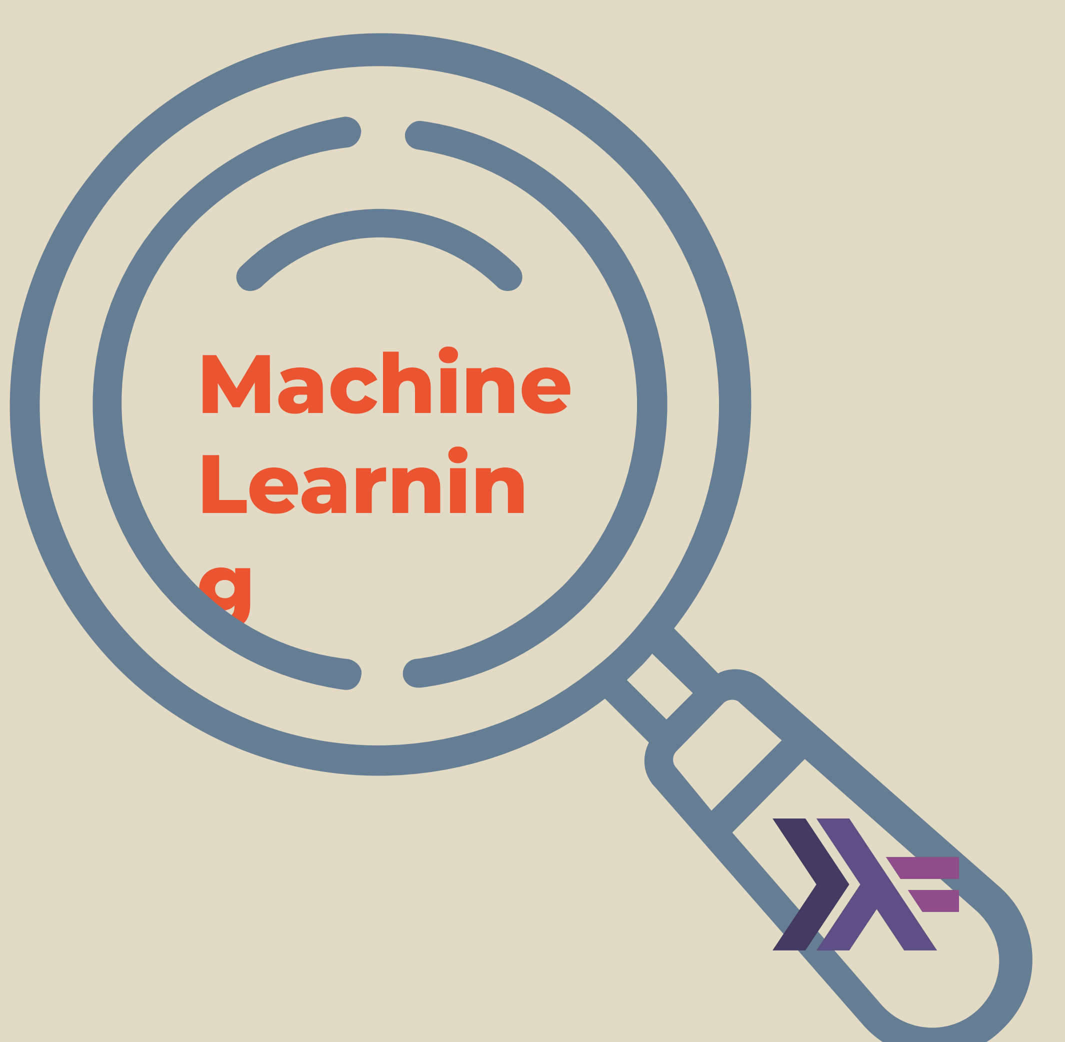


- Programming Languages
- Compilers
- Type Systems / Type Theory
- Functional Programming
- Security

My Research Areas

- Programming Languages
- Compilers
- Type Systems / Type Theory
- Functional Programming
- Security

NOT (Machine Learning)


Machine
Learnin
g

Differentiable Programming

Differentiable Programming

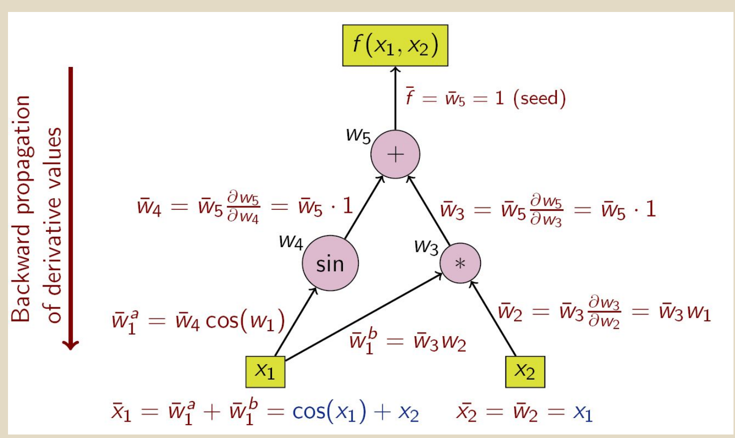
The Simple Essence of Automatic Differentiation

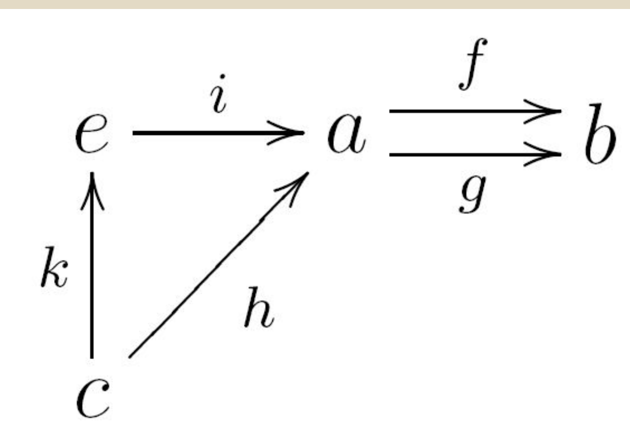
CONAL ELLIOTT, Target, USA

Automatic differentiation (AD) in reverse mode (RAD) is a central component of deep learning and other uses of large-scale optimization. Commonly used RAD algorithms such as backpropagation, however, are complex and stateful, hindering deep understanding, improvement, and parallel execution. This paper develops a simple, generalized AD algorithm calculated from a simple, natural specification. The general algorithm is then specialized by varying the representation of derivatives. In particular, applying well-known constructions to a naive representation yields two RAD algorithms that are far simpler than previously known. In contrast to commonly used RAD implementations, the algorithms defined here involve no graphs, tapes, variables, partial derivatives, or mutation. They are inherently parallel-friendly, correct by construction, and usable directly from an existing programming language with no need for new data types or programming style, thanks to use of an AD-agnostic compiler plugin.

The Simple Essence of Automatic Differentiation

CONAL ELLIOTT, Target, USA

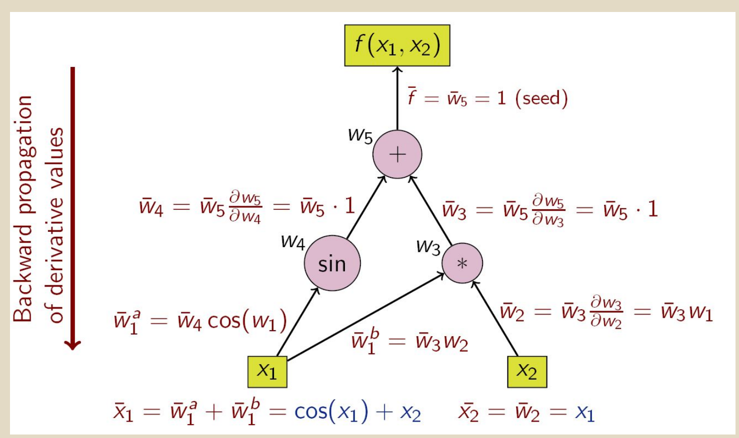


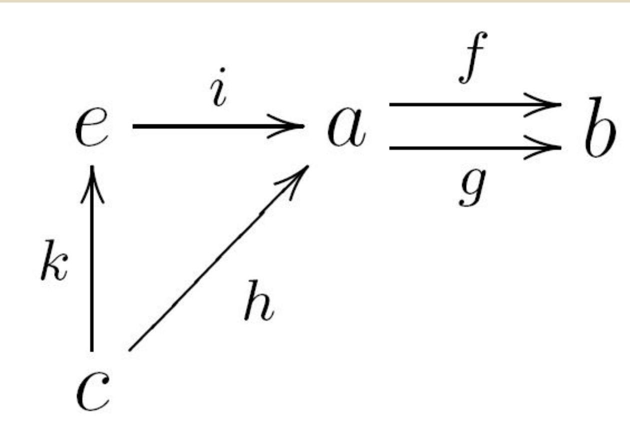


Category Theory

The Simple Essence of Automatic Differentiation

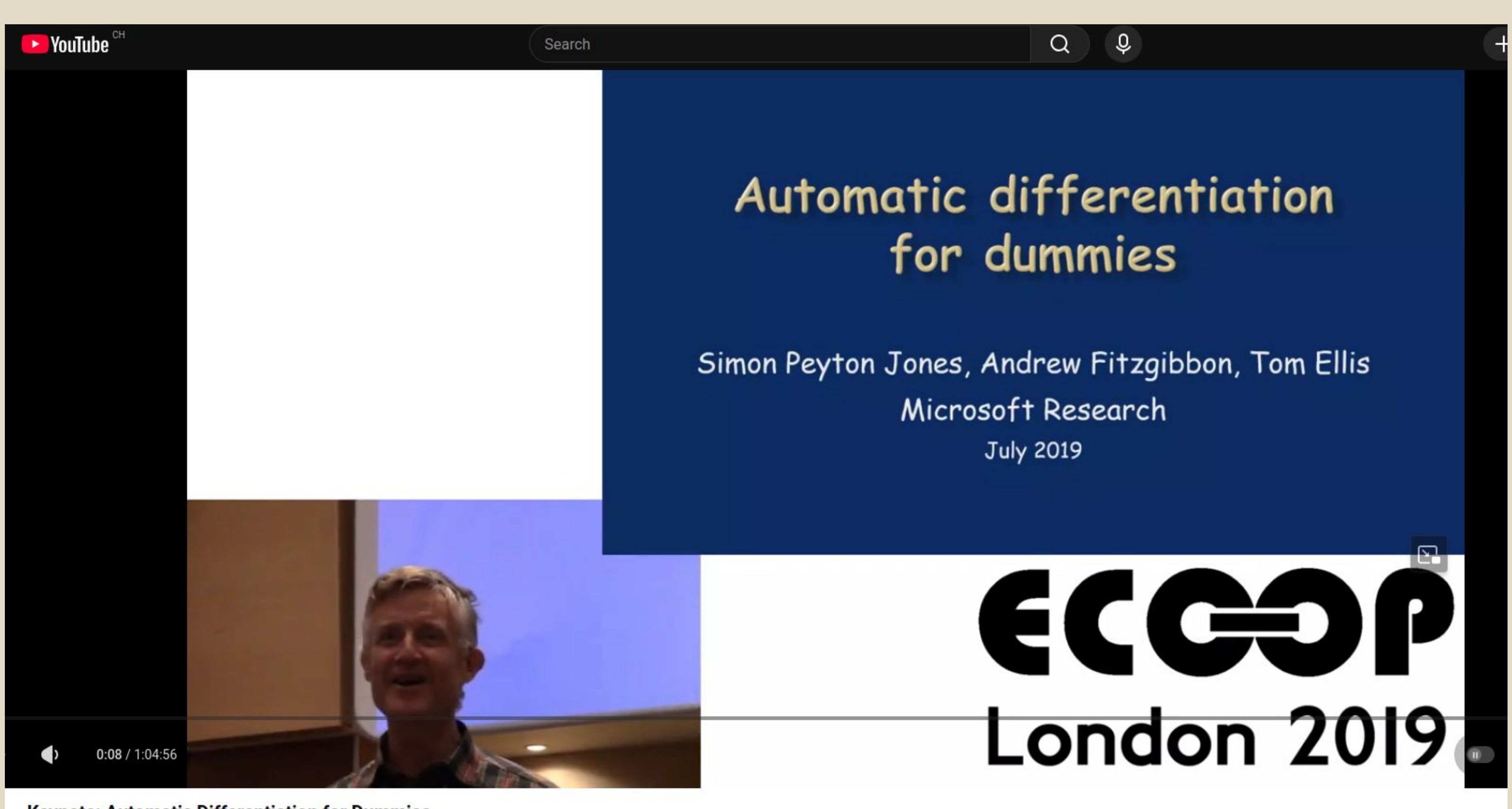
CONAL ELLIOTT, Target, USA

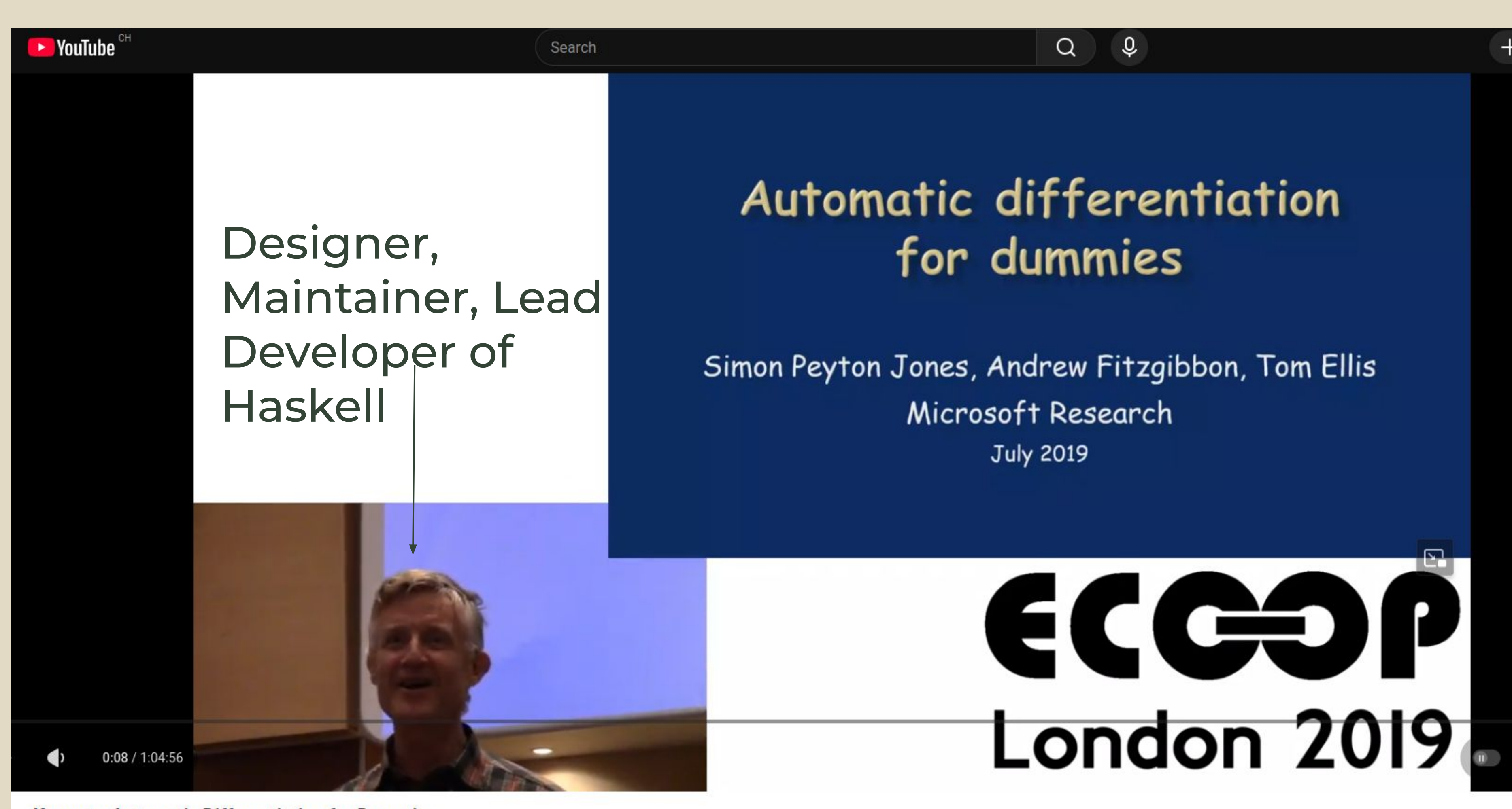


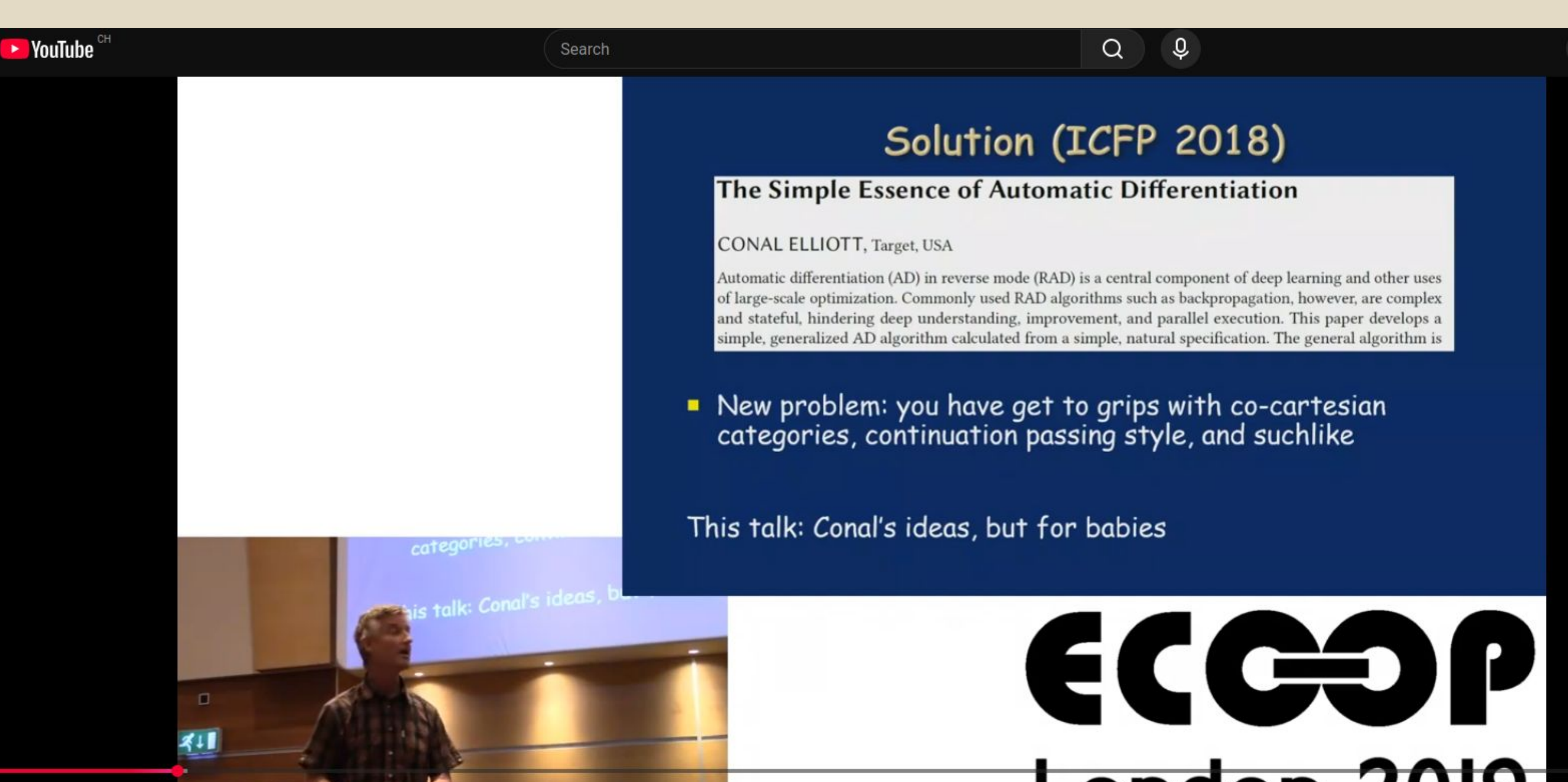


Category Theory

Is this really simple?
Or am I missing something?


YouTube CH
Search
Automatic differentiation
for dummies
Simon Peyton Jones, Andrew Fitzgibbon, Tom Ellis
Microsoft Research
July 2019
ECOOP
London 2019
0:08 / 1:04:56
Keynote: Automatic Differentiation for Dummies


YouTube CH
Search
Designer, Maintainer, Lead Developer of Haskell
Automatic differentiation for dummies
Simon Peyton Jones, Andrew Fitzgibbon, Tom Ellis
Microsoft Research
July 2019
ECOOP
London 2019
0:08 / 1:04:56
Keynote: Automatic Differentiation for Dummies



Keynote: Automatic Differentiation for Dummies

@sapito169 6 years ago

this is not for dummies

32

Reply

@sapito169 6 years ago

this is not for dummies

32 Reply

@rrr00bb1 3 years ago

this is by far the most vexing explanation ever.

2 Reply

@sapito169 6 years ago

this is not for dummies

32 Reply

@rrr00bb1 3 years ago

this is by far the most vexing explanation ever.

2 Reply

@haps3000 2 years ago

Also, congrats to the audience for supressing their inner rage towards the guy that created Haskell.

Reply

Automatic Differentiation in Machine Learning: a Survey

Atılım Güneş Baydin GUNES@ROBOTS.OX.AC.UK
Department of Engineering Science
University of Oxford
Oxford OX1 3PJ, United Kingdom

Barak A. Pearlmutter BARAK@PEARLMUTTER.NET
Department of Computer Science
National University of Ireland Maynooth
Maynooth, Co. Kildare, Ireland

Alexey Andreyevich Radul AXCH@MIT.EDU
Department of Brain and Cognitive Sciences
Massachusetts Institute of Technology
Cambridge, MA 02139, United States

Jeffrey Mark Siskind QOBI@PURDUE.EDU
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907, United States

Abstract

Derivatives, mostly in the form of gradients and Hessians, are ubiquitous in machine learning. Automatic differentiation (AD), also called algorithmic differentiation or simply "autodiff", is a family of techniques similar to but more general than backpropagation for effi-

Chain Rules

1. **Sum / difference**

$$\frac{d}{dx}[f(x)+g(x)] = f'(x)+g'(x)$$

Linear: derivative distributes over sums.

2. **Constant multiple**

$$\frac{d}{dx}[c\,f(x)] = c\,f'(x)$$

Constants factor out.

3. **Product (Leibniz) — two factors**

$$\frac{d}{dx}[f(x)g(x)] = f'(x)g(x)+f(x)g'(x)$$

Derivative of a product = sum of each factor differentiated while the other is held.

4. **Quotient**

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{f'(x)g(x)-f(x)g'(x)}{g(x)^2}$$

Can be derived from product + chain rules (or use $f/g = f \cdot g^{-1}$).

5. **Power rule (real exponent n)**

$$\frac{d}{dx}[x^n] = nx^{n-1} \quad \text{(for constant } n\text{)}$$

More generally for $f(x)^n$: $\frac{d}{dx}[f(x)^n] = nf(x)^{n-1}f'(x)$.

f(x), g(x) is differentiable.

A function $f : \mathbb{R} \to \mathbb{R}$ is **differentiable at a point** a if the following limit exists:

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

If this limit exists (as a real number), then:

- the derivative at a is $f'(a)$,
- and f is said to be **differentiable at a**.

1. **Sum / difference**

$$\frac{d}{dx}[f(x) + g(x)] = f'(x) + g'(x)$$

Linear: derivative distributes over sums.

2. **Constant multiple**

$$\frac{d}{dx}[c\, f(x)] = c\, f'(x)$$

Constants factor out.

3. **Product (Leibniz) — two factors**

$$\frac{d}{dx}[f(x)g(x)] = f'(x)g(x) + f(x)g'(x)$$

Derivative of a product = sum of each factor differentiated while the other is held.

4. **Quotient**

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}$$

Can be derived from product + chain rules (or use $f/g = f \cdot g^{-1}$).

5. **Power rule (real exponent n)**

$$\frac{d}{dx}[x^n] = nx^{n-1} \quad \text{(for constant } n\text{)}$$

More generally for $f(x)^n$: $\frac{d}{dx}[f(x)^n] = nf(x)^{n-1}f'(x)$.

$$h(x) = f(x)g(x)$$
$$h'(x) = f'(x)g(x) + f(x)g'(x)$$

$$h(x) = f(x)g(x)$$

$$h'(x) = f'(x)g(x) + f(x)g'(x)$$

$$f(x) = u(x)v(x)$$

$$h'(x) = (u'(x)v(x) + u(x)v'(x))g(x) + f(x)g'(x)$$

Automatic Differentiation

$$f(x1, x2) = (\sin(x1/x2) + (x1/x2) - e^{(x2)}) * ((x1/x2) - e^{(x2)})$$

```
def f(x1, x2):
    a = x1 / x2
    b = math.exp(x2)
    return (math.sin(a) + a - b) * (a - b)
```

Work with values rather than expressions

Automatic Differentiation

```
def f(x1, x2):
    a = x1 / x2
    b = math.exp(x2)
    return (math.sin(a) + a - b) * (a - b)
```

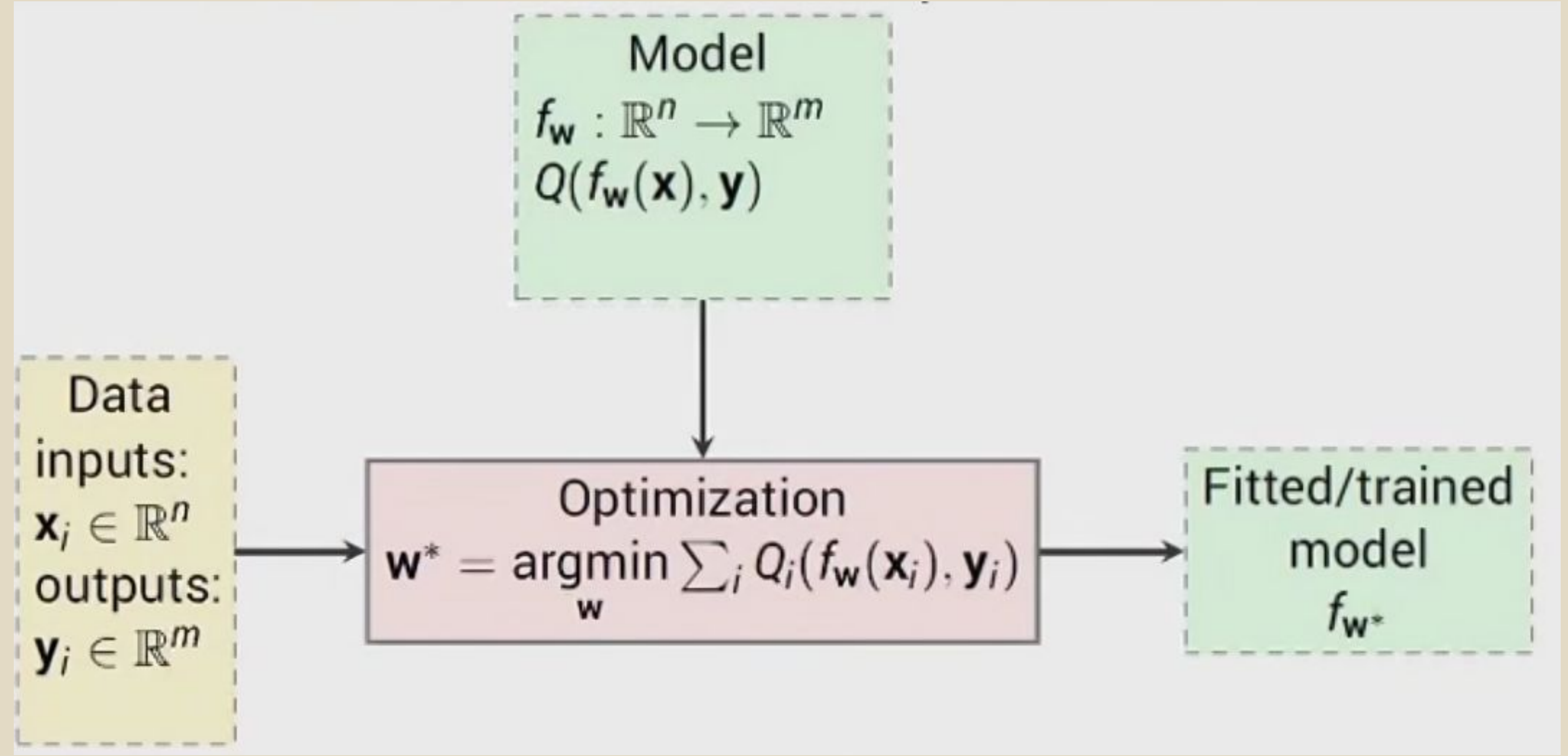
Intermediate Values

Automatic Differentiation

```
def f(x1, x2):
    a = x1 / x2
    b = math.exp(x2)
    return (math.sin(a) + a - b) * (a - b)
```

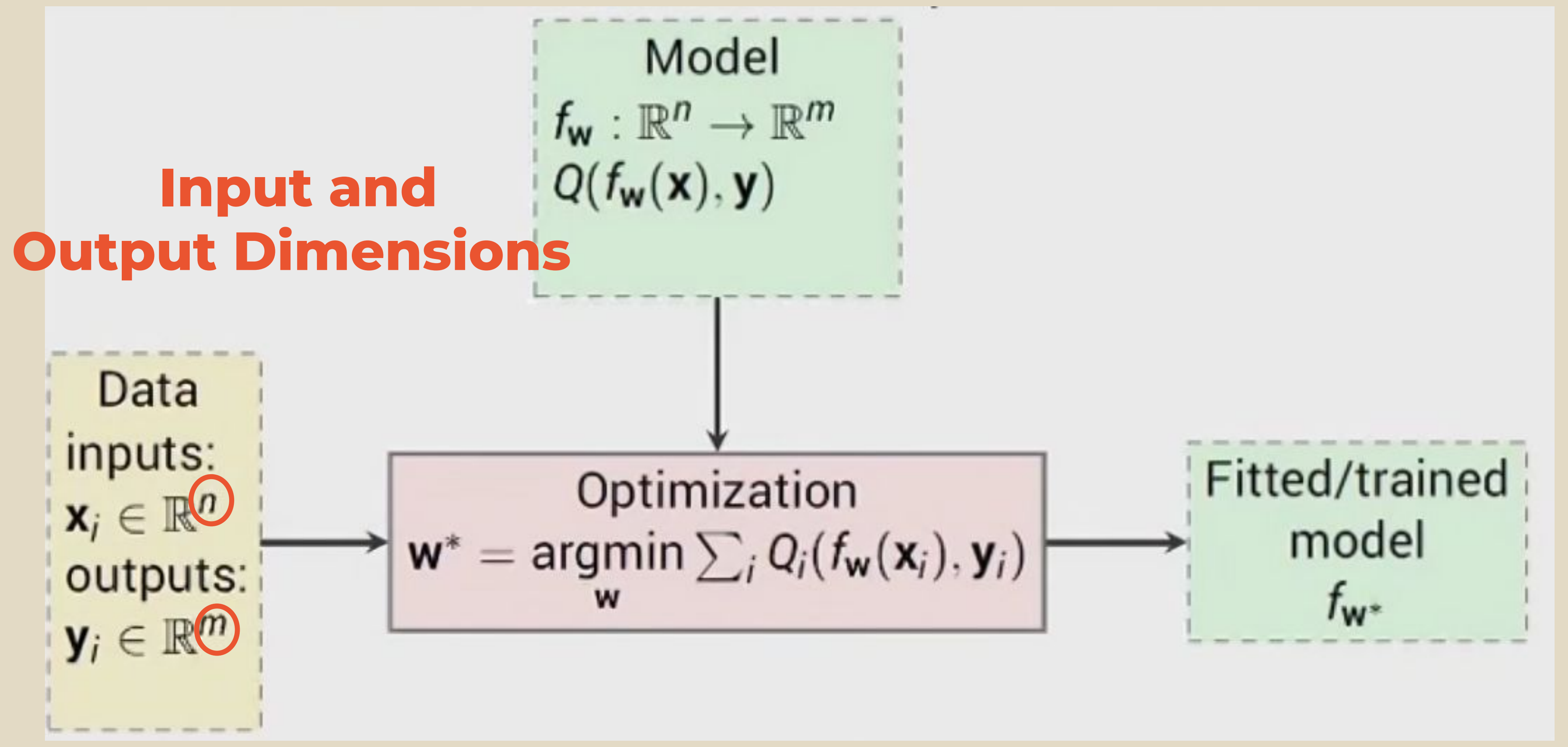
Elementary Operations - Derivatives are

Derivatives and Optimization



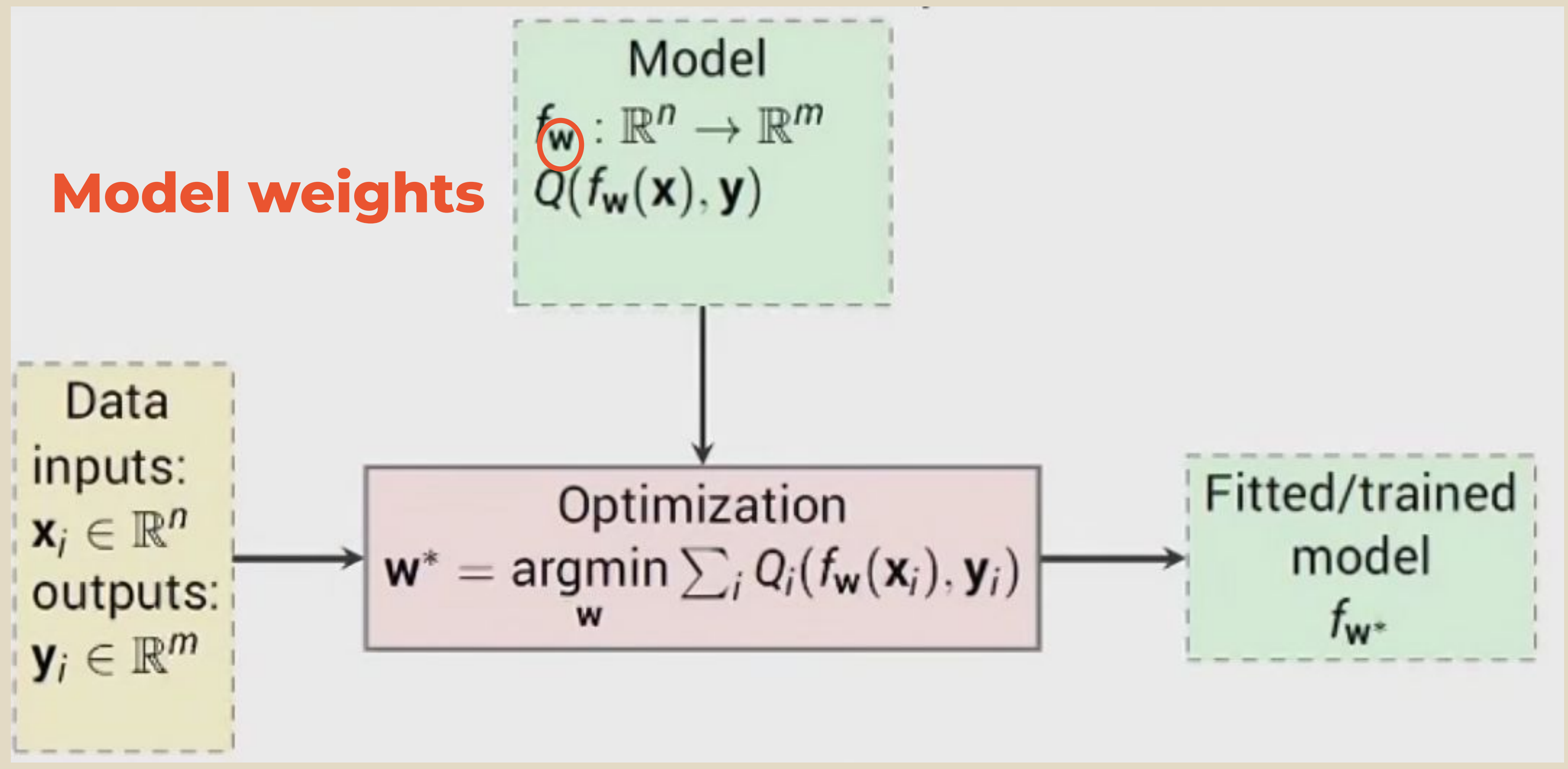
Baydin et al. 2018

Derivatives and Optimization



Baydin et al. 2018

Derivatives and Optimization



$f_w(x) = Wx + b, w = (W, b)$ where $W \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.

Baydin et al. 2018

$$w^* = \arg\min_w \sum_i Q(f_w(x_i),\ y_i)$$

Total Loss

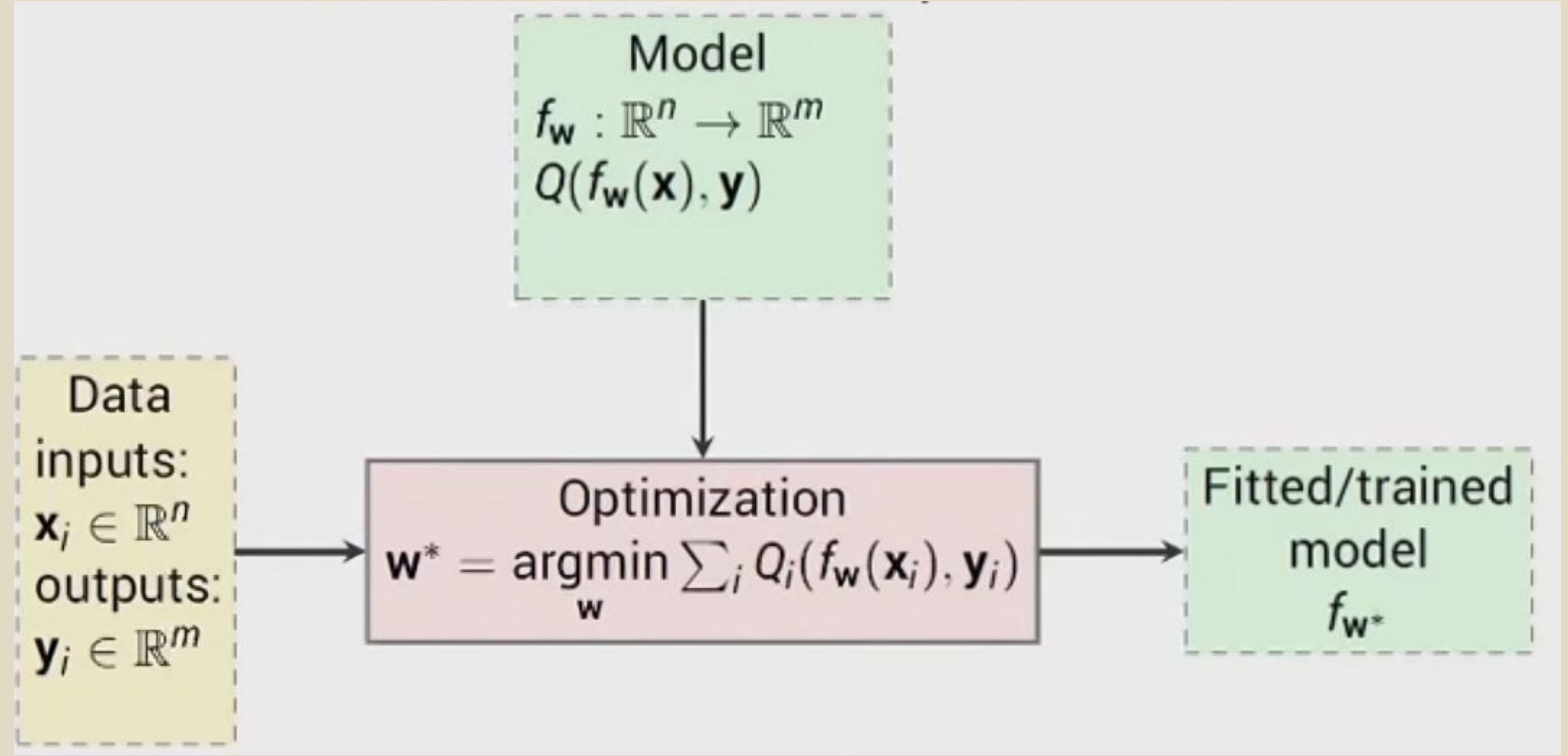
$$L(w) = \sum_{i=1}^{N} Q(f_w(x_i), y_i)$$

arg min_w = Find the parameter w that minimise L(w)

$$\nabla_w L(w) = \sum_{i=1}^{N} \frac{\partial}{\partial w} Q(f_w(x_i), y_i).$$

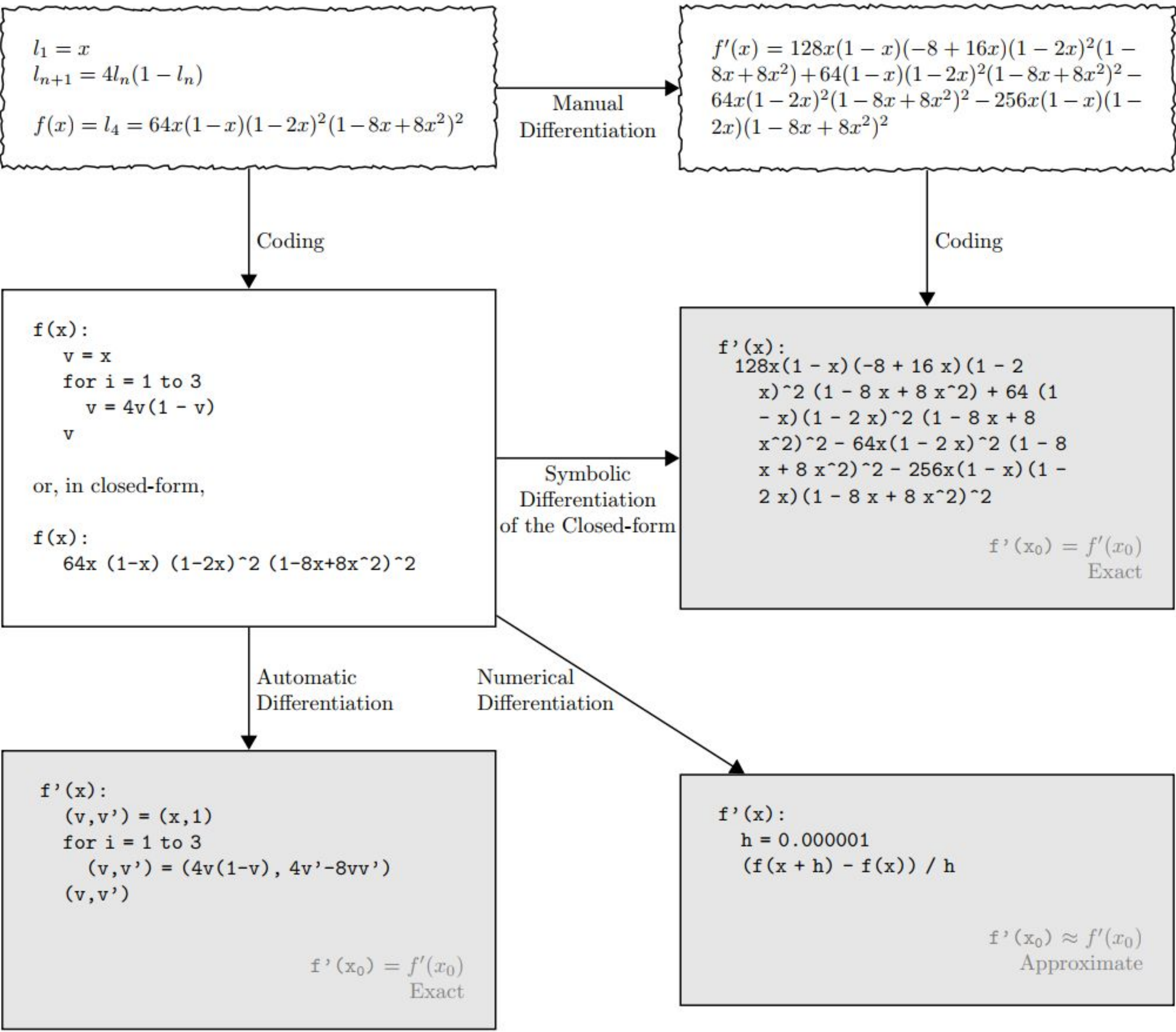
$$\frac{\partial}{\partial w} Q(f_w(x_i), y_i) = \frac{\partial Q}{\partial \hat{y}}\bigg|_{\hat{y}=f_w(x_i)} \cdot \frac{\partial f_w(x_i)}{\partial w}.$$

Derivatives and Optimization

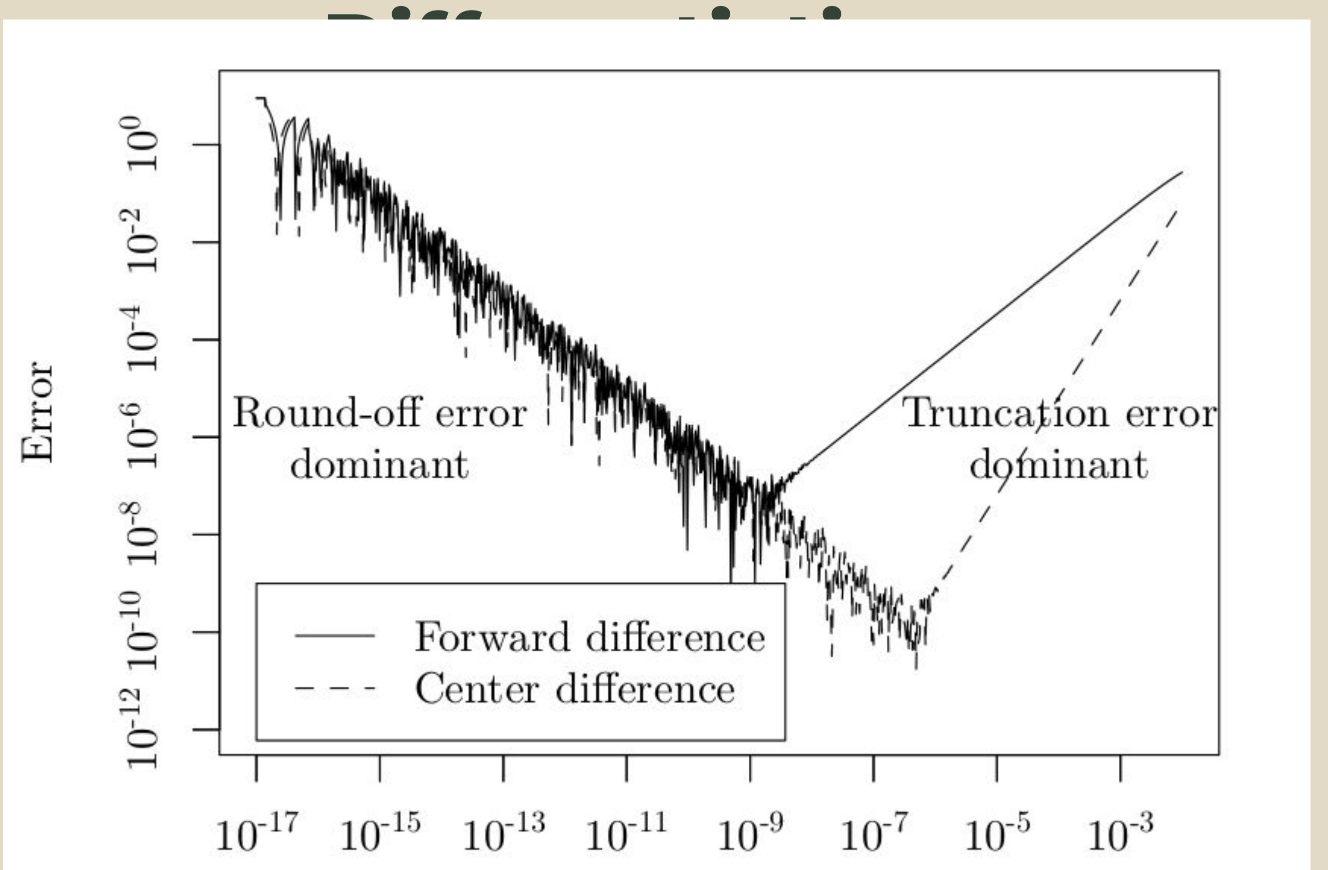


Baydin et al. 2018

Derivatives as code



Numeric Differentiation



Symbolic Differentiation

Table 1: Iterations of the logistic map $l_{n+1} = 4l_n(1 - l_n)$, $l_1 = x$ and the corresponding derivatives of l_n with respect to x, illustrating expression swell.

n	l_n	$\frac{d}{dx}l_n$	$\frac{d}{dx}l_n$ (Simplified form)
1	x	1	1
2	$4x(1-x)$	$4(1-x) - 4x$	$4 - 8x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2 - 16x(1-2x)^2 - 64x(1-x)(1-2x)$	$16(1 - 10x + 24x^2 - 16x^3)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2) + 64(1-x)(1-2x)^2(1-8x+8x^2)^2 - 64x(1-2x)^2(1-8x+8x^2)^2 - 256x(1-x)(1-2x)(1-8x+8x^2)^2$	$64(1 - 42x + 504x^2 - 2640x^3 + 7040x^4 - 9984x^5 + 7168x^6 - 2048x^7)$

Symbolic

- Expression Swell
- Closed Form Expressions

Table 1: Iterations of the logistic map $l_{n+1} = 4l_n(1 - l_n)$, $l_1 = x$ and the corresponding derivatives of l_n with respect to x, illustrating expression swell.

n	l_n	$\frac{d}{dx}l_n$	$\frac{d}{dx}l_n$ (Simplified form)
1	x	1	1
2	$4x(1-x)$	$4(1-x)-4x$	$4-8x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2 - 16x(1-2x)^2 - 64x(1-x)(1-2x)$	$16(1-10x+24x^2-16x^3)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$	$64(1-42x+504x^2-2640x^3+7040x^4-9984x^5+7168x^6-2048x^7)$

Automatic Differentiation

Automatic Differentiation

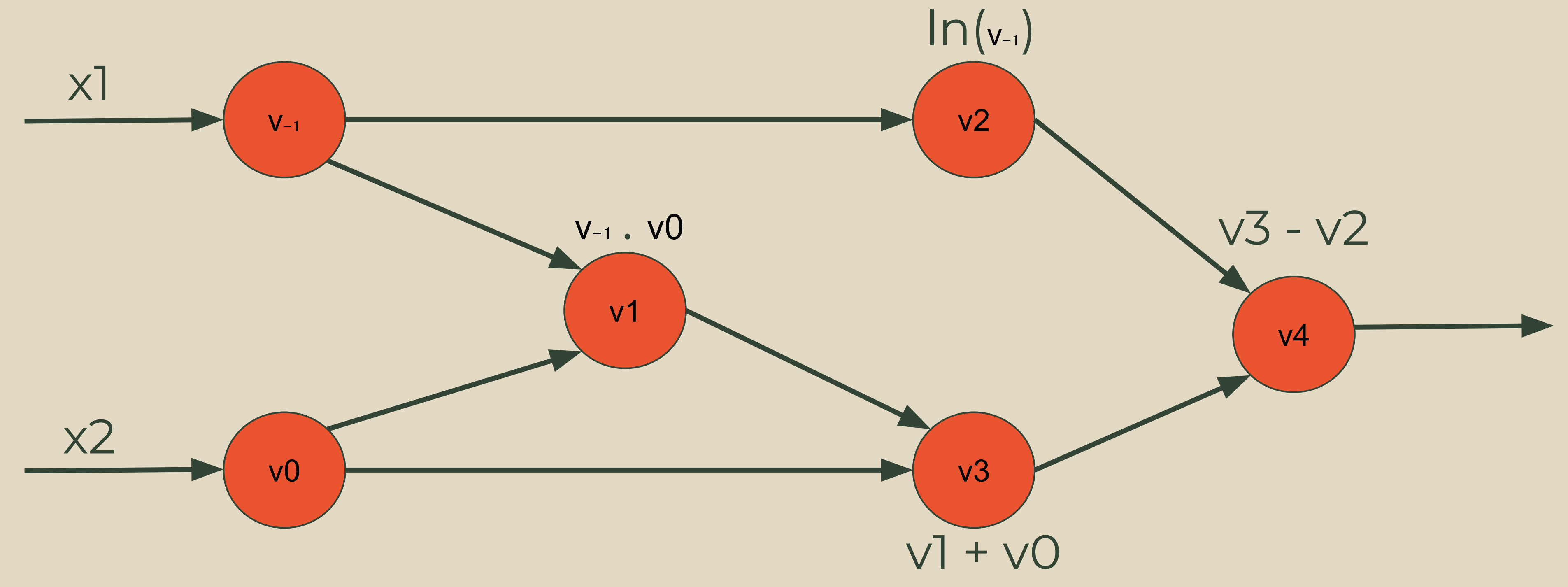
- Forward Mode
- Reverse Mode

Forward Mode AD

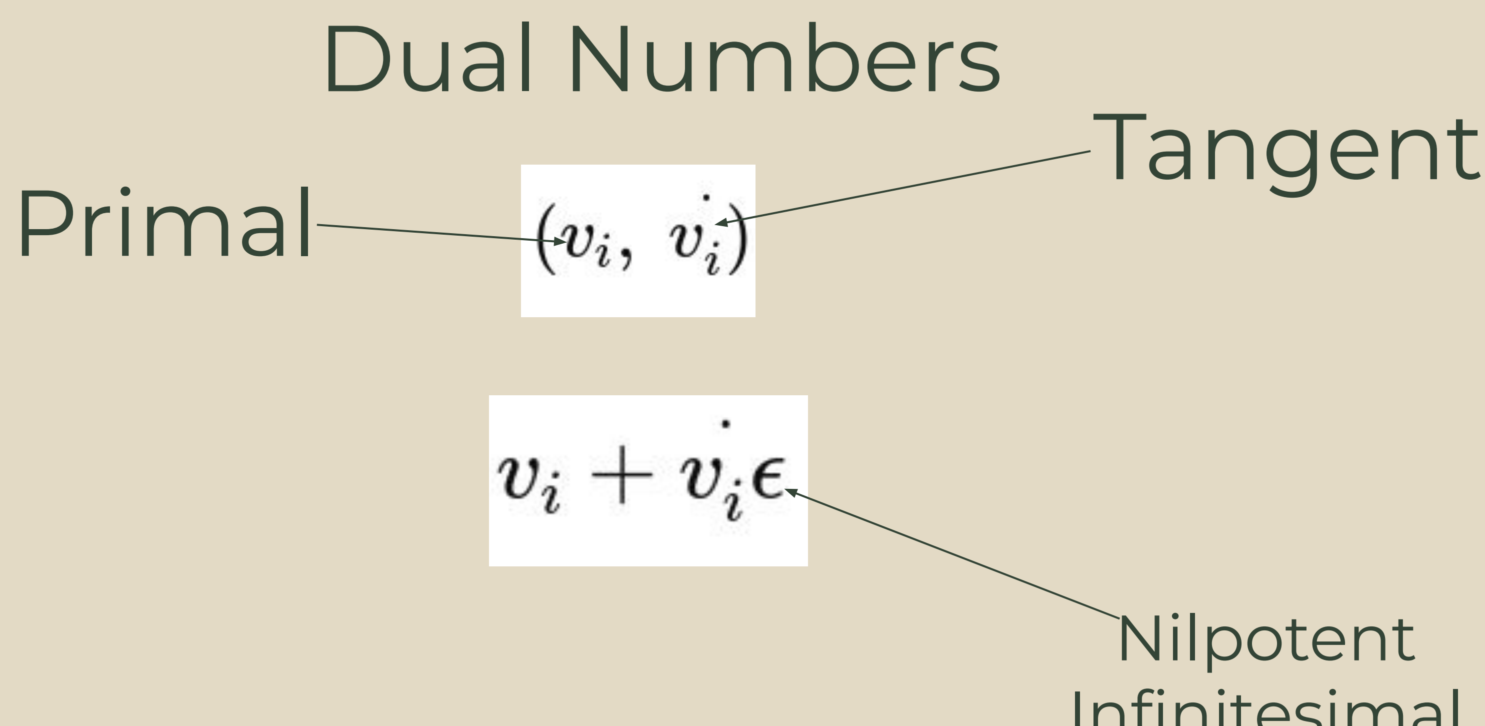
$$y = f(x1, x2) = x1.x2 + x2 - \ln(x1)$$

Forward Mode AD

$$y = f(x1, x2) = x1.x2 + x2 - \ln(x1)$$

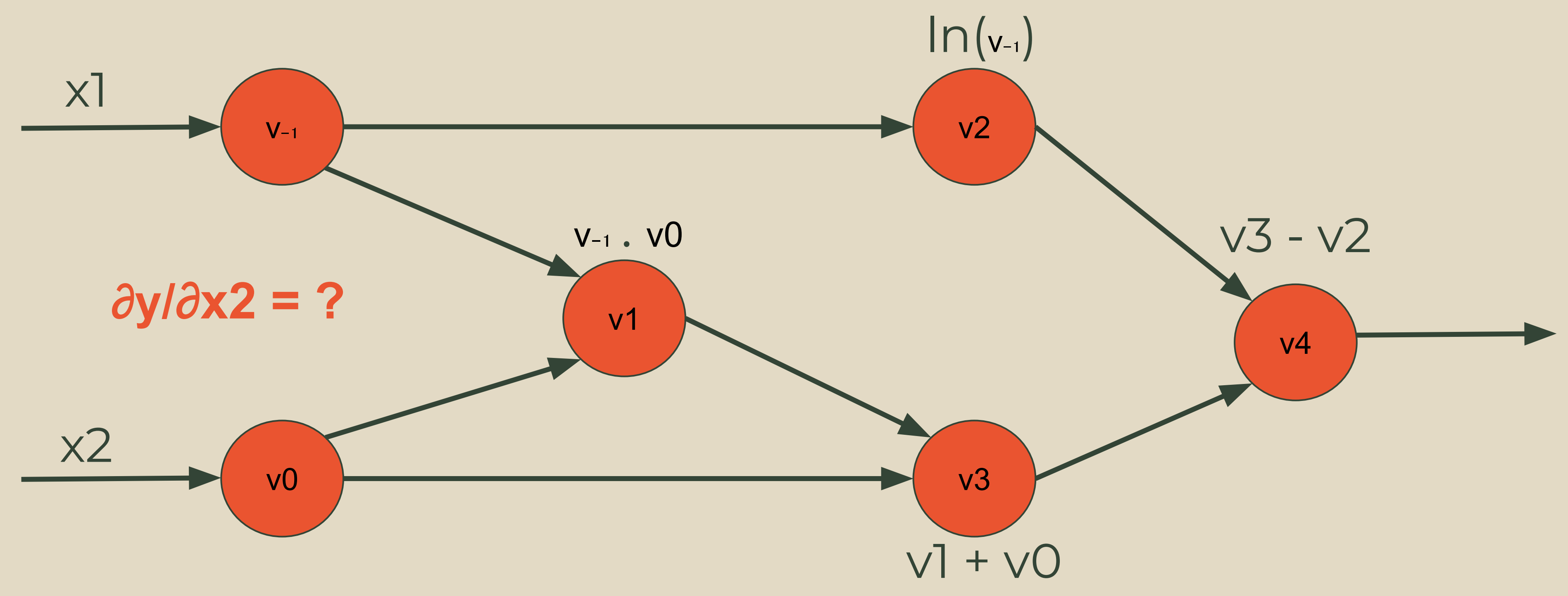


Forward Mode AD



Forward Mode AD

$$y = f(x1, x2) = x1.x2 + x2 - \ln(x1)$$



Forward Mode AD

Forward Primal Trace	Output	Forward Tangent Trace	Output
$v_{-1} = x_1$	3	$\dot{v}_{-1} = \dot{x}_{-1}$	0
$v_0 = x_2$	-4	$\dot{v}_0 = \dot{x}_2$	1
$v_1 = v_{-1}v_0$	$3 \cdot -4 = -12$	$\dot{v}_1 = \dot{v}_{-1}v_0 + \dot{v}_0 v_{-1}$	$0 \cdot -4 + 1 \cdot 3 = 3$
$v_2 = \ln(v_{-1})$	$\ln(3) = 1.10$	$\dot{v}_2 = \dot{v}_{-1} \cdot (1 / v_{-1})$	$0 \cdot (1 / 3) = 0$
$v_3 = v_1 + v_0$	$-12 + -4 = -16$	$\dot{v}_3 = \dot{v}_1 + \dot{v}_0$	$3 + 1 = 4$
$v_4 = v_3 - v_2$	$-16 - 1.10 = -17.10$	$\dot{v}_4 = \dot{v}_3 - \dot{v}_2$	$4 - 0 = 4$
$y = v_4$	-17.10	$\dot{y} = \dot{v}_4$	4

Forward Mode AD

Jacobian Matrix

$$\mathbf{J} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Forward Mode AD

Jacobian-vector Product

$$\mathbf{J} \cdot \mathbf{r} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \cdot \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix}$$

Forward Mode AD Implementation

- Source to source transformation
- Special purpose compilers
- Operator Overloading

Reverse Mode AD

Reverse Mode AD

Adjoint

$$\bar{v} = \frac{\partial y}{\partial v}$$

How sensitive is y to small changes in v?

Reverse Mode AD

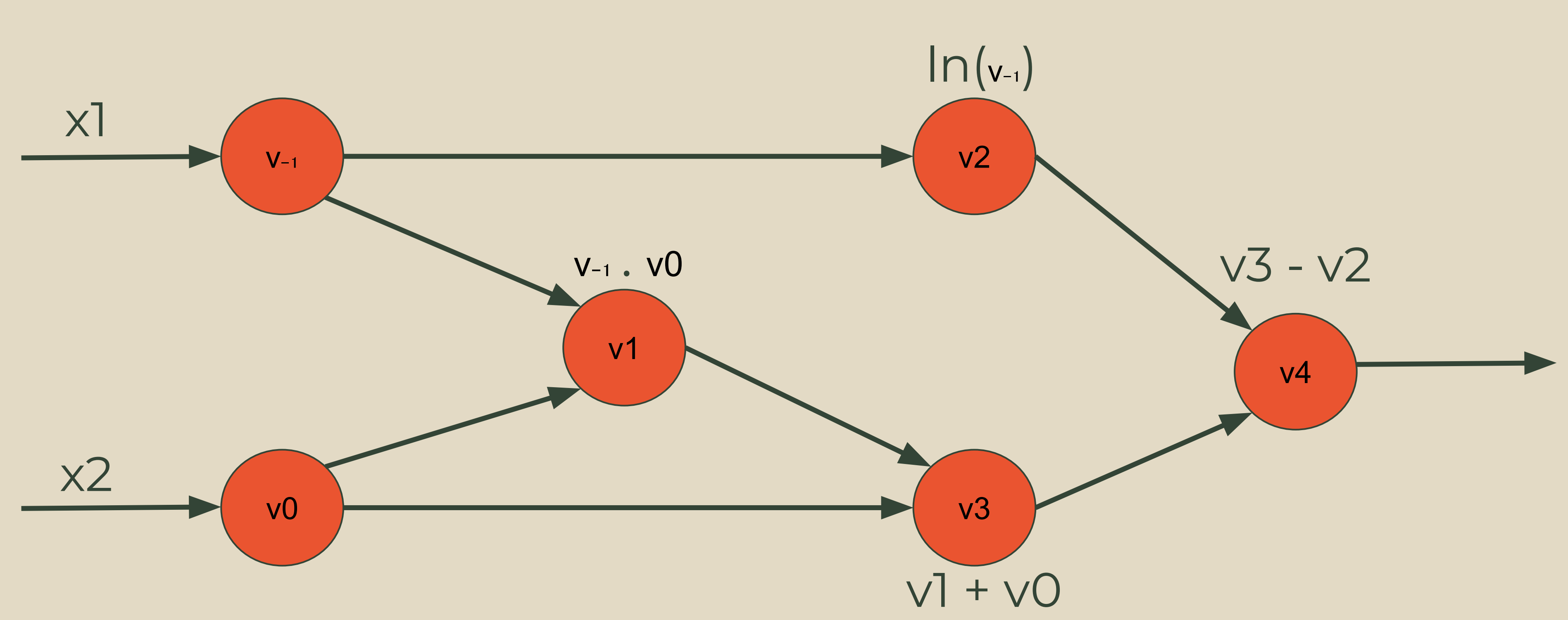
Adjoint

$$\bar{v} = \frac{\partial y}{\partial v}$$

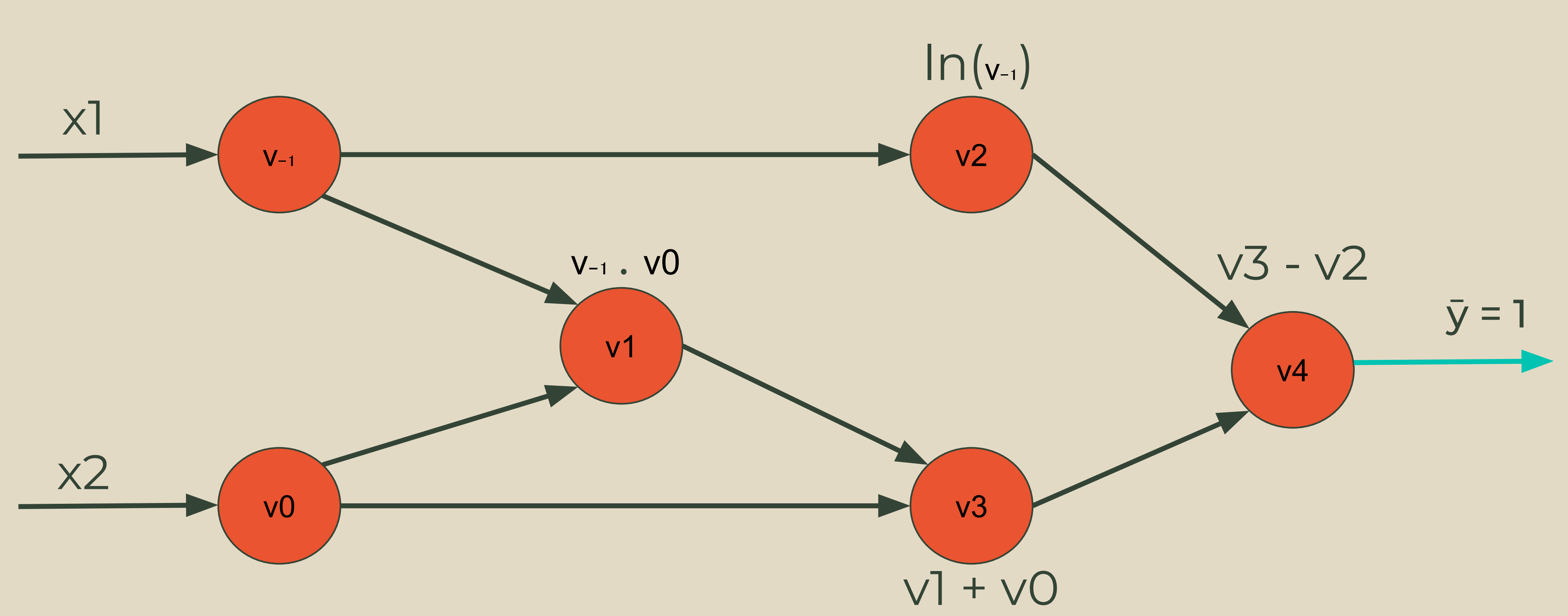
For the ith node

$$\bar{v}_i = \sum_{j \in \text{child}(i)} \bar{v}_j \cdot \frac{\partial v_j}{\partial v_i}$$

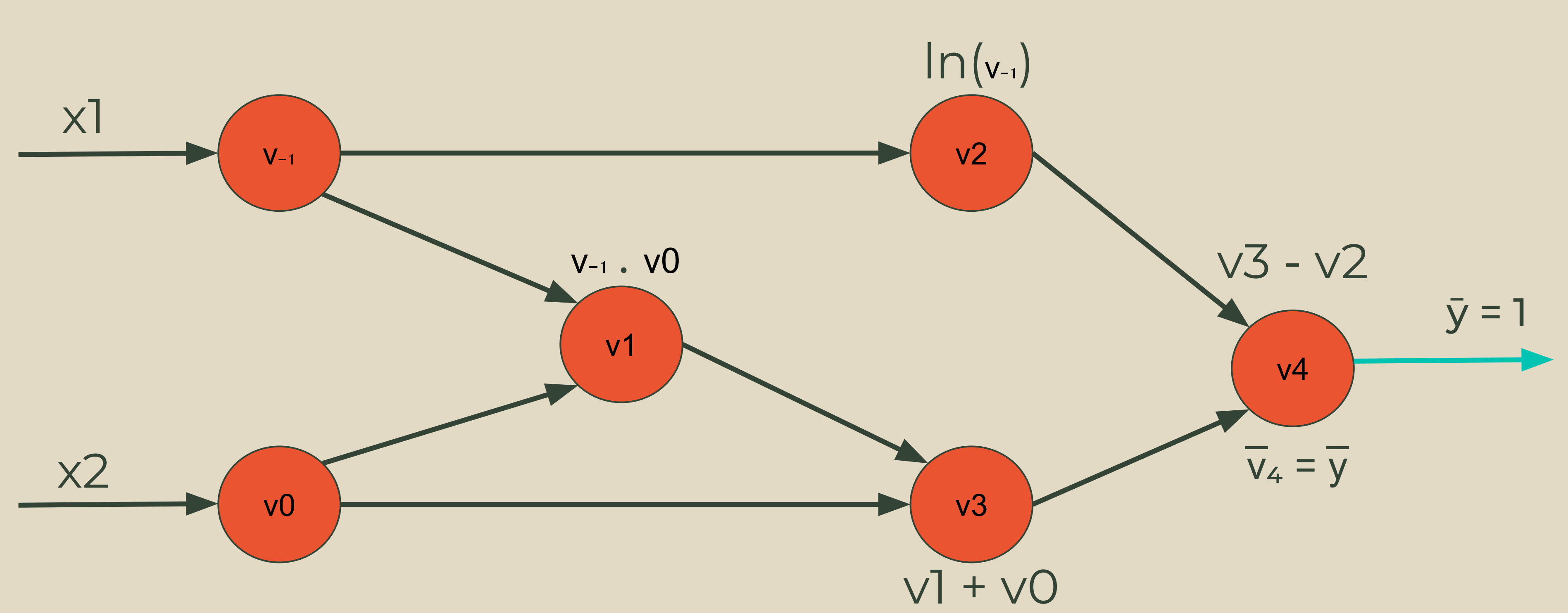
Reverse Mode AD



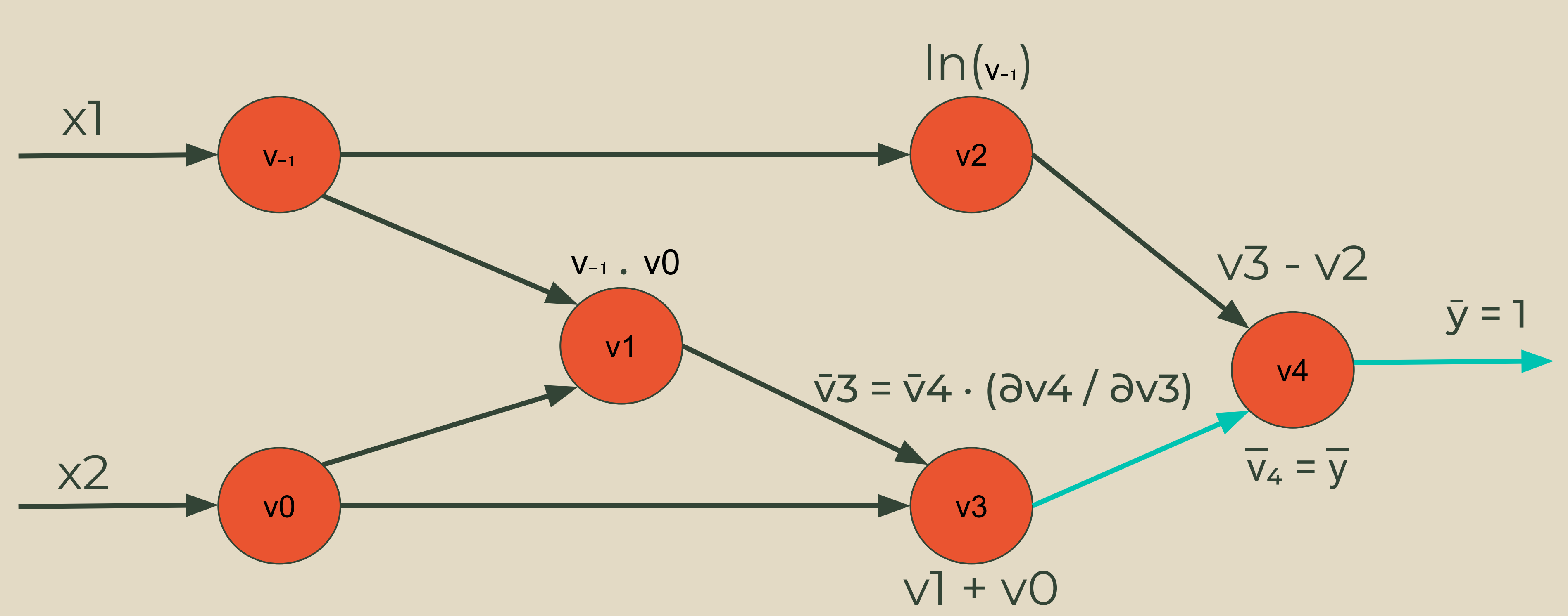
Reverse Mode AD



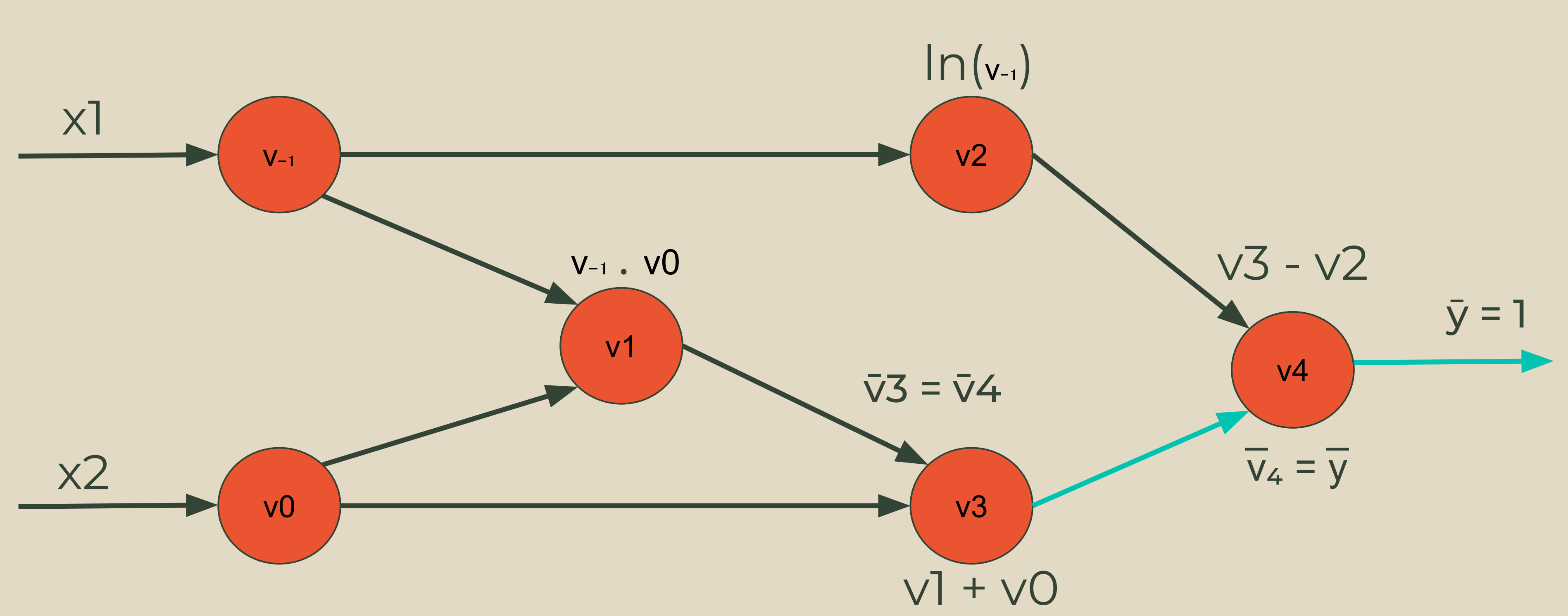
Reverse Mode AD



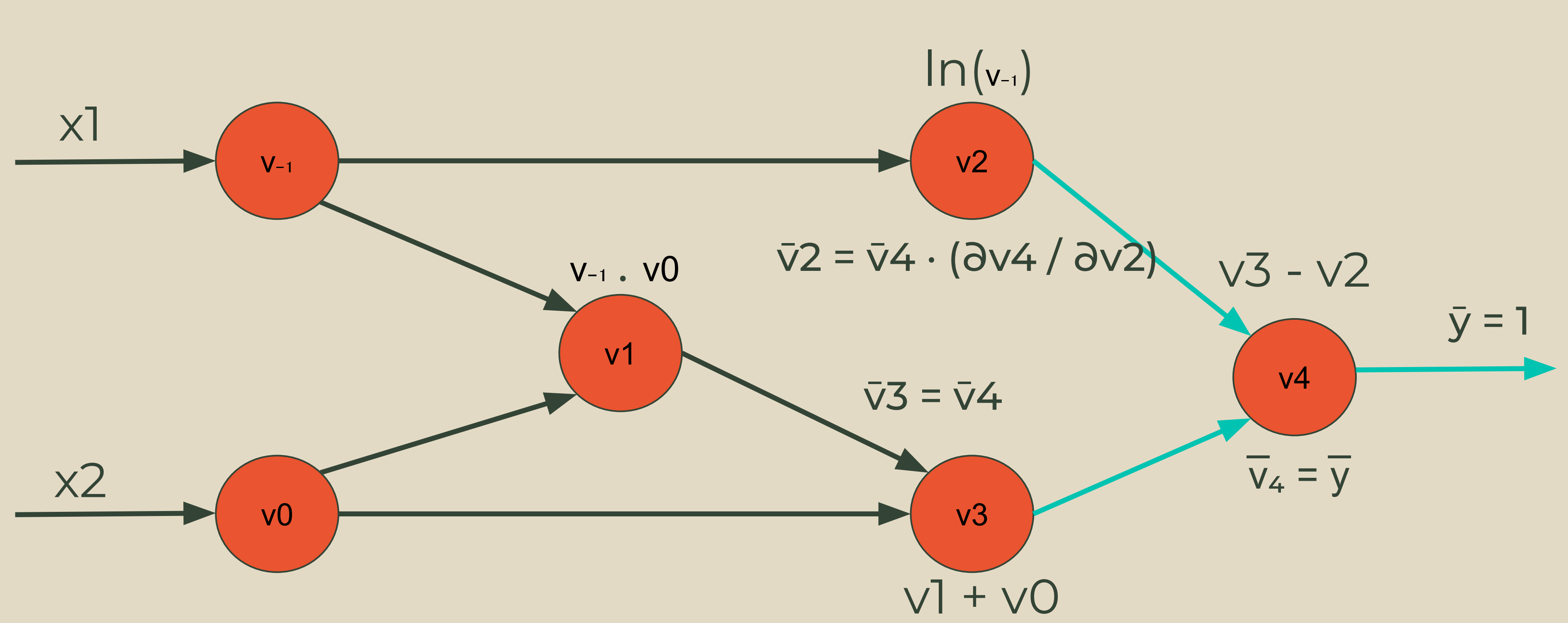
Reverse Mode AD



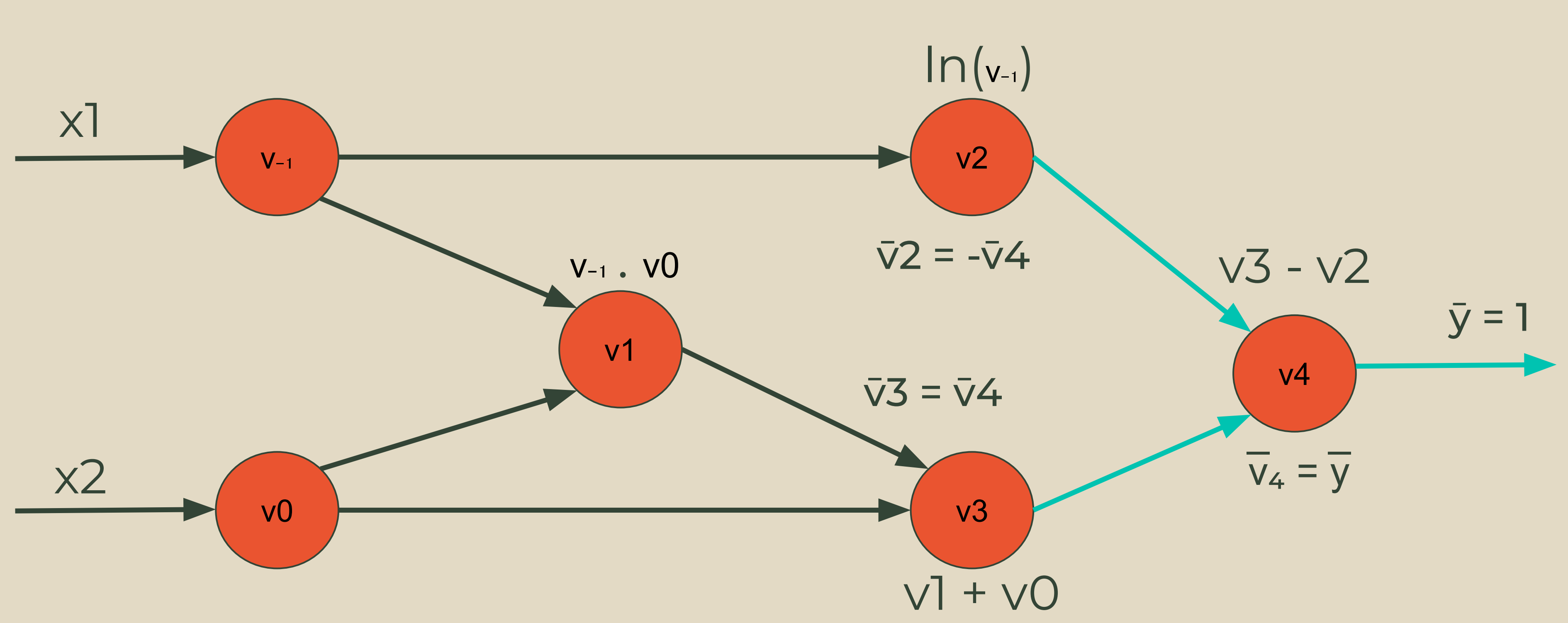
Reverse Mode AD



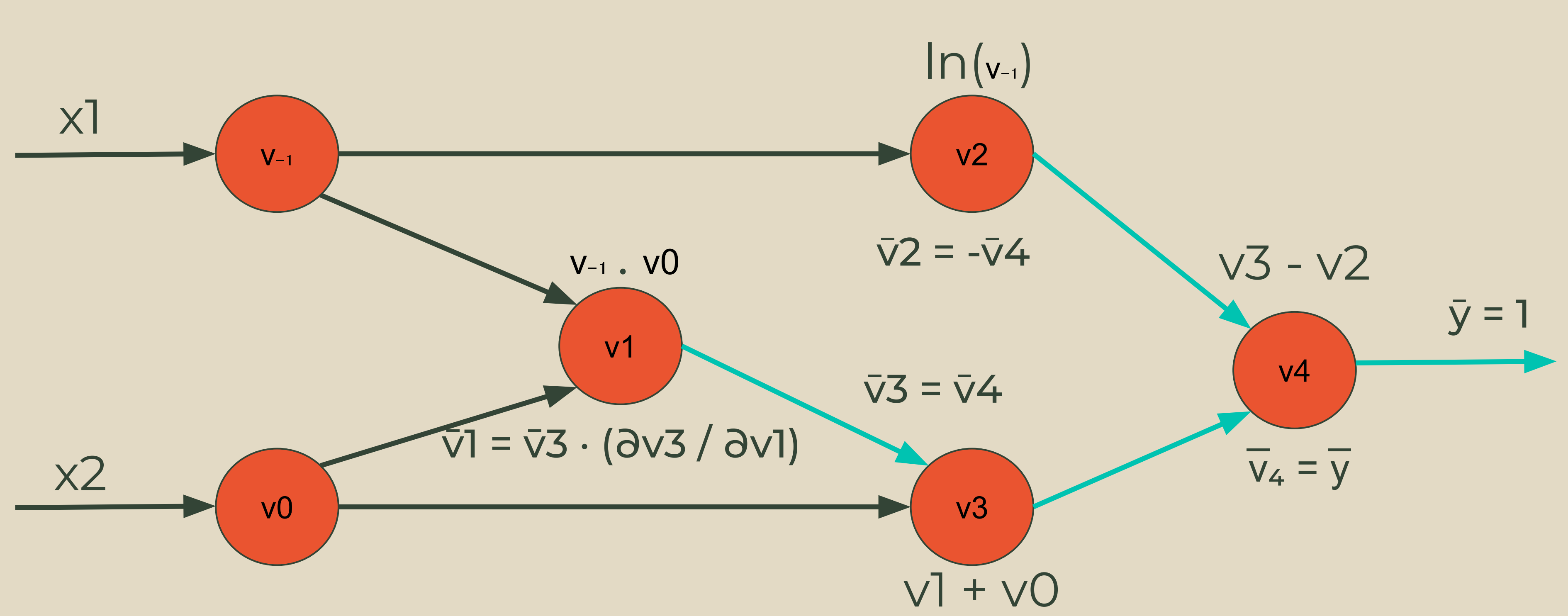
Reverse Mode AD



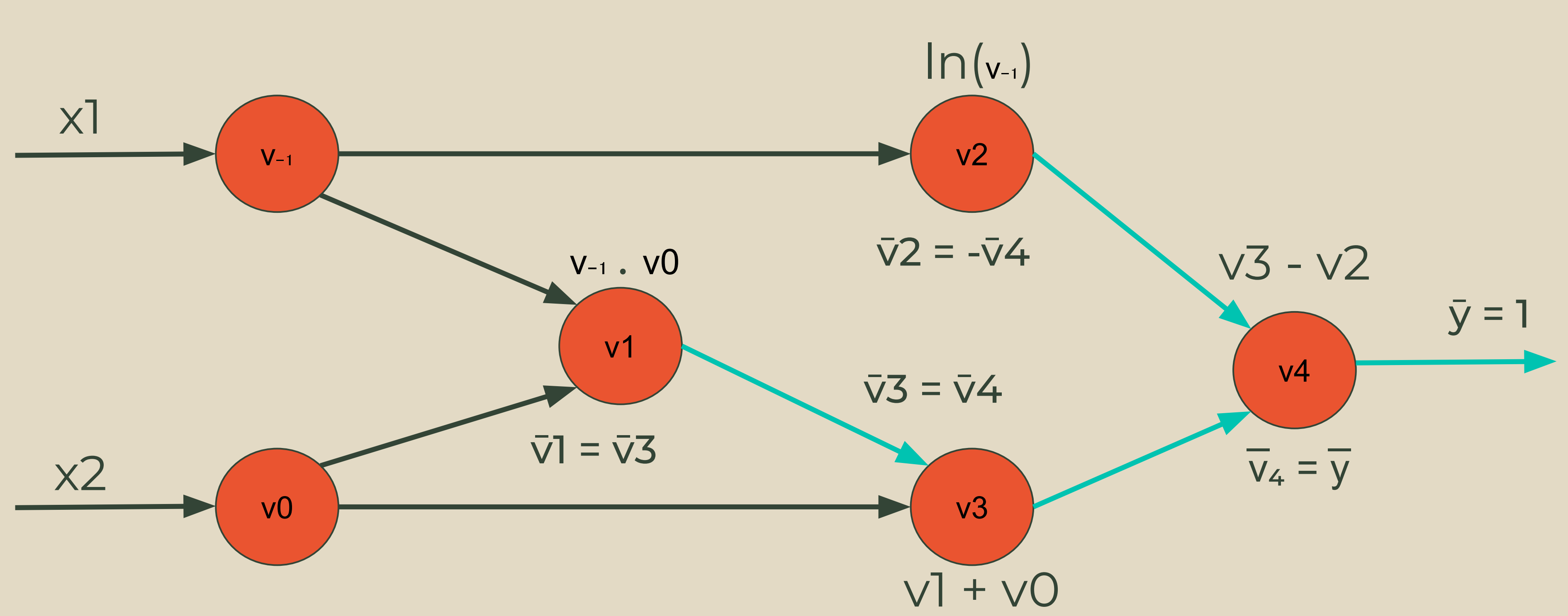
Reverse Mode AD



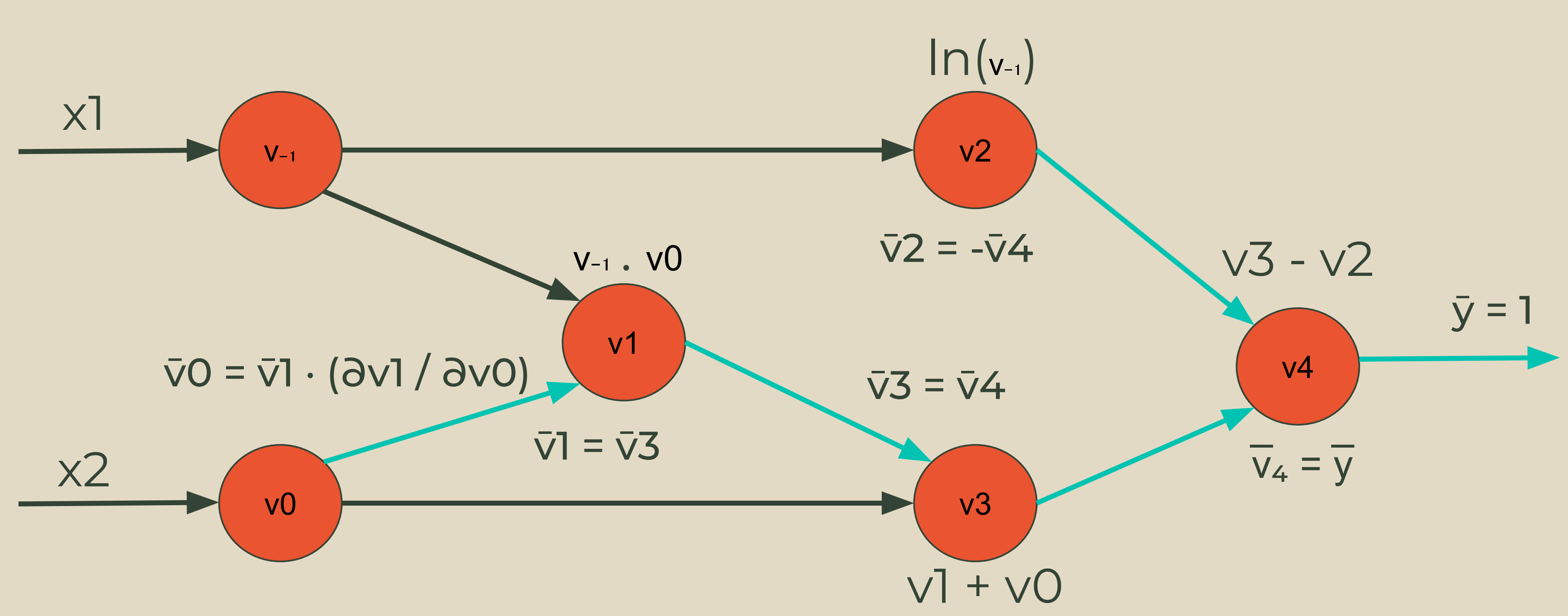
Reverse Mode AD



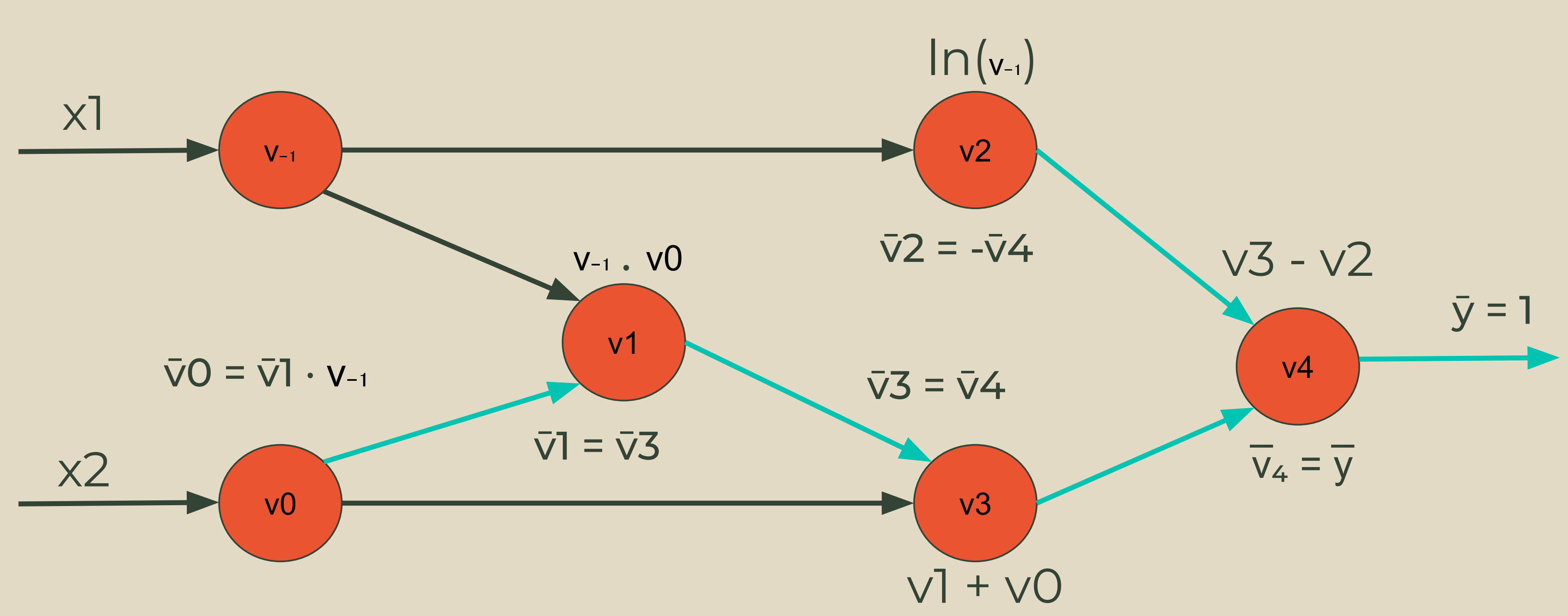
Reverse Mode AD



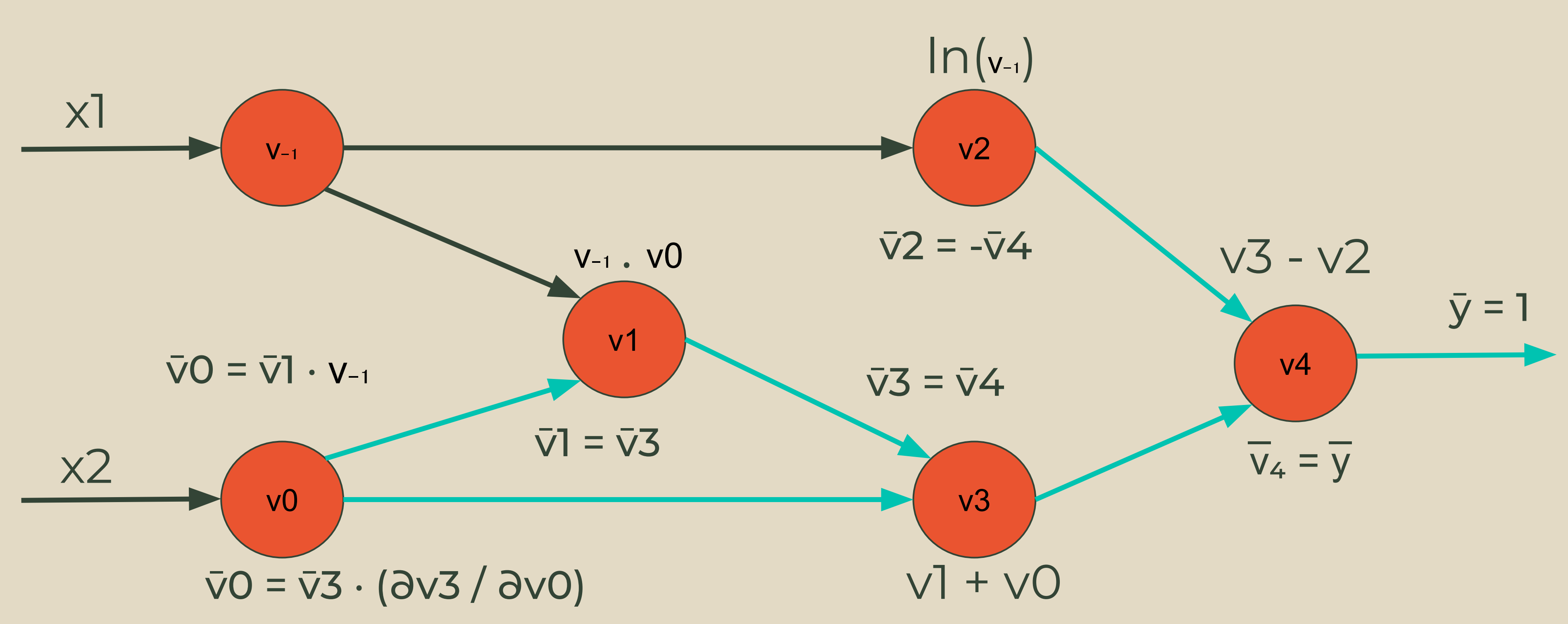
Reverse Mode AD



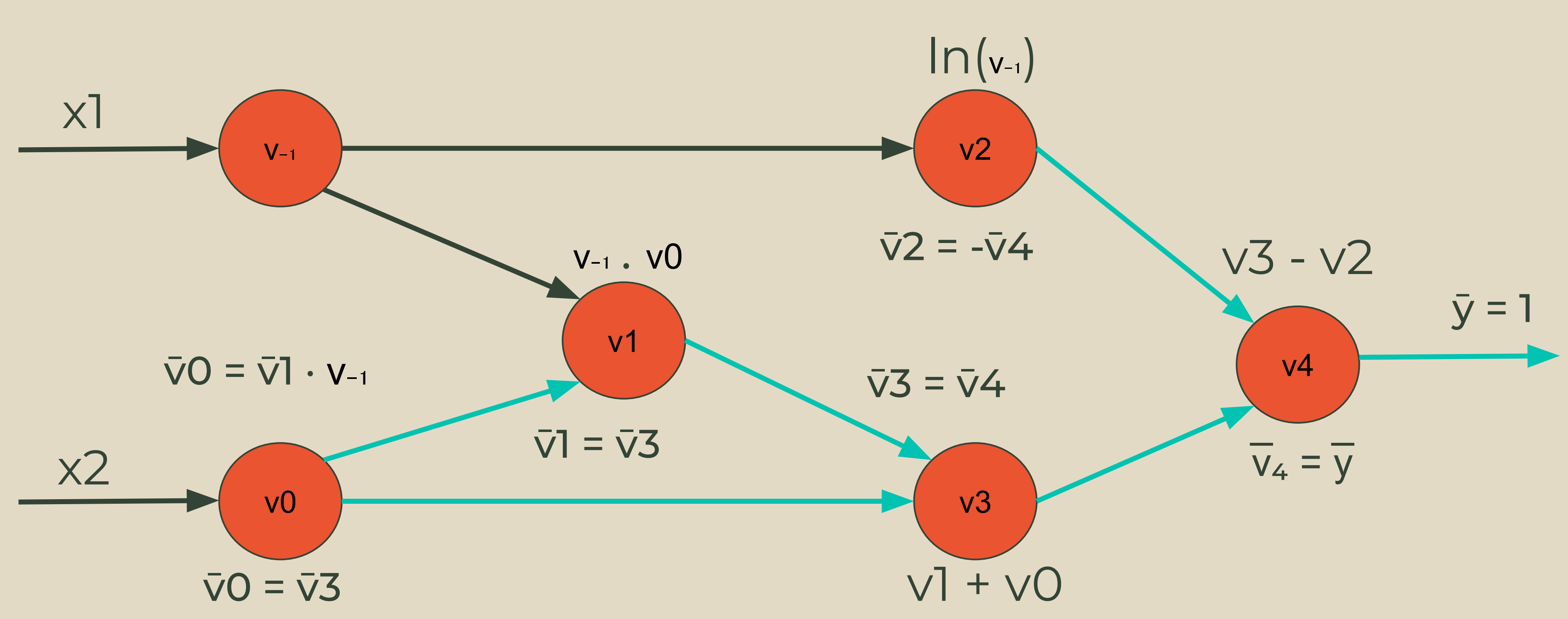
Reverse Mode AD



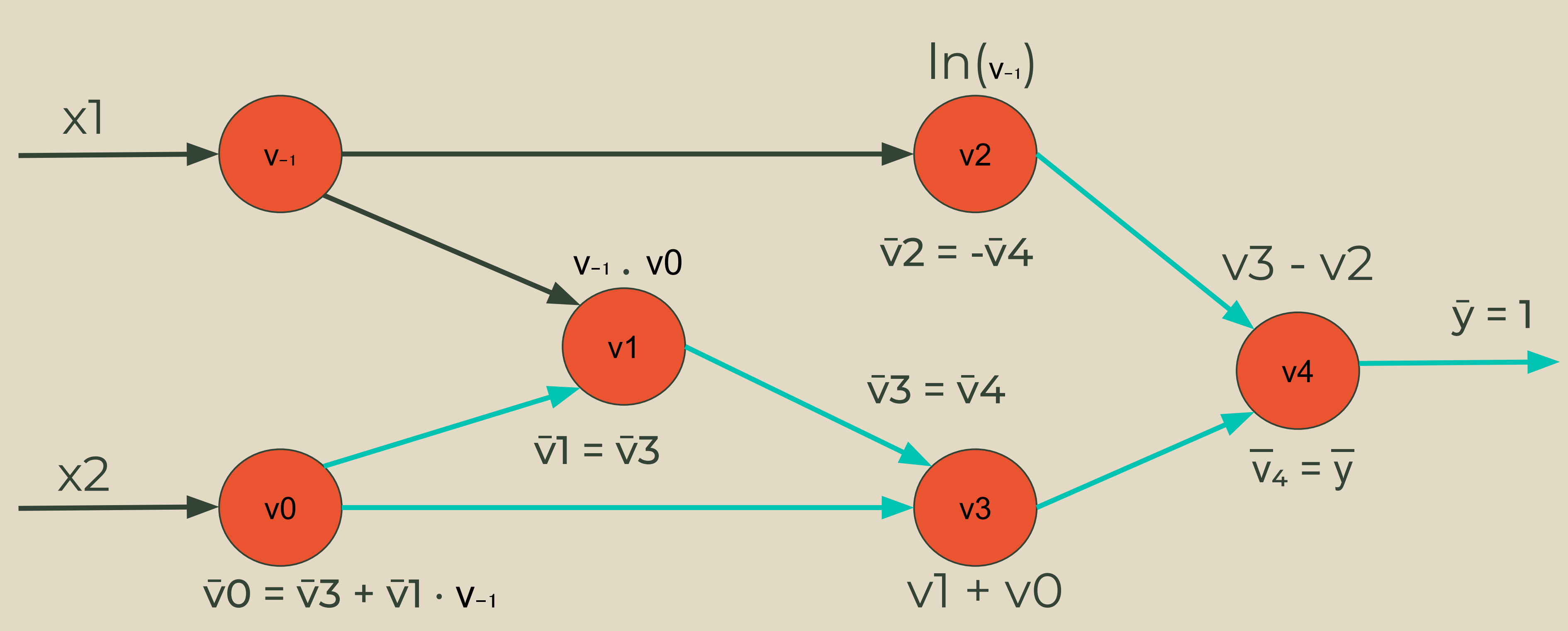
Reverse Mode AD



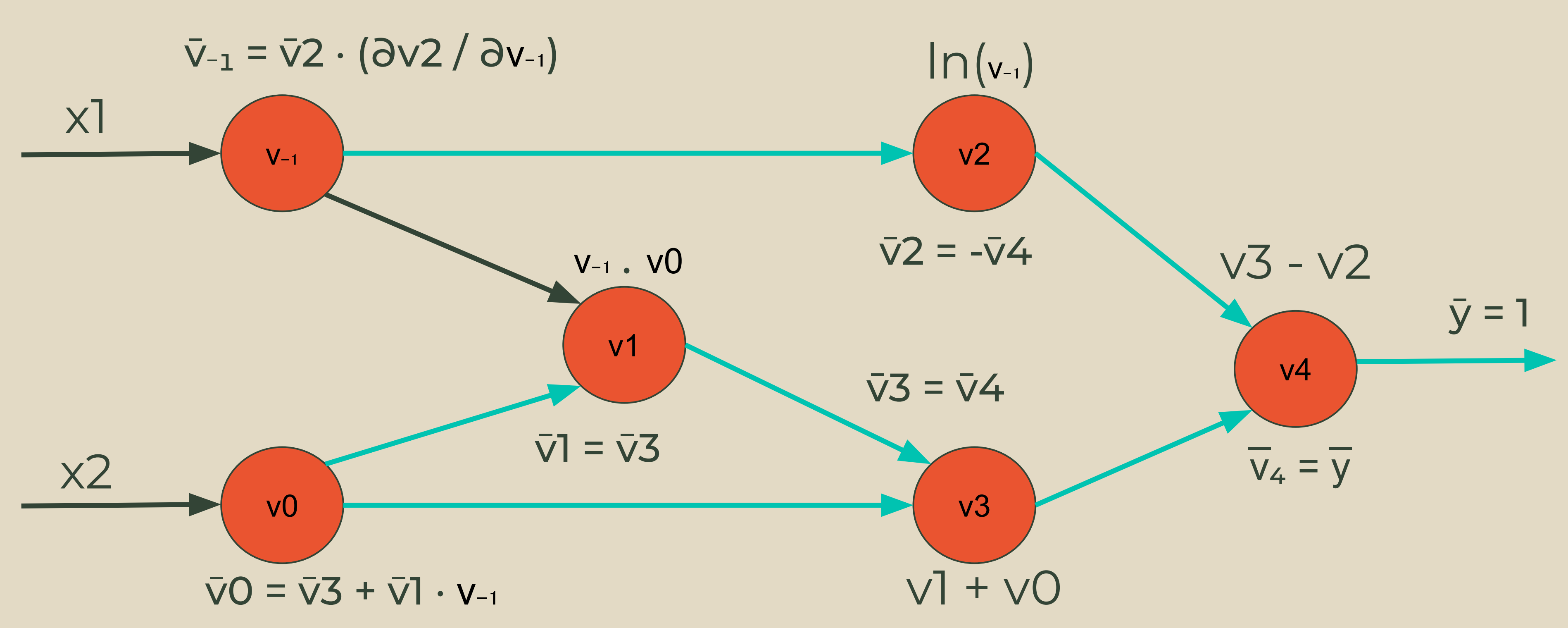
Reverse Mode AD



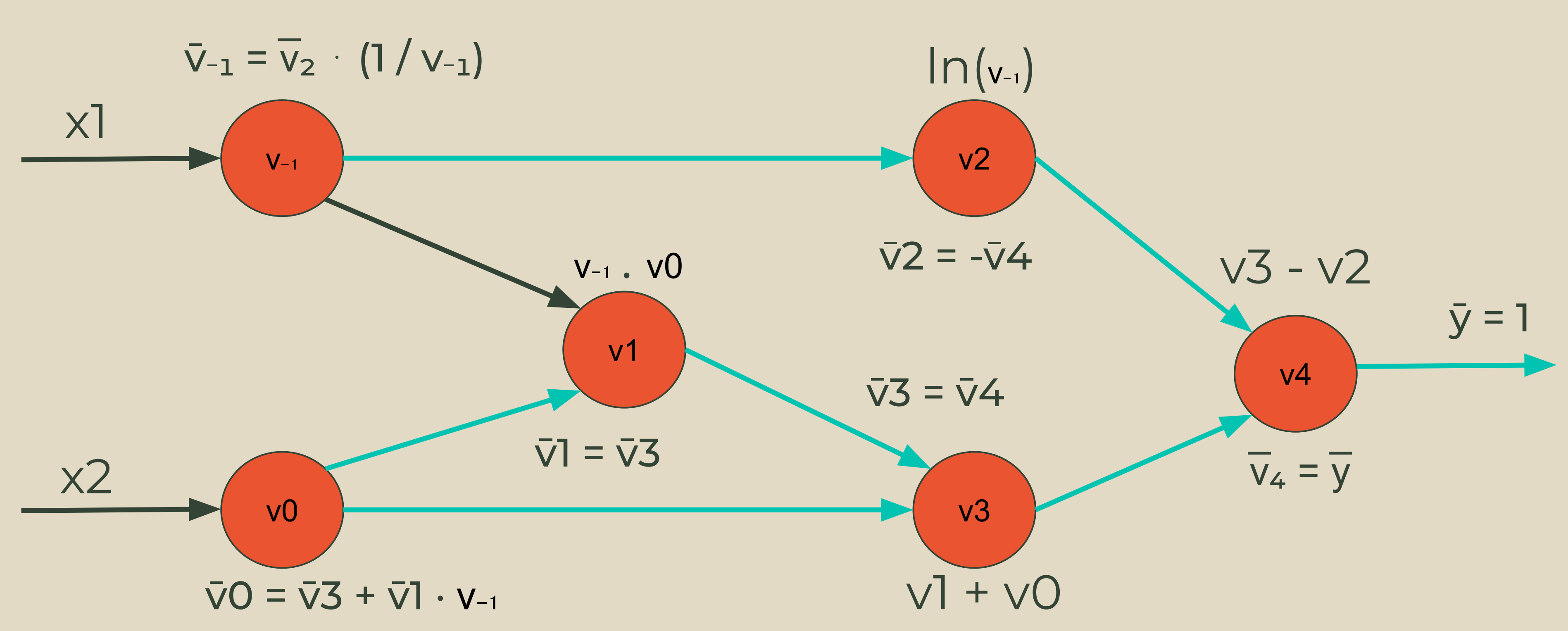
Reverse Mode AD



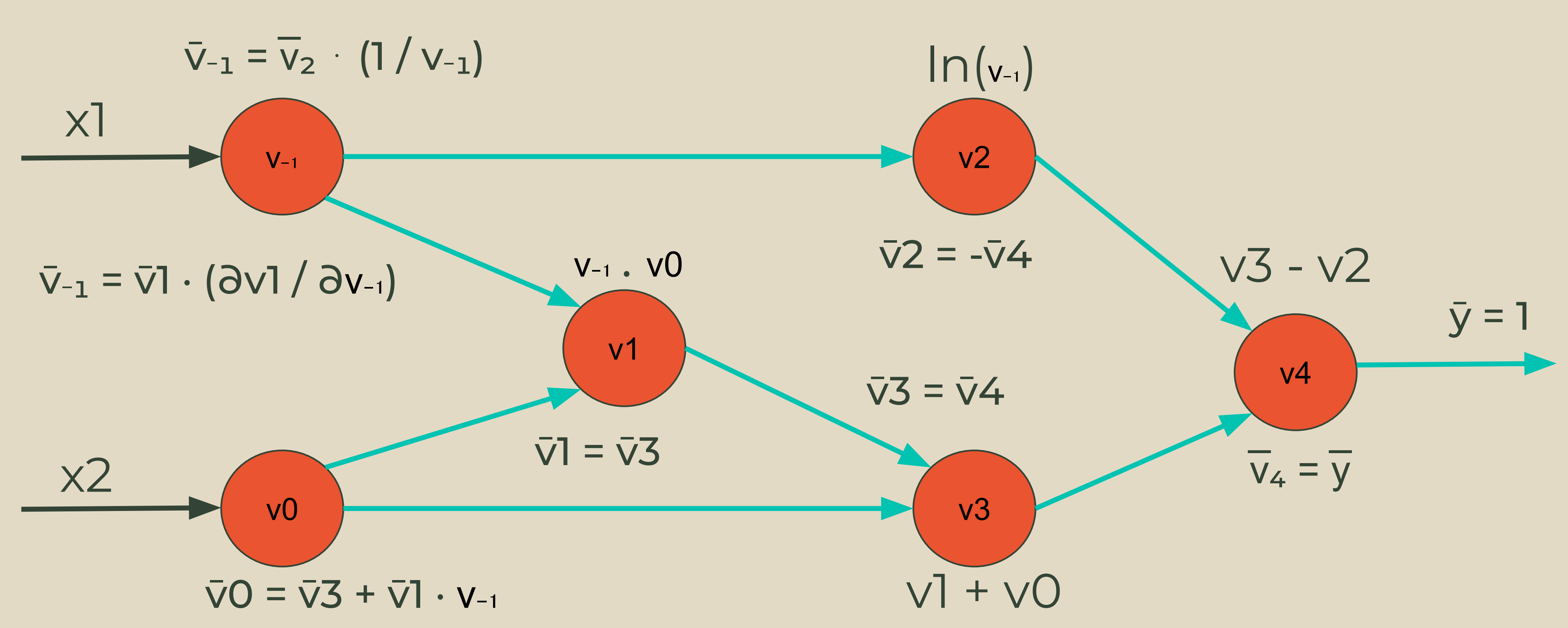
Reverse Mode AD



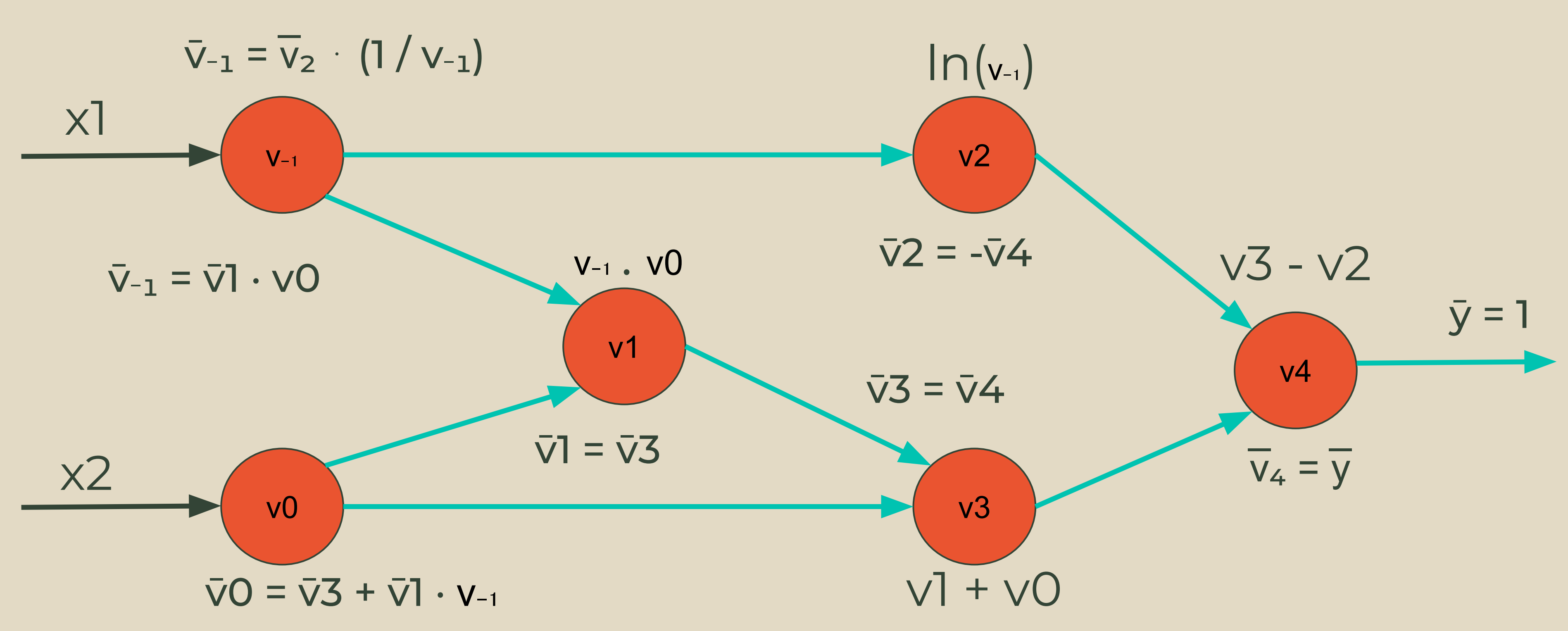
Reverse Mode AD



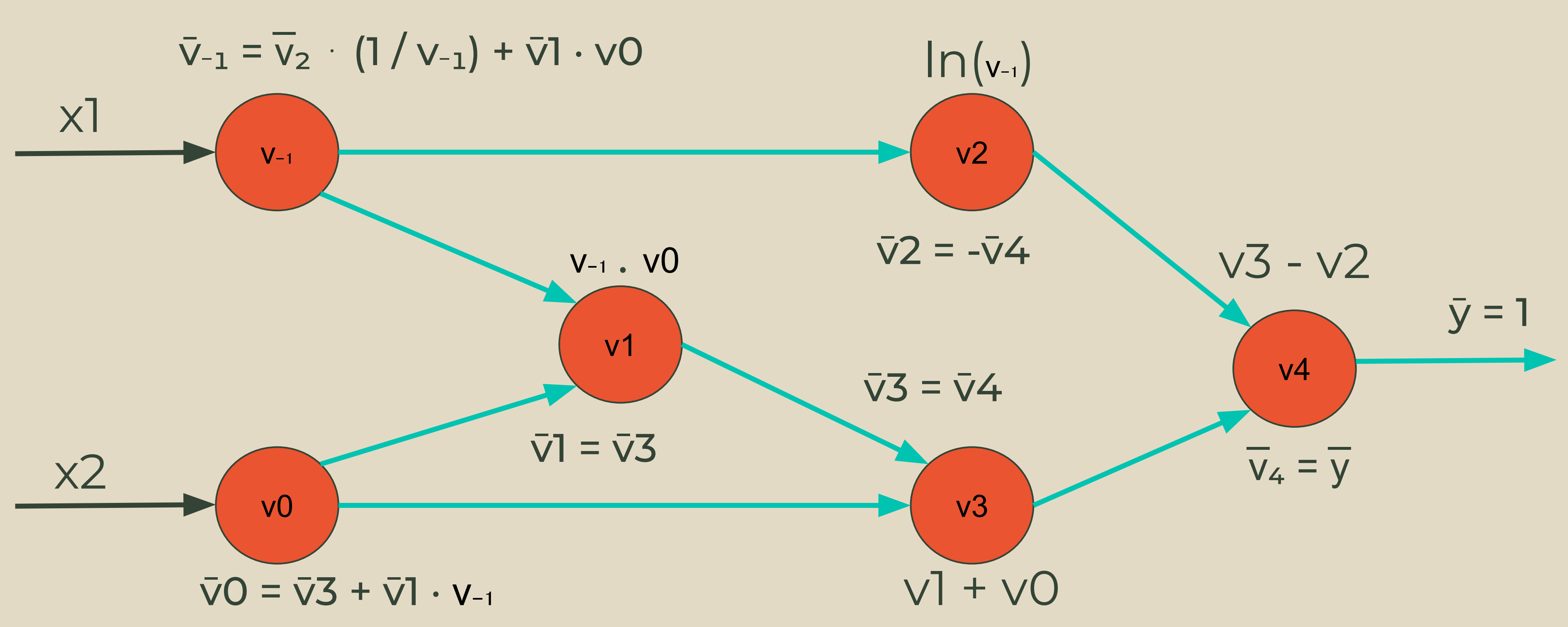
Reverse Mode AD



Reverse Mode AD



Reverse Mode AD



Reverse Mode AD

Forward Primal Trace	Output	Reverse Adjoint Trace	Output
$v_{-1} = x_1$	3	$\bar{v}_{-1} = \bar{x}_1 = \bar{v}_2 \cdot (1 / v_{-1}) + \bar{v}_1 \cdot v_0$	-1 · (1 / 3) + 1 · -4 = -4.33
$v_0 = x_2$	-4	$\bar{v}_0 = \bar{x}_2 = \bar{v}_3 \cdot 1 + \bar{v}_1 \cdot v_{-1}$	1 · 1 + 1 · 3 = 4
$v_1 = v_{-1} v_0$	3 · -4 = -12	$\bar{v}_1 = \bar{v}_3 \cdot 1$	1 · 1 = 1
$v_2 = \ln(v_{-1})$	ln(3) = 1.10	$\bar{v}_2 = \bar{v}_4 \cdot -1$	1 · -1 = -1
$v_3 = v_1 + v_0$	-12 + -4 = -16	$\bar{v}_3 = \bar{v}_4 \cdot 1$	1 · 1 = 1
$v_4 = v_3 - v_2$	-16 - 1.10 = -17.10	$\bar{v}_4 = \bar{y}$	1
$y = v_4$	-17.10	$\bar{y}$	1

Reverse Mode AD

Vector-Jacobian Product

$$\mathbf{r}^T \cdot \mathbf{J} = \begin{bmatrix} r_1 \dots r_m \end{bmatrix}^T \cdot \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Reverse Mode AD

IMPLEMENTATION

Forward vs Reverse Mode AD

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

Forward Mode

- m >> n
- No extra memory

Reverse Mode

- n >> m (appealing for ML)
- Requires a tape to store the forward pass values

In Summary we learnt

- Elementary Operations and their chain rules
- Computation Graph (Dynamic vs Static)
- Forward Mode AD
    - Dual Numbers
    - Operator Overloading
- Reverse Mode AD
    - Adjoint
    - Local Derivative
    - Wengert List (implicit in our implementation)

Applications

- Neural Network Training (Backpropagation)
- Hyperparameter optimization
- Probabilistic Inference
- Differentiable Solvers (Scientific Computing)
- Differentiable Rendering (Computer Graphics)
- Risk Sensitivity (Finance)
- ...

Where to go from here?

- Higher Order Derivatives
- Hack around in PyTorch/Tensorflow
- The Simple Essence of AD - Conal Elliott
- Reverse Mode AD in a functional framework - Lambda the ultimate backpropagator - Pearlmutter, Siskind
- Semantic preserving Differentiable Compilers to GPUs
- Automatic Vectorisation?

</EOM>