Automatic Differentiation

CSE 849 Deep Learning

Spring 2025

Zijun Cui

Slides are revised based on 1. CSC321 University of Toronto by Roger Grosse and Jimmy Ba; 2. CS340 UBC by Frank Wood

- Let’s first finish the backpropagation

- Scalar functions of Affine functions

$$D = f(\mathbf{z})$$

scalar

$d_z \times d_y$

$$\mathbf{z} = \mathbf{W}\mathbf{y} + \mathbf{b}$$

$d_z \times 1$ $\quad d_y \times 1$ $\quad d_z \times 1$

Matching Dimension:

$d_z \times d_y$

$$\nabla_{\mathbf{y}} D = \nabla_{\mathbf{z}}(D)\mathbf{W}$$

$1 \times d_y$ $\quad 1 \times d_z$

$$\nabla_{\mathbf{b}} D = \nabla_{\mathbf{z}}(D)$$

$1 \times d_z$ $\quad 1 \times d_z$

$d_y \times d_z$

$$\nabla_{\mathbf{W}} D = \mathbf{y}\nabla_{\mathbf{z}}(D)$$

$d_y \times 1$ $\quad 1 \times d_z$

Why? $\Delta D = \alpha \Delta \boldsymbol{y}$

scalar $\quad d_y \times 1$

So α must be $1 \times d_y$

and $\alpha = \nabla_{\boldsymbol{y}} D$

$\Delta D = \alpha \Delta \boldsymbol{W}$

scalar $\quad d_z \times d_y$

and $\alpha = \nabla_{\boldsymbol{W}} D$ with dimension $d_y \times d_z$

Wrong.....

- Scalar functions of Affine functions

$$\Delta D = \sum_{i,j} \frac{\partial D}{\partial W_{ij}} \Delta W_{ij}$$

Frobenius Inner Product for matrix

$$\mathbf{z} = \mathbf{W}\mathbf{y} + \mathbf{b}$$

We want to convert so that W becomes variables
That is why we want to transpose

$$\mathbf{z}^\top = \mathbf{y}^\top \mathbf{W}^\top + \mathbf{b}^\top$$

$$\nabla_{\boldsymbol{W}^\top} D = \nabla_{\boldsymbol{z}^\top} D\ \nabla_{\boldsymbol{W}^\top} \boldsymbol{z}^\top = \nabla_{\boldsymbol{z}^\top} D\ \ \mathbf{y}^\top$$

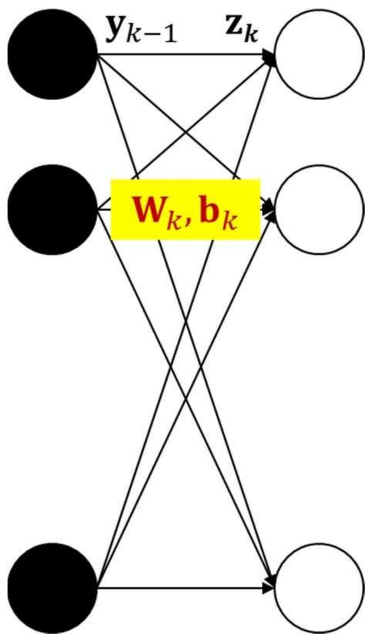
$$\nabla_{\boldsymbol{W}} D = (\nabla_{\boldsymbol{W}^\top} D)^\top = \mathbf{y} \nabla_{\boldsymbol{z}} D$$

Special Case: Application to a network

- Scalar functions of Affine functions

$$\mathbf{z}_k = \mathbf{W}_k \mathbf{y}_{k-1} + \mathbf{b}_k$$

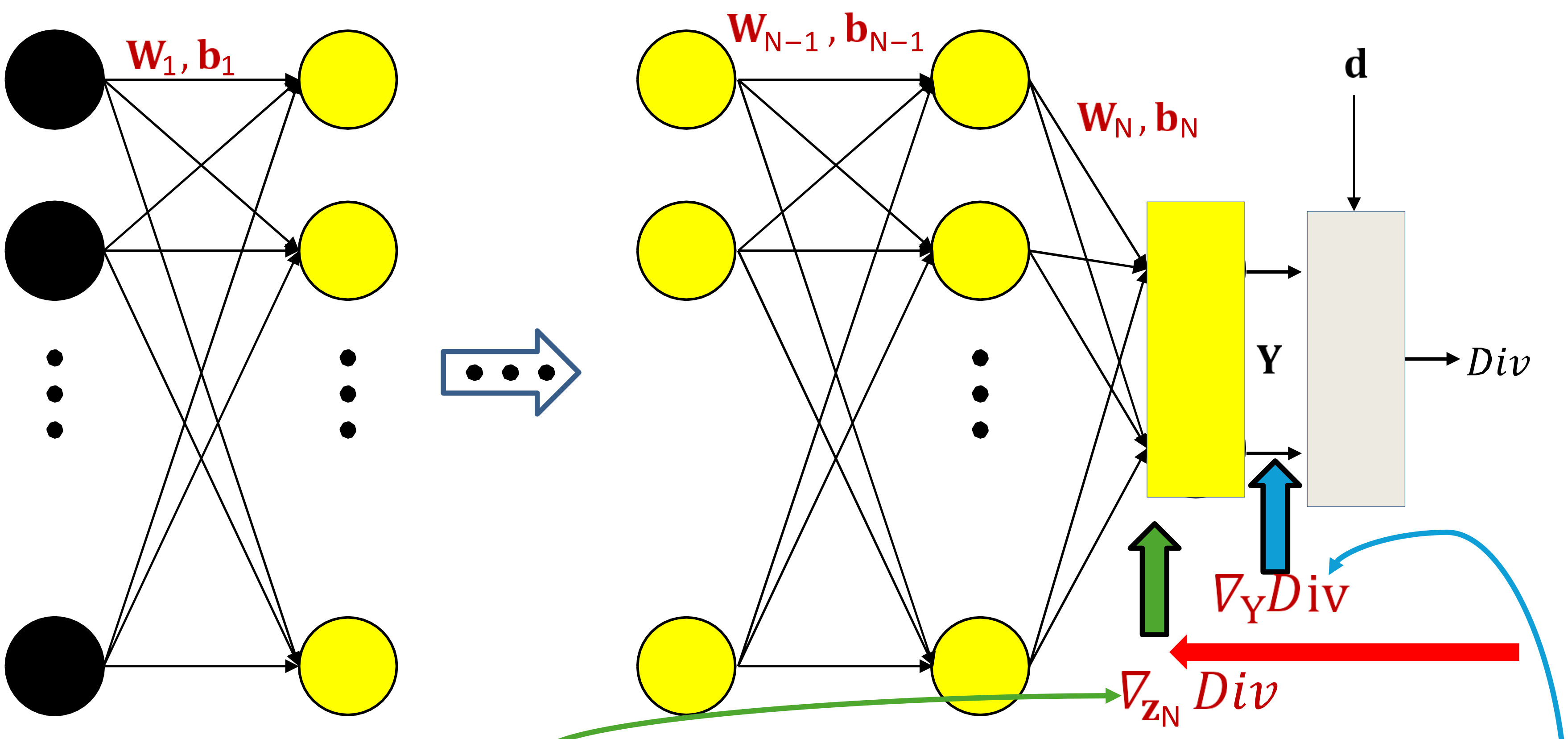
$$Div = Div(\mathbf{z}_k) \quad \Rightarrow \quad \nabla_{\mathbf{y}_{k-1}} Div = \nabla_{\mathbf{z}_k} Div \mathbf{W}_k$$



$$\nabla_{\mathbf{b}_k} Div = \nabla_{\mathbf{z}_k} Div$$

$$\nabla_{\mathbf{W}_k} D = \mathbf{y}_{k-1} \nabla_{\mathbf{z}_k} Div$$

The backward pass

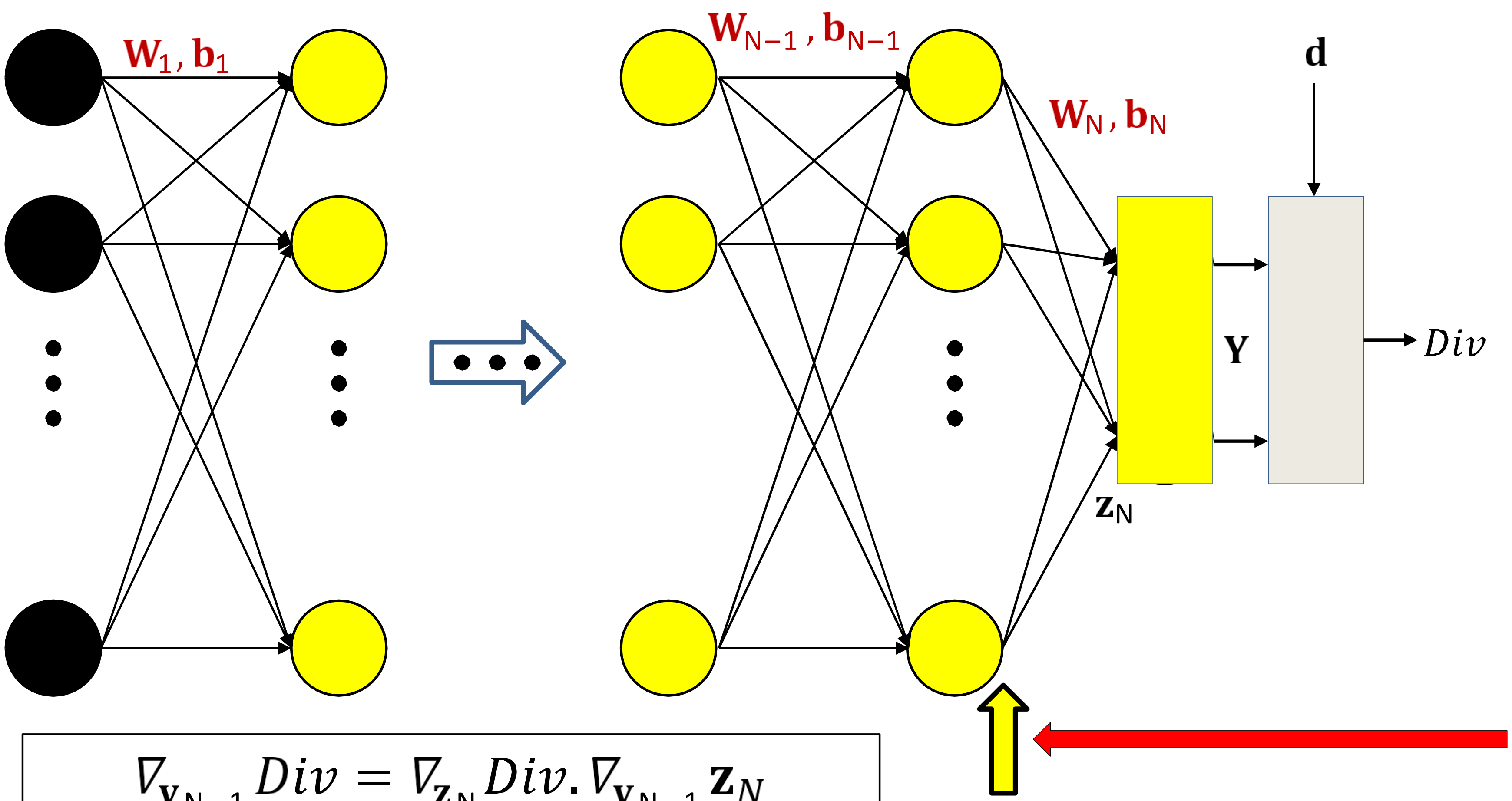


The divergence is a nested function: $Div(\mathbf{Y}(\mathbf{z}_N))$

$$\nabla_{\mathbf{z}_N} Div = \underbrace{\nabla_{\mathrm{Y}} Div}_{\text{Already computed}} . \underbrace{\nabla_{\mathbf{z}_N} \mathbf{Y}}_{\text{New term}} = \nabla_{\mathrm{Y}} Div . J_{\mathbf{Y}}(\mathbf{z}_N)$$

First compute the derivative of the divergence w.r.t Y. The actual derivative depends on the divergence function.

The backward pass

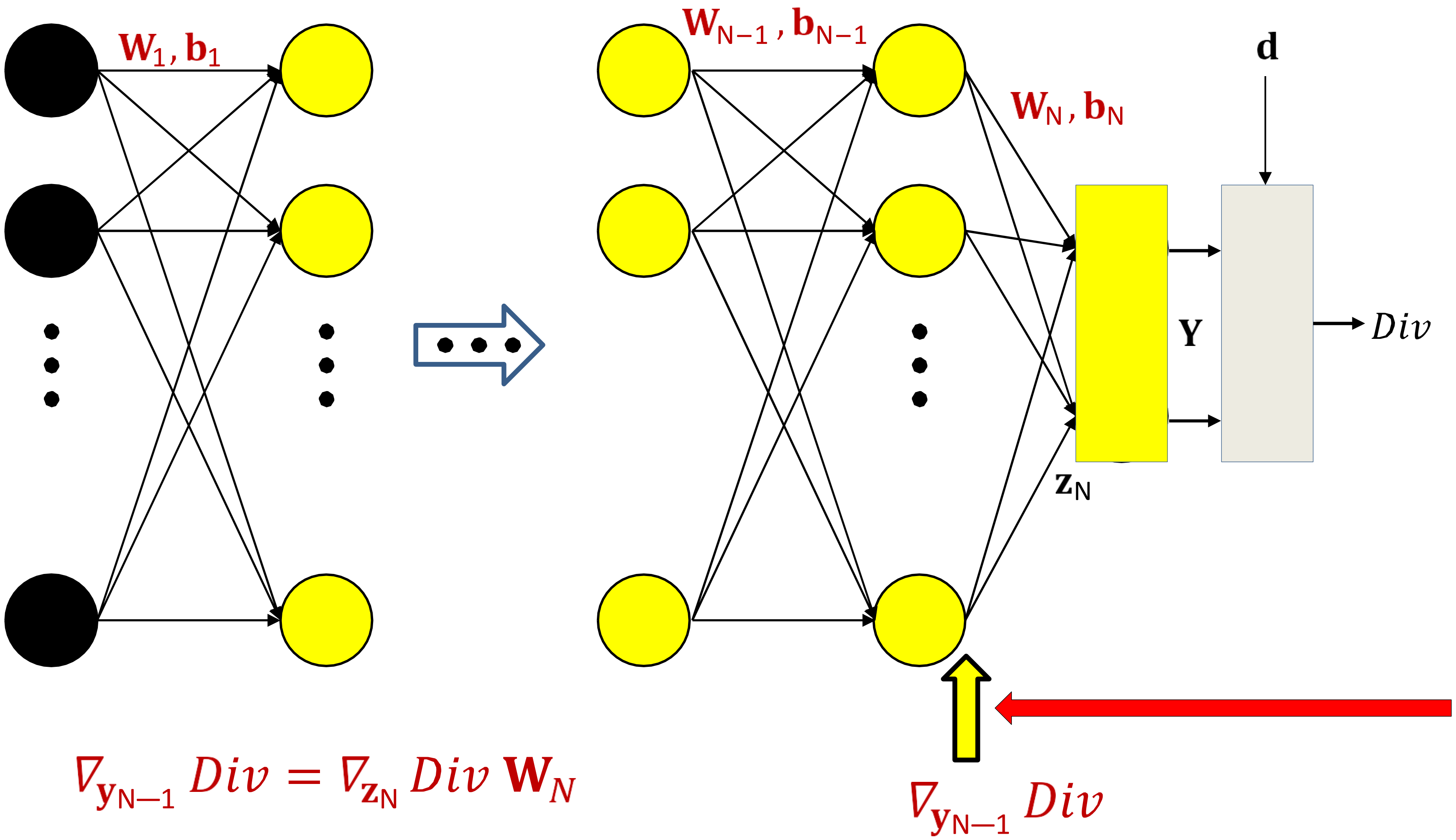


$$\boxed{\nabla_{\mathbf{y}_{N-1}} Div = \nabla_{\mathbf{z}_N} Div . \nabla_{\mathbf{y}_{N-1}} \mathbf{z}_N}$$

$$\mathbf{z}_N = \mathbf{W}_N \mathbf{y}_{N-1} + \mathbf{b}_N \quad \Rightarrow \quad \nabla_{\mathbf{y}_{N-1}} \mathbf{z}_N = \mathbf{W}_N$$

$$\nabla_{\mathbf{y}_{N-1}} Div = \underbrace{\nabla_{\mathbf{z}_N} Div}_{\text{Already computed}} \underbrace{\mathbf{W}_N}_{\text{New term}}$$

The backward pass

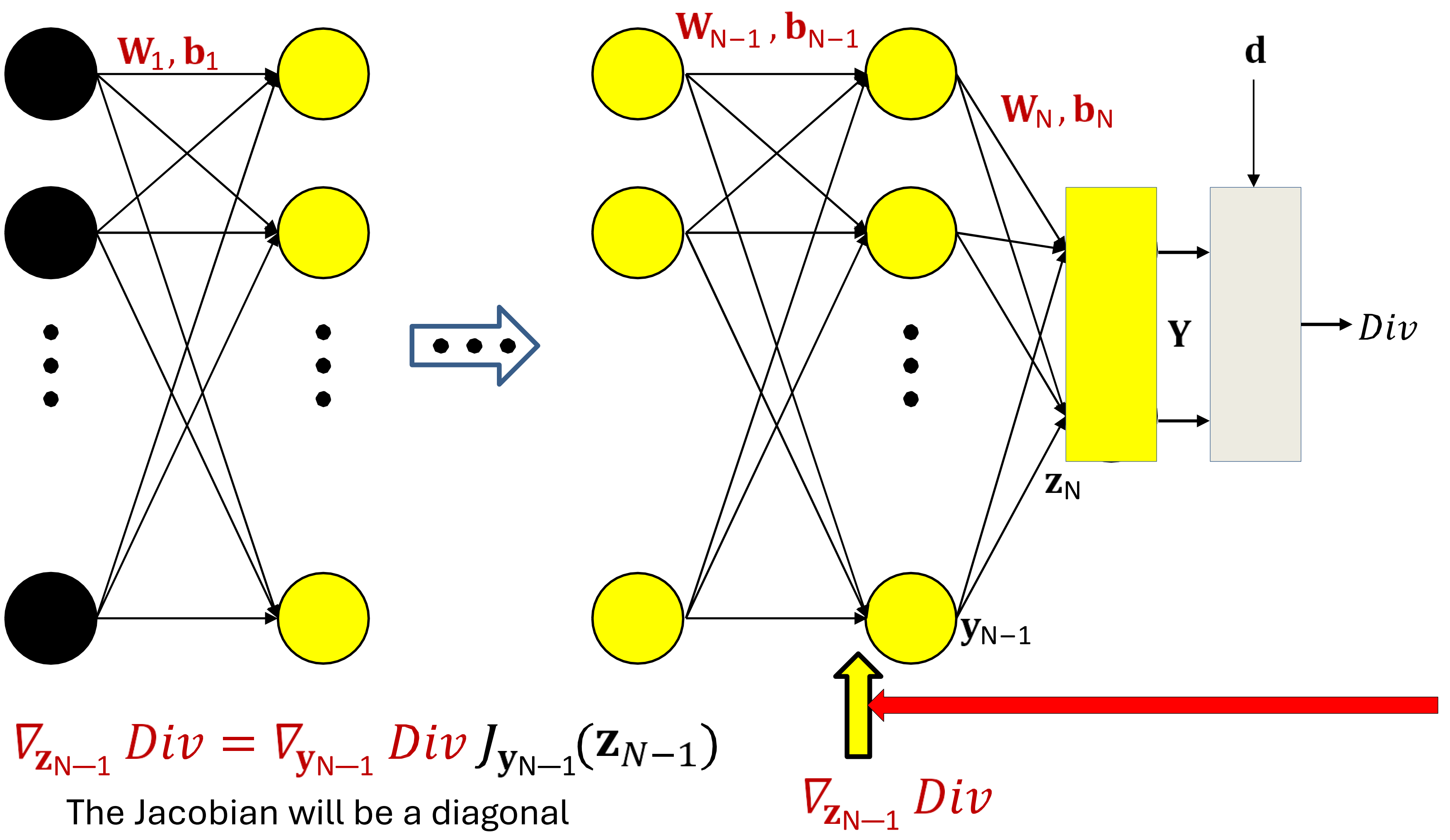


$$\nabla_{\mathbf{y}_{N-1}} Div = \nabla_{\mathbf{z}_N} Div\, \mathbf{W}_N$$

$$\boxed{\nabla_{\mathbf{W}_N} Div = \mathbf{y}_{N-1} \nabla_{\mathbf{z}_N} Div}$$

$$\boxed{\nabla_{\mathbf{b}_N} Div = \nabla_{\mathbf{z}_N} Div}$$

The backward pass

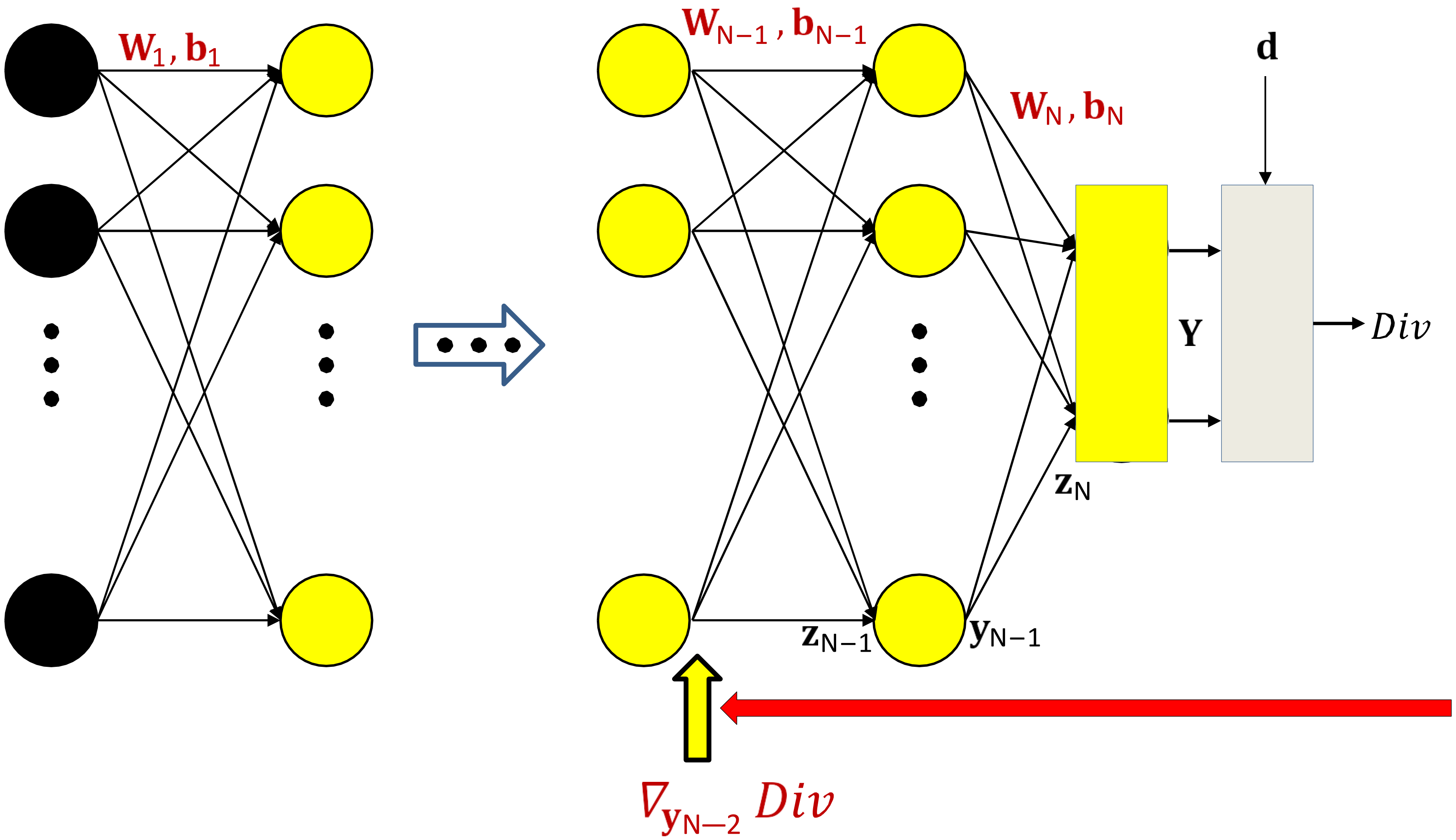


$$\nabla_{\mathbf{z}_{N-1}} Div = \nabla_{\mathbf{y}_{N-1}} Div \, J_{\mathbf{y}_{N-1}}(\mathbf{z}_{N-1})$$

The Jacobian will be a diagonal matrix for scalar activations

$$\boxed{\nabla_{\mathbf{z}_{N-1}} Div = \underbrace{\nabla_{\mathbf{y}_{N-1}} Div}_{\text{Already computed}} . \underbrace{\nabla_{\mathbf{z}_{N-1}} \mathbf{y}_{N-1}}_{\text{New term}}}$$

The backward pass



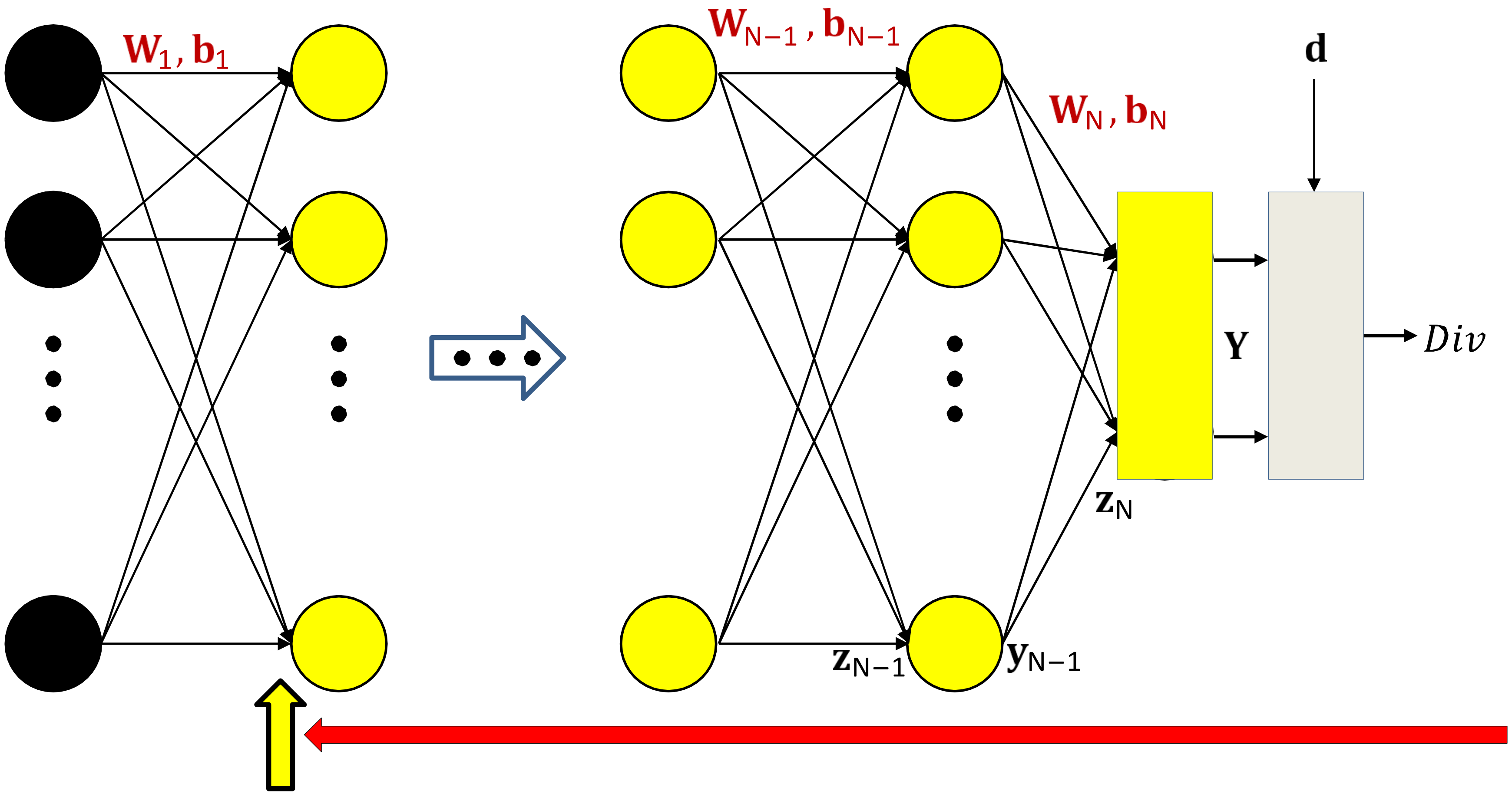
$$\boxed{\nabla_{\mathbf{y}_{N-2}} Div = \nabla_{\mathbf{z}_{N-1}} Div . \nabla_{\mathbf{y}_{N-2}} \mathbf{z}_{N-1}}$$

$$\Rightarrow \nabla_{\mathbf{y}_{N-2}} Div = \nabla_{\mathbf{z}_{N-1}} Div \, \mathbf{W}_{N-1}$$

$$\boxed{\nabla_{\mathbf{W}_{N-1}} Div = \mathbf{y}_{N-2} \nabla_{\mathbf{z}_{N-1}} Div}$$

$$\boxed{\nabla_{\mathbf{b}_{N-1}} Div = \nabla_{\mathbf{z}_{N-1}} Div}$$

The backward pass



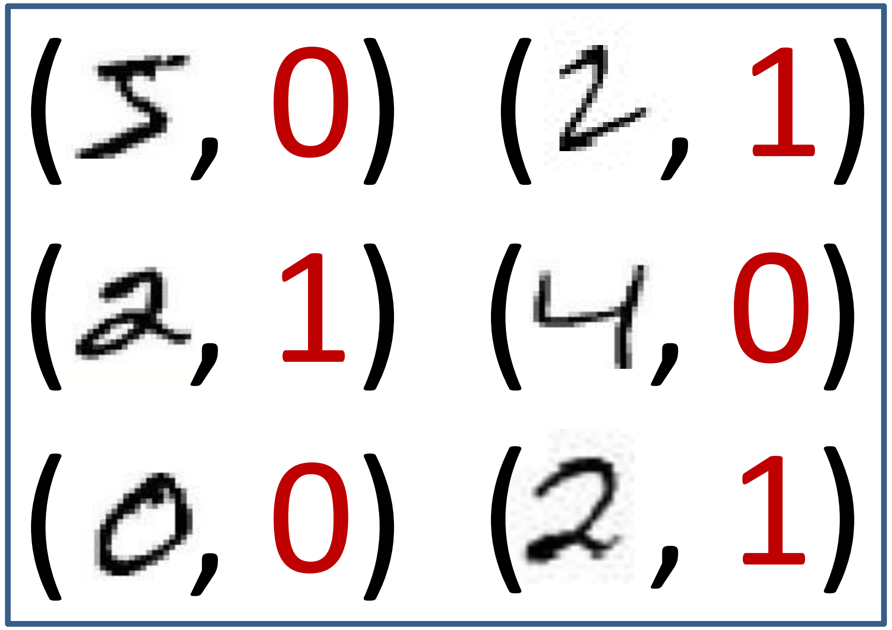
$$\nabla_{\mathbf{z}_1} Div = \nabla_{\mathbf{y}_1} Div \, J_{\mathbf{y}_1}(\mathbf{z}_1)$$

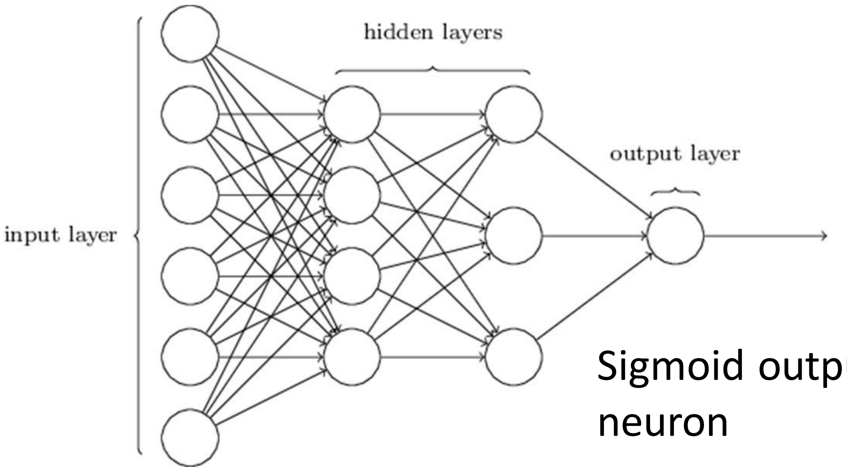
$$\nabla_{\mathbf{W}_1} Div = \mathbf{x} \nabla_{\mathbf{z}_1} Div$$

$$\nabla_{\mathbf{b}_1} Div = \nabla_{\mathbf{z}_1} Div$$

Setting up for digit recognition

Training data



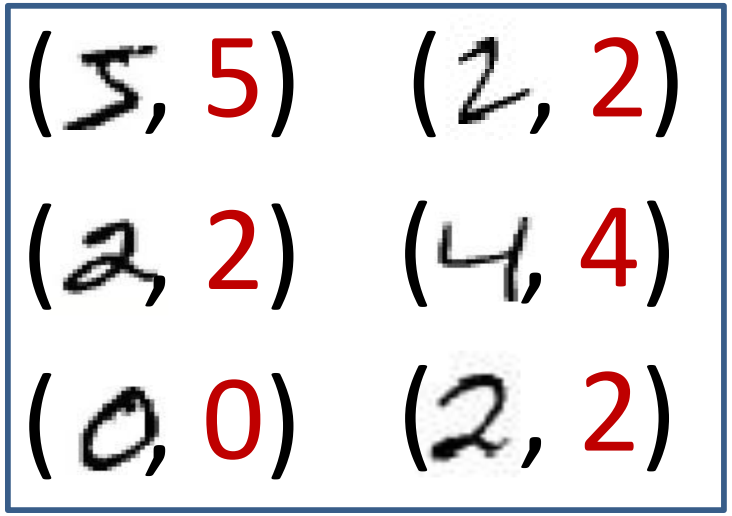


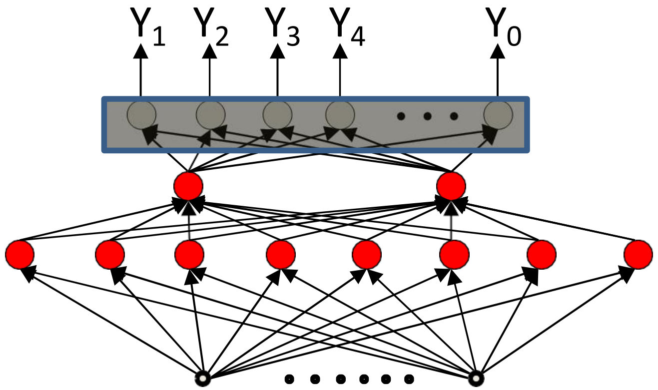
Sigmoid output neuron

- Simple Problem: Recognizing "2" or "not 2"
- Single output with sigmoid activation
  - $Y \in (0,1)$
  - d *is either* 0 *or* 1
- Use KL divergence
- Backpropagation to compute derivatives
  - To apply in gradient descent to learn network parameters

Recognizing the digit

Training data



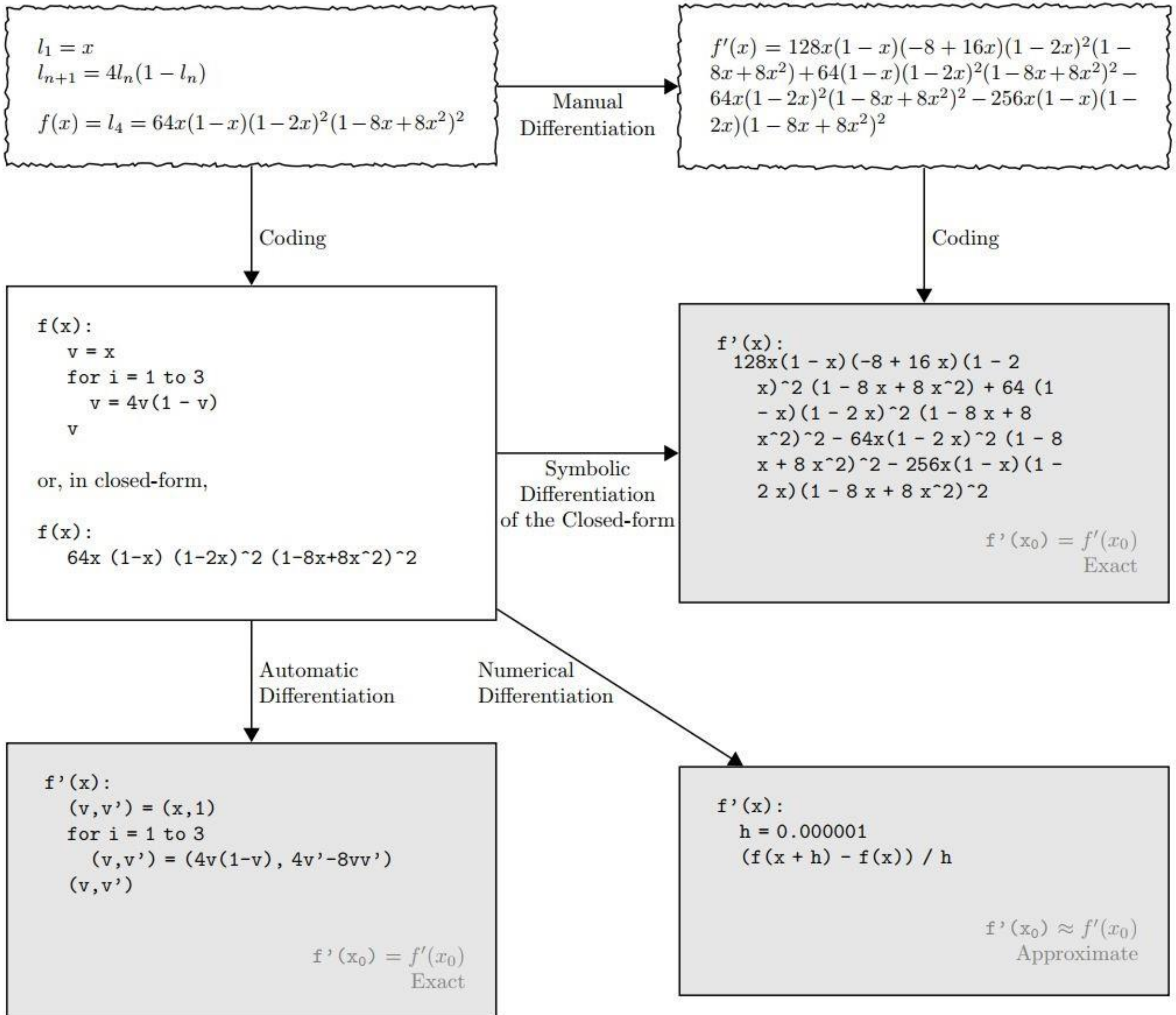


- More complex problem: Recognizing digit
- Network with 10 (or 11) outputs
  - First ten outputs correspond to the ten digits
    - Optional 11th is for none of the above
- Softmax output layer:
  - Ideal output: One of the outputs goes to 1, the others go to 0
- Backpropagation with KL divergence
  - To compute derivatives for gradient descent updates to learn network

- Back to today’s topic on Automatic Differentiation

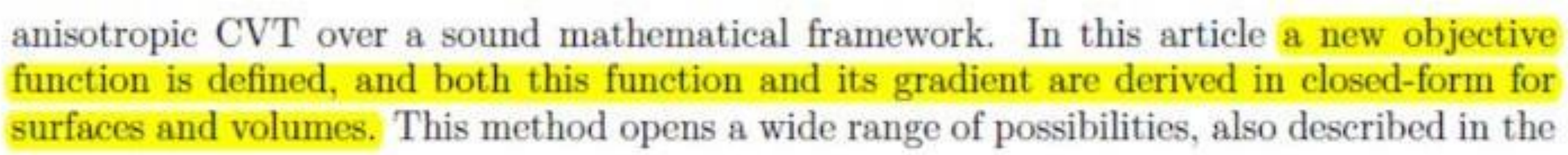
Derivatives as code

We can compute the derivatives **not just of mathematical functions, but of general programs** (with control flow)

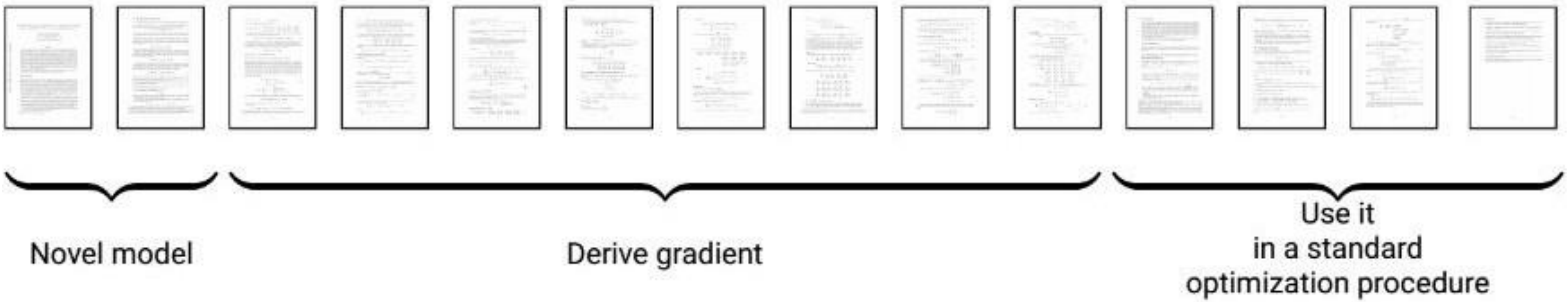


Manual Differentiation

You can see papers like this:



anisotropic CVT over a sound mathematical framework. In this article a new objective function is defined, and both this function and its gradient are derived in closed-form for surfaces and volumes. This method opens a wide range of possibilities, also described in the



Analytic derivatives are needed for **theoretical insight**

- analytic solutions, proofs
- mathematical analysis, e.g., stability of fixed points

Unnecessary when we just need derivative evaluations for optimization

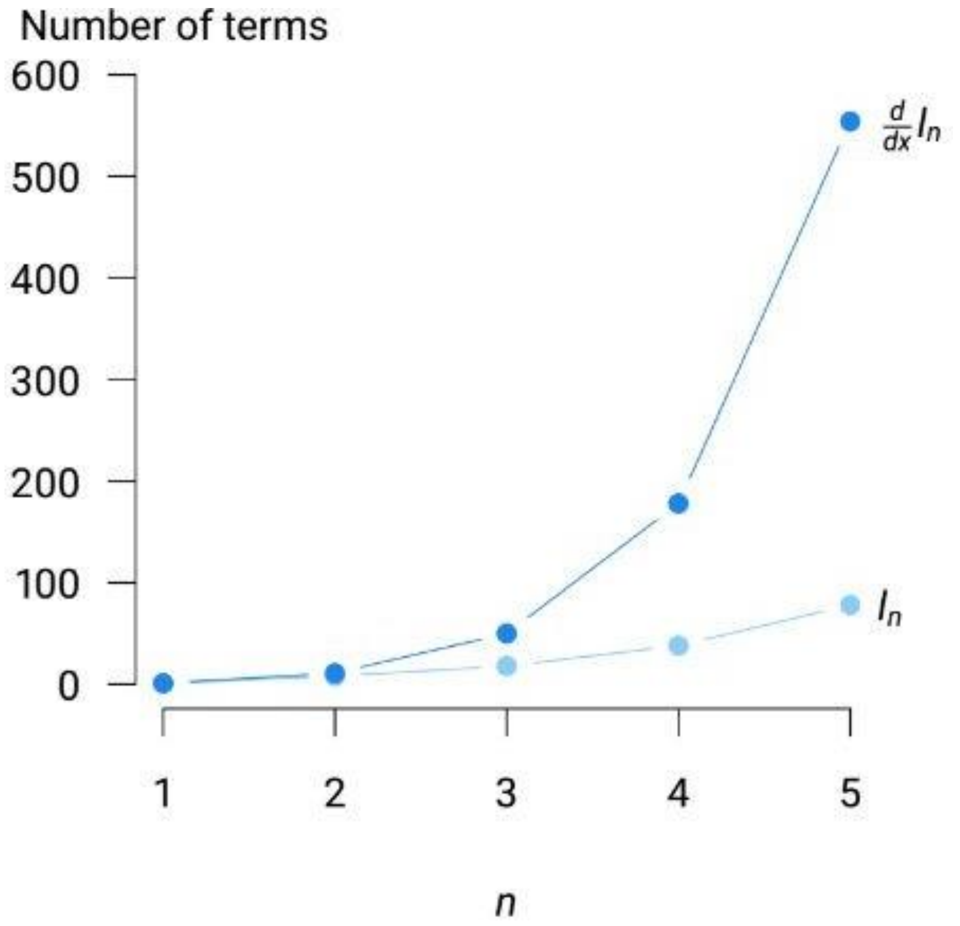
Symbolic differentiation

Symbolic computation with Mathematica, Maple, Maxima, and deep learning frameworks such as Theano

Problem: expression swell

Logistic map $l_{n+1} = 4l_n(1 - l_n)$, $l_1 = x$

n	l_n	$\frac{d}{dx} l_n$
1	x	1
2	$4x(1-x)$	$4(1-x) - 4x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2 - 16x(1-2x)^2 - 64x(1-x)(1-2x)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$



Symbolic differentiation

- Mathematica's derivatives for one layer of soft ReLU (univariate case):

```
D[Log[1 + Exp[w * x + b]], w]
```

Out[11]=
$$\frac{e^{b+w\,x}\,w}{1+e^{b+w\,x}}$$

- Derivatives for two layers of soft ReLU:

```
In[19]:= D[Log[1 + Exp[w2 * Log[1 + Exp[w1 * x + b1]] + b2]], w1]
```

Out[19]=
$$\frac{e^{b1+b2+w1\,x+w2\,\mathrm{Log}\left[1+e^{b1+w1\,x}\right]}\,w2\,x}{\left(1+e^{b1+w1\,x}\right)\left(1+e^{b2+w2\,\mathrm{Log}\left[1+e^{b1+w1\,x}\right]}\right)}$$

Symbolic differentiation

Problem: only applicable to **closed-form mathematical functions**

You can find the derivative of

```
In [1]: def f(x):
            return 64 *(1-x) *(1-2*x)^2 *(1-8*x+8*x*x)^2
```

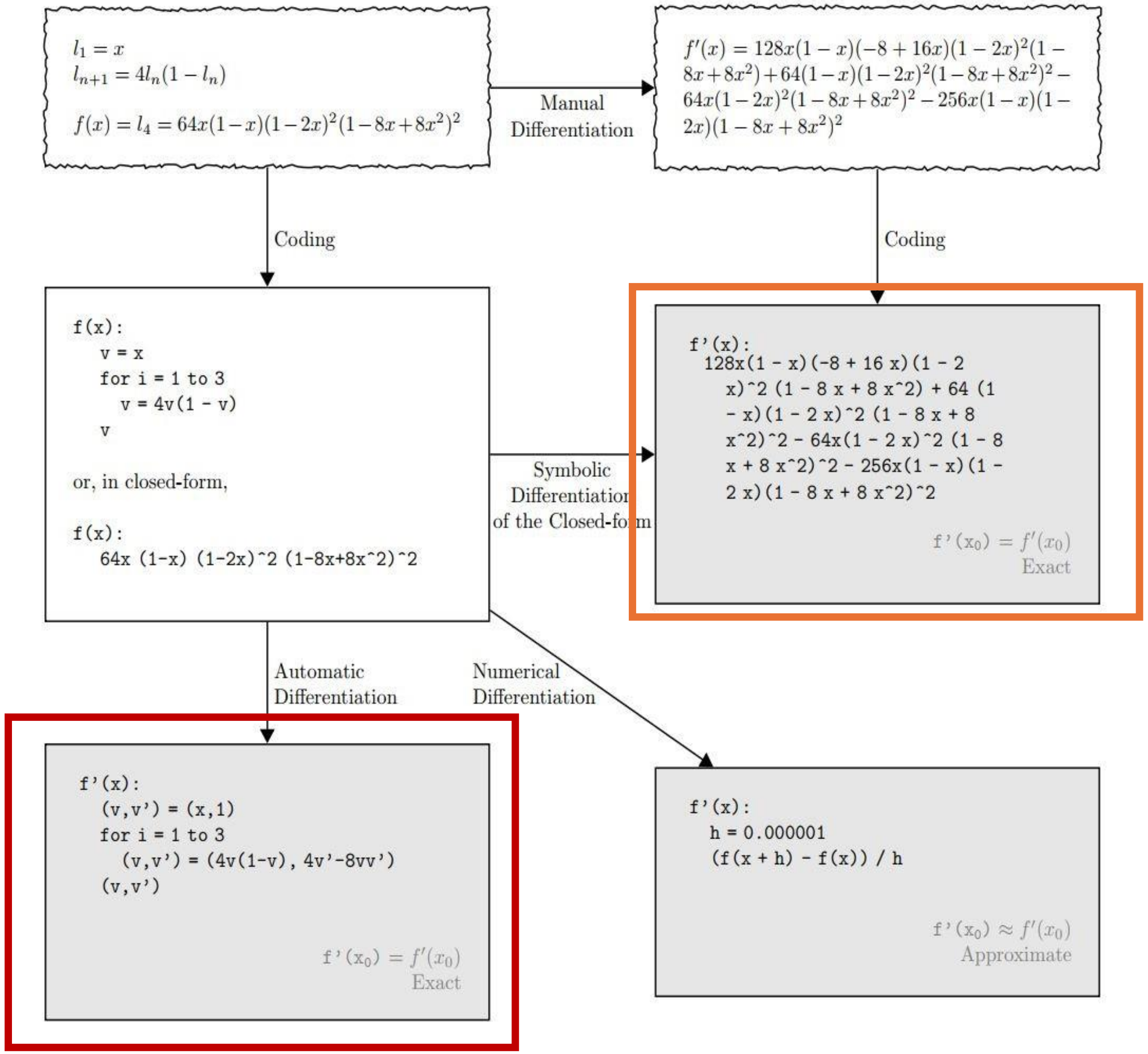
but not of

```
In [2]: def f(x,n):
            if n == 1:
                return x
            else:
                v = x
                for i in range(1,n):
                    v = 4*v*(1-v)
                return v
```

There might not be a convenient formula for the derivatives.

Autodiff Versus Symbolic differentiation

- The goal of autodiff is not a formula, but a procedure for computing derivatives.



Numerical differentiation

Finite difference approximation of ∇f, $f : \mathbb{R}^n \to \mathbb{R}$

$$\frac{\partial f(\mathbf{x})}{\partial x_i} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}, \quad 0 < h \ll 1$$

Problem: needs to be evaluated n times, once with each $\mathbf{e}_i \in \mathbb{R}^n$

Problem: we must select h and we face **approximation errors**

Numerical differentiation

Finite difference approximation of ∇f, $f : \mathbb{R}^n \to \mathbb{R}$

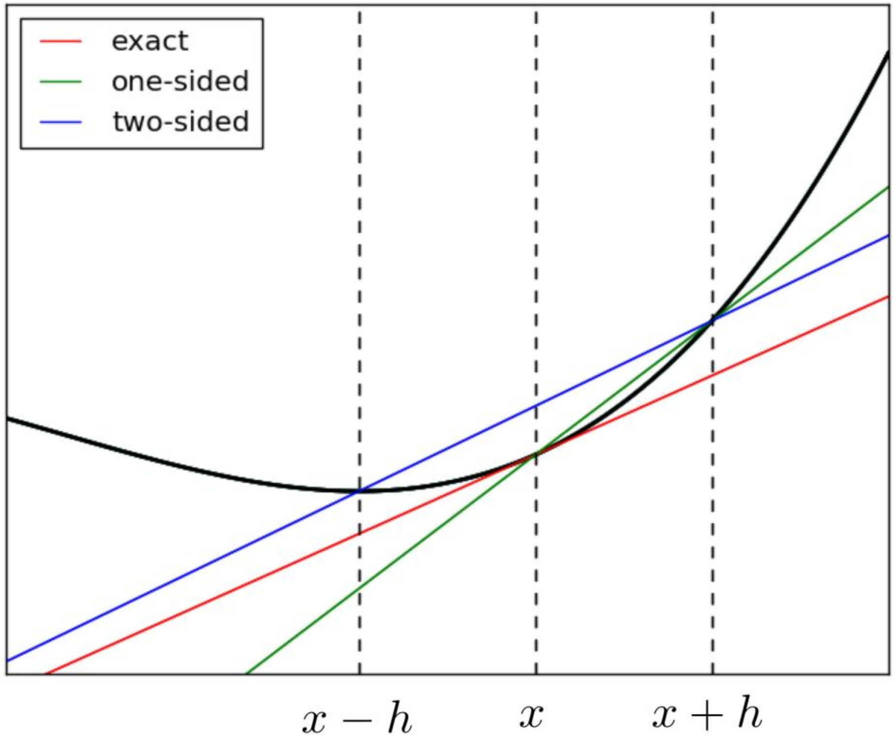
$$\frac{\partial f(\mathbf{x})}{\partial x_i} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}, \quad 0 < h \ll 1$$

Better approximations exist:

- Higher-order finite differences

  e.g., center difference:

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x} - h\mathbf{e}_i)}{2h} + O(h^2)$$



These increase rapidly in complexity
and **never completely eliminate the error**

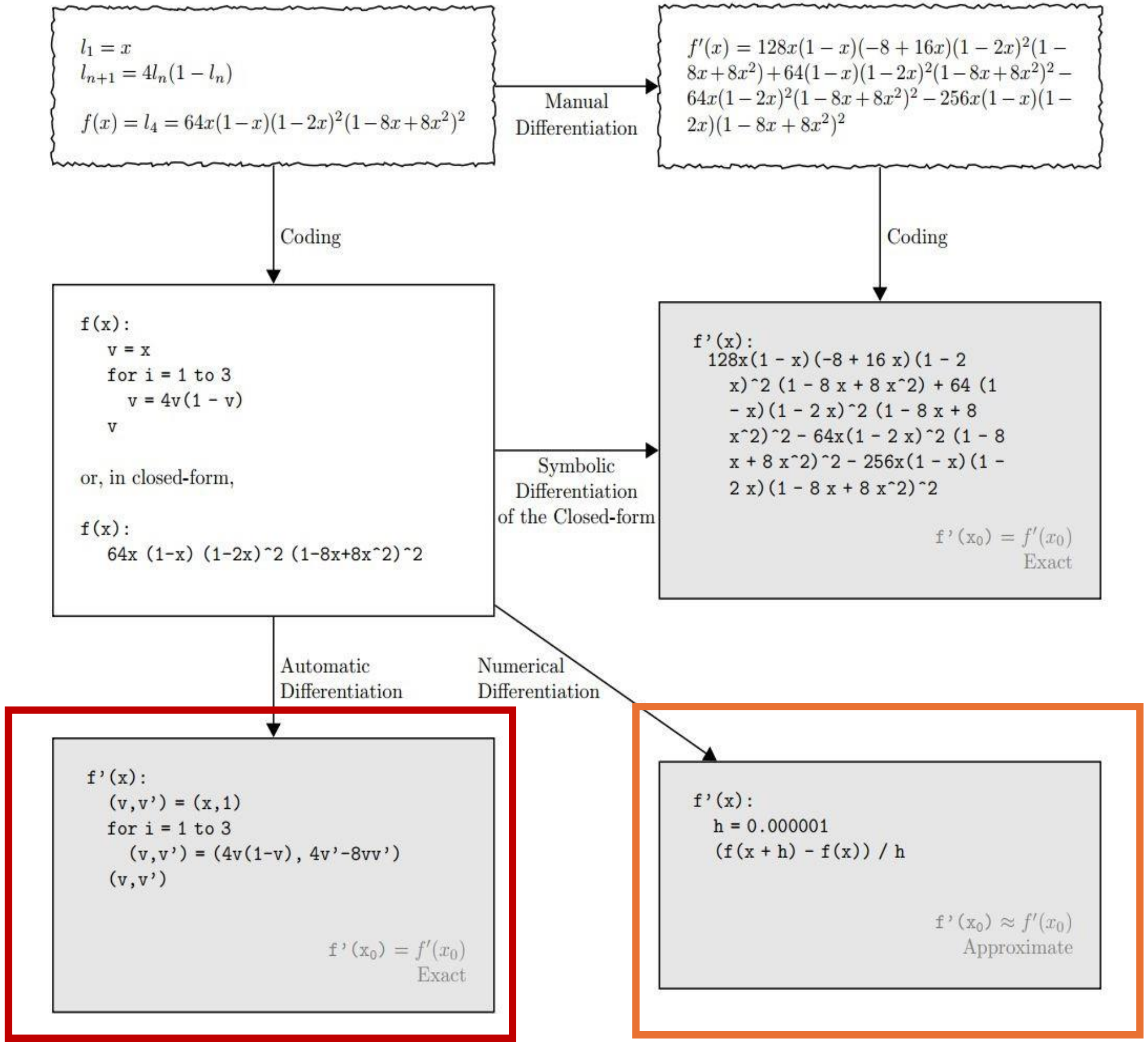
Autodiff Versus Finite Differences

Finite differences.

Still extremely useful as a **quick check of our gradient implementations**

Normally, we only use it for testing.

Autodiff is both efficient and numerically stable. **Is exact!**



Automatic differentiation

If we don't need analytic derivative expressions, we can **evaluate a gradient exactly** with only one forward and one reverse execution

$$f : \mathbb{R}^n \to \mathbb{R} \qquad \nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

In machine learning, this is known as **backpropagation** or "backprop"

- Automatic differentiation is more than backprop
- Or, backprop is a specialized *reverse mode* automatic differentiation

nature
International journal of science

Nature 323, 533–536 (9 October 1986)

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a

Confusing Terminology

- Automatic differentiation (autodiff) refers to a general way of taking a program which computes a value, and automatically constructing a procedure for computing derivatives of that value.

- Backpropagation is the special case of autodiff applied to neural nets

  But in machine learning, we often use backprop synonymously with autodiff

- Autograd is the name of a particular autodiff package.

  But lots of people started using “autograd” to mean “autodiff”

What Autodiff Is

An autodiff system will convert the program into a sequence of primitive operations which have specified routines for computing derivatives.

In this representation, backprop can be done in a completely mechanical way.

Original program:

$$z = wx + b$$

$$y = \frac{1}{1 + \exp(-z)}$$

$$L = \frac{1}{2}(y - t)^2$$

Sequence of primitive operations:

$$t_1 = wx$$

$$z = t_1 + b$$

$$t_3 = -z$$

$$t_4 = \exp(t_3)$$

$$t_5 = 1 + t_4$$

$$y = 1/t_5$$

$$t_6 = y - t$$

$$t_7 = t_6^2$$

$$L = t_7/2$$

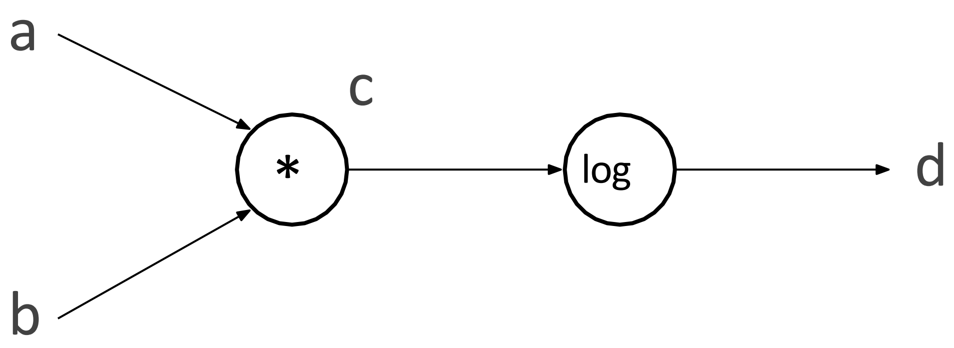
Automatic differentiation

All numerical algorithms, when executed, evaluate to compositions of a finite set of elementary operations with known derivatives

- Called a **trace** or a **Wengert list** (Wengert,1964)
- Alternatively represented as a **computational graph** showing dependencies

$$f(a, b) = \log(ab)$$

$$\nabla f(a, b) = (1/a, 1/b)$$



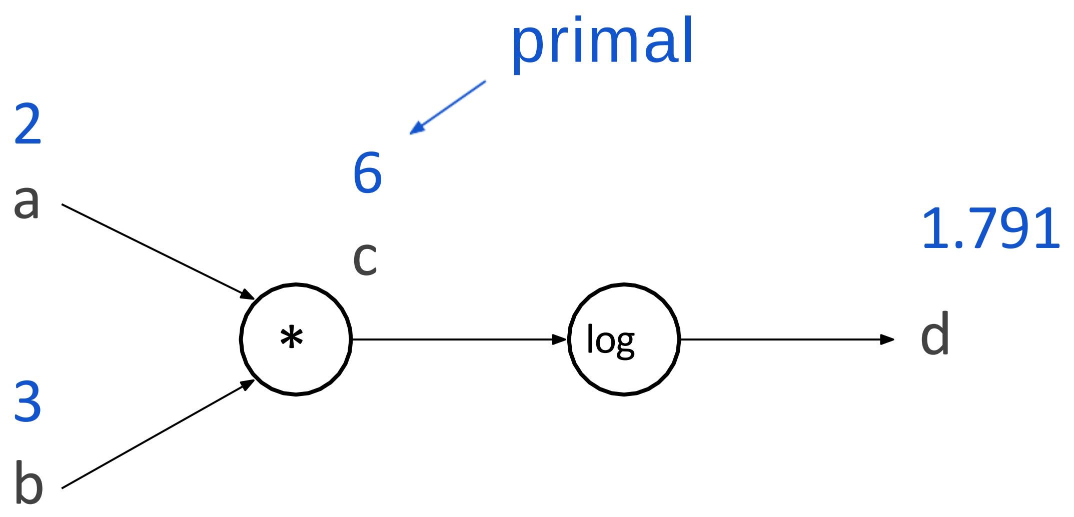

```
f(a, b):
    c = a * b
    d = log(c)
    return d
```

Automatic differentiation

Primal: The value computed during the forward pass of a computational graph

```
f(a, b):
    c = a * b
    d = log(c)
    return d
```

1.791 = f(2, 3)



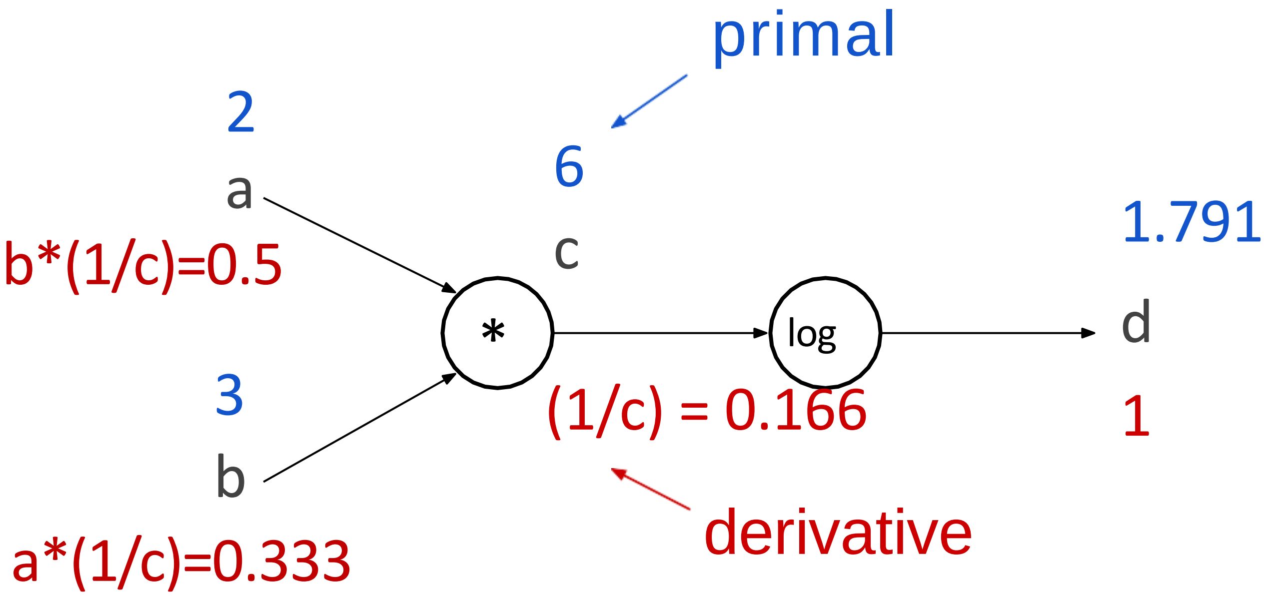
Automatic differentiation

```
f(a, b):
    c = a * b
    d = log(c)
    return d
```

1.791 = f(2, 3)

[0.5, 0.333] = f'(2, 3)

$$\nabla f(a, b) = (1/a, 1/b)$$



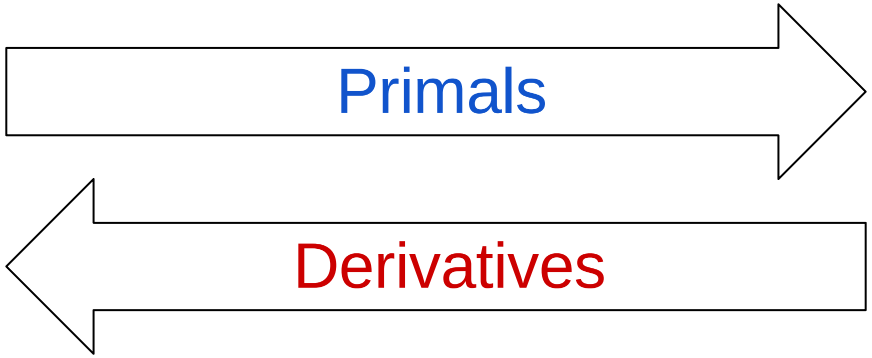
Automatic differentiation

Two main flavors

Forward mode

(Tangents)

Reverse mode (a.k.a. backprop)



(Adjoints)

Nested combinations

(higher-order derivatives, Hessian–vector products, etc.)

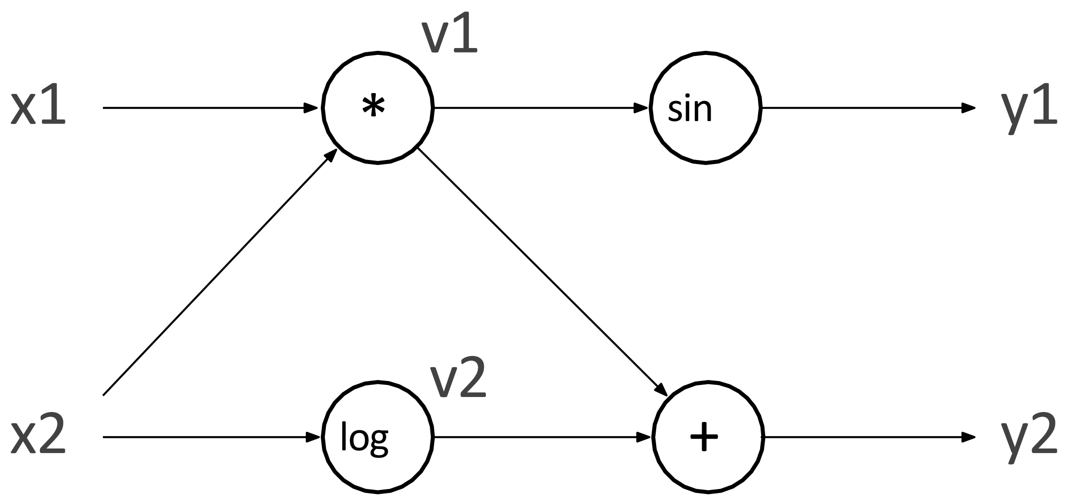
- Forward-on-reverse
- Reverse-on-forward
- ...

Forward mode

Primals
Derivatives (tangents)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```



Forward mode

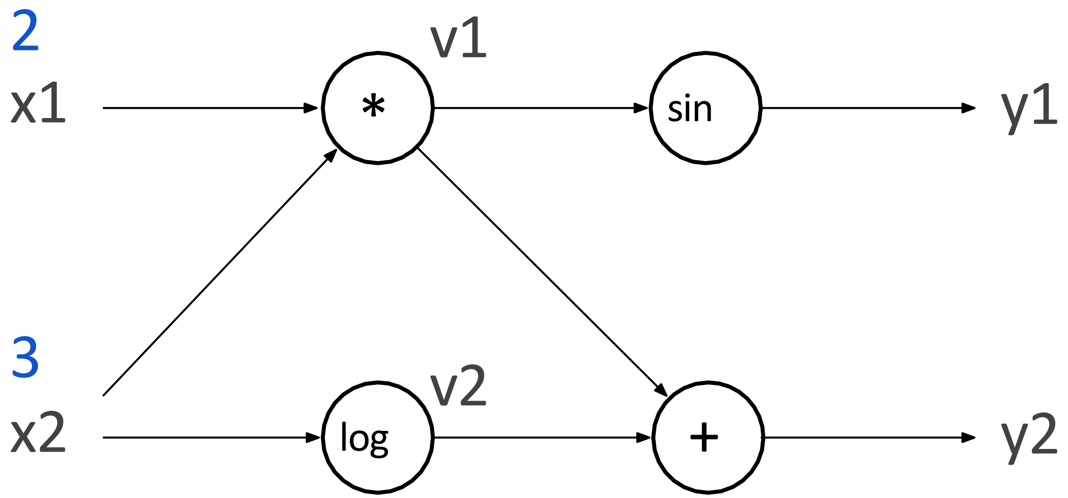
Primals

Derivatives (tangents)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =   x1 * x2
    v2  =   log(x2)
    y1  =   sin(v1)
    y2  =   v1 + v2
    return (y1, y2)
```

f(2, 3)



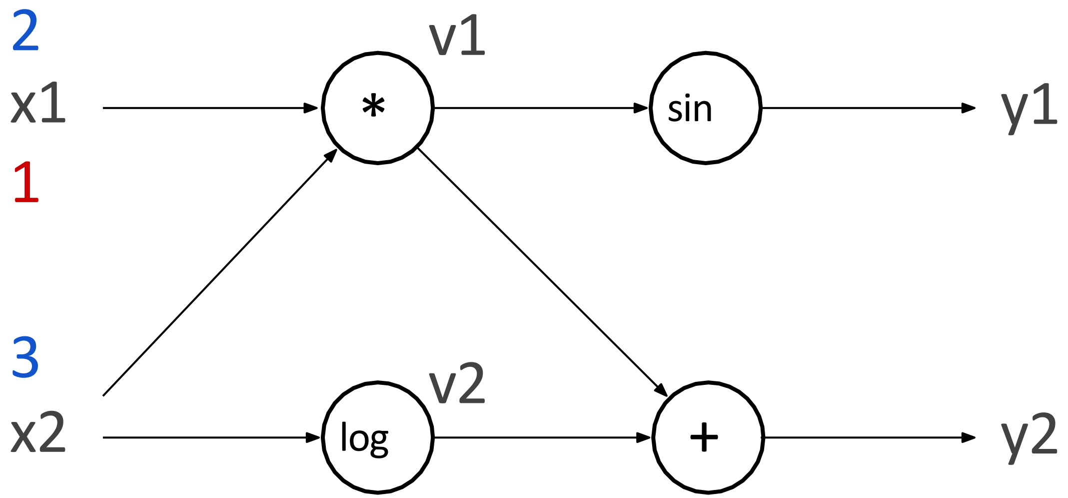
Forward mode

Primals
Derivatives (tangents)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



$$\frac{\partial x_1}{\partial x_1} = 1$$

Forward mode

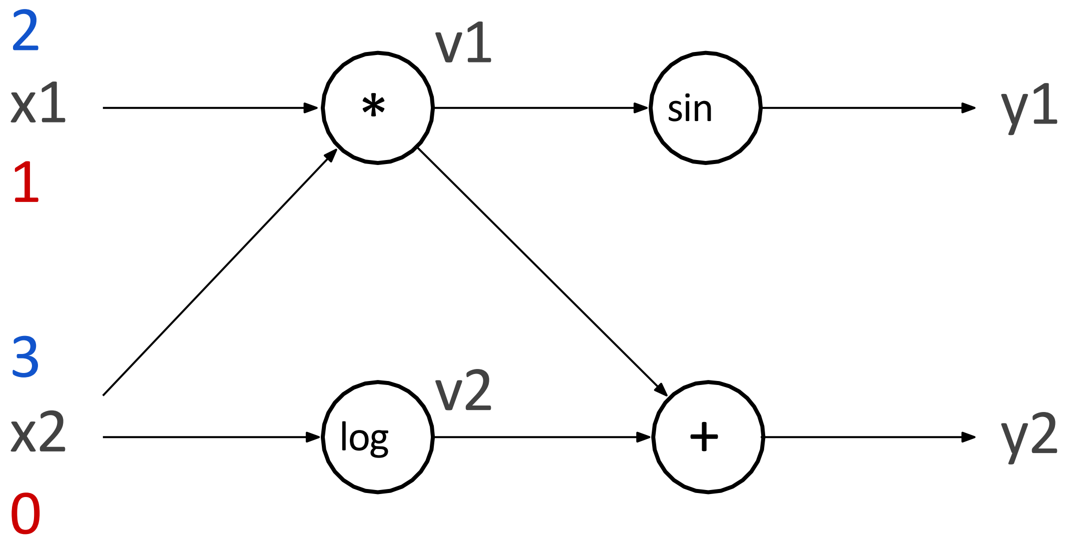
Primals

Derivatives (tangents)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



$$\frac{\partial x_2}{\partial x_1} = 0$$

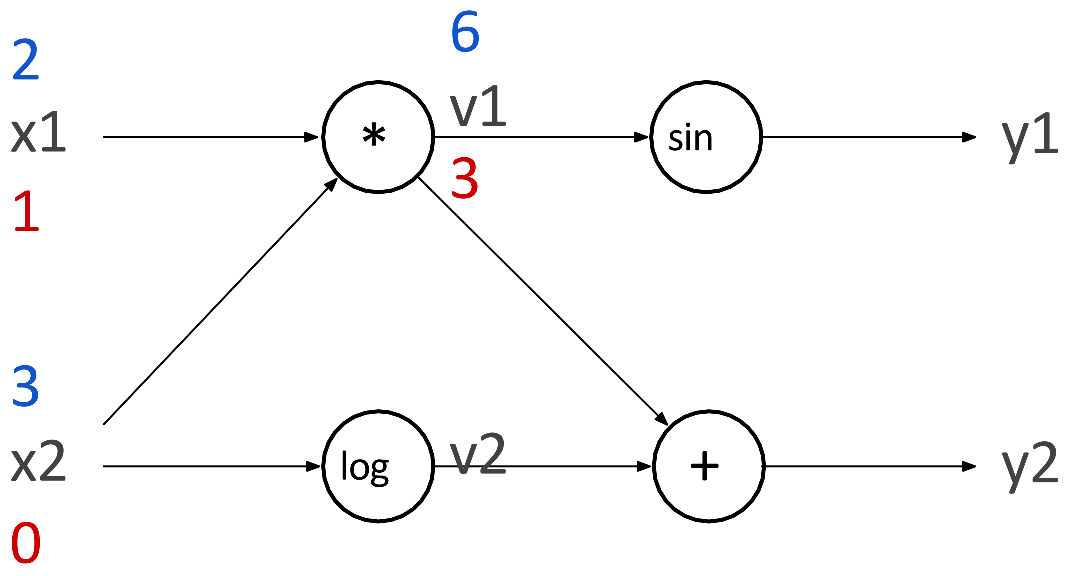
Forward mode

Primals
Derivatives (tangents)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



$$\frac{\partial v_1}{\partial x_1} = \frac{\partial x_1}{\partial x_1} x_2 + x_1 \frac{\partial x_2}{\partial x_1} = x_2$$

Forward mode

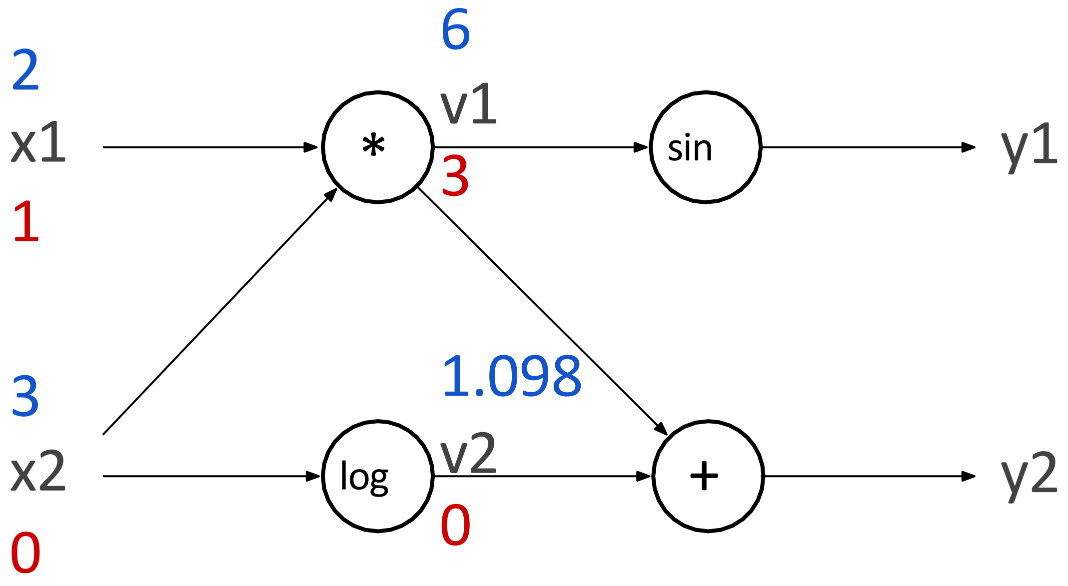
Primals
Derivatives (tangents)

$f : \mathbb{R}^2 \to \mathbb{R}^2$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2

    return (y1, y2)
```

f(2, 3)



$$\frac{\partial v_2}{\partial x_1} = \frac{1}{x_2}\frac{\partial x_2}{\partial x_1} = 0$$

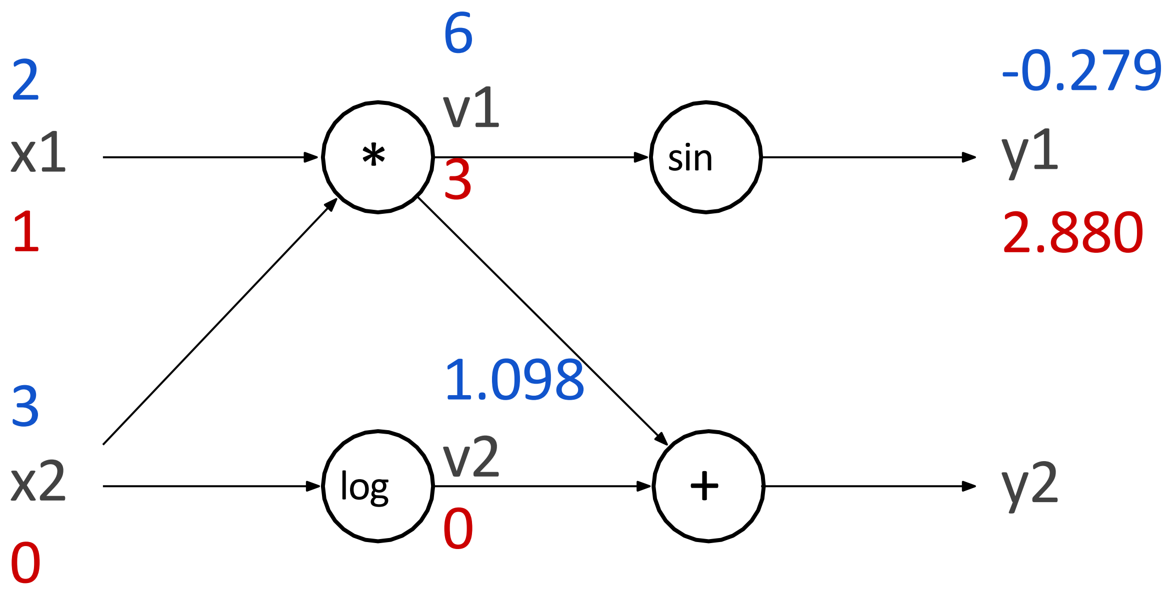
Forward mode

Primals
Derivatives (tangents)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



$$\frac{\partial y_1}{\partial x_1} = \cos(v_1)\frac{\partial v_1}{\partial x_1}$$

Forward mode

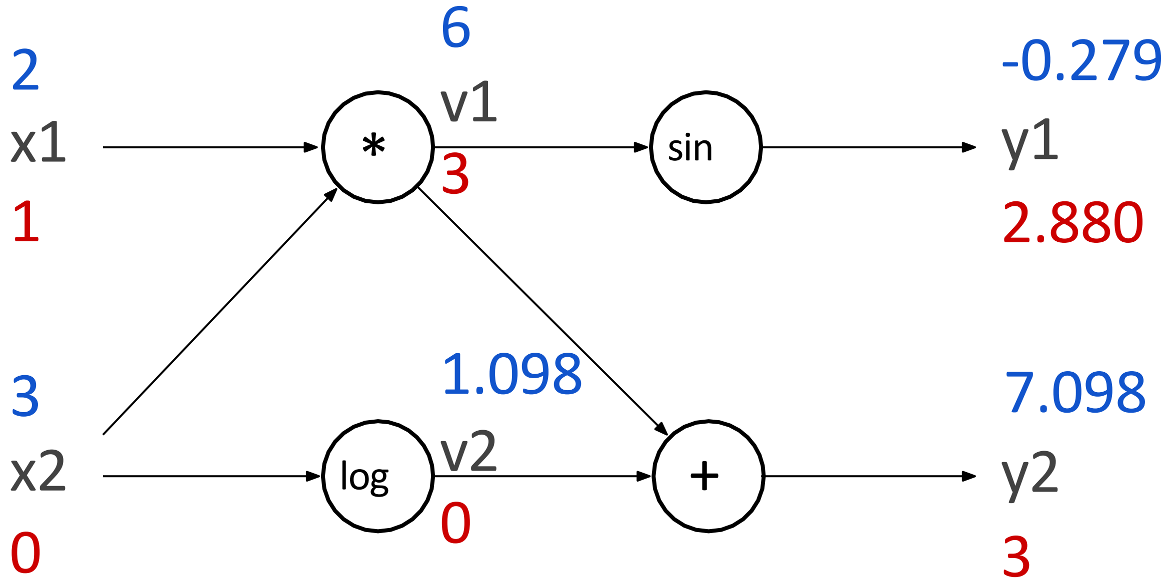
Primals

Derivatives (tangents)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



$$\frac{\partial y_2}{\partial x_1} = \frac{\partial v_1}{\partial x_1} + \frac{\partial v_2}{\partial x_1}$$

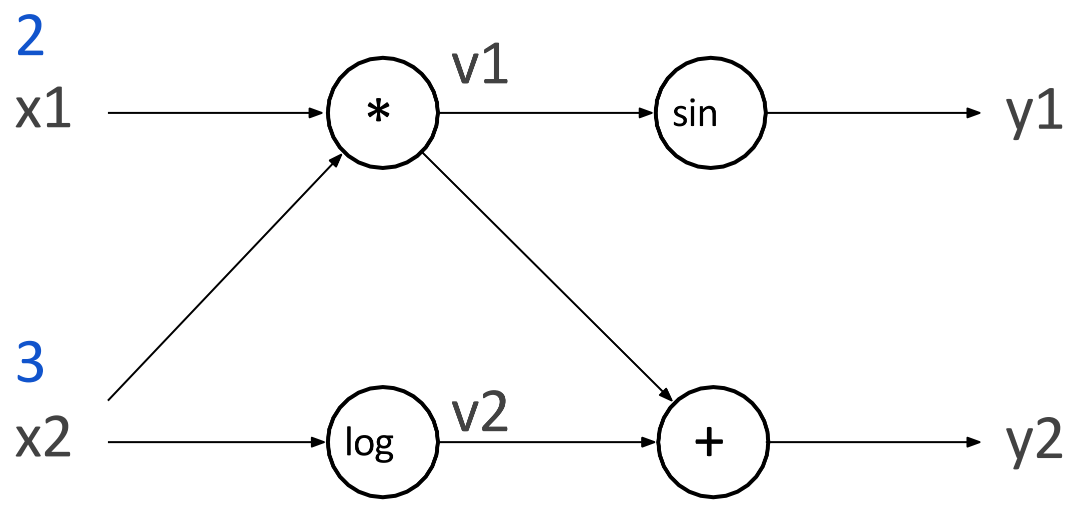
Reverse mode

Primals
Derivatives (adjoints)

$f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2

    return (y1, y2)
```



f(2, 3)

Reverse mode

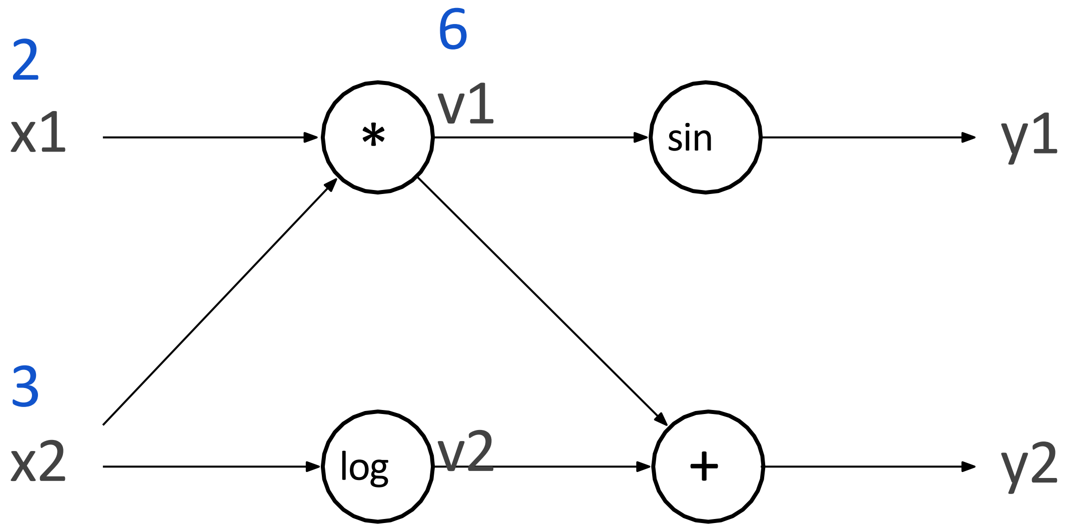
Primals

Derivatives (adjoints)

$f : \mathbb{R}^2 \to \mathbb{R}^2$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



Reverse mode

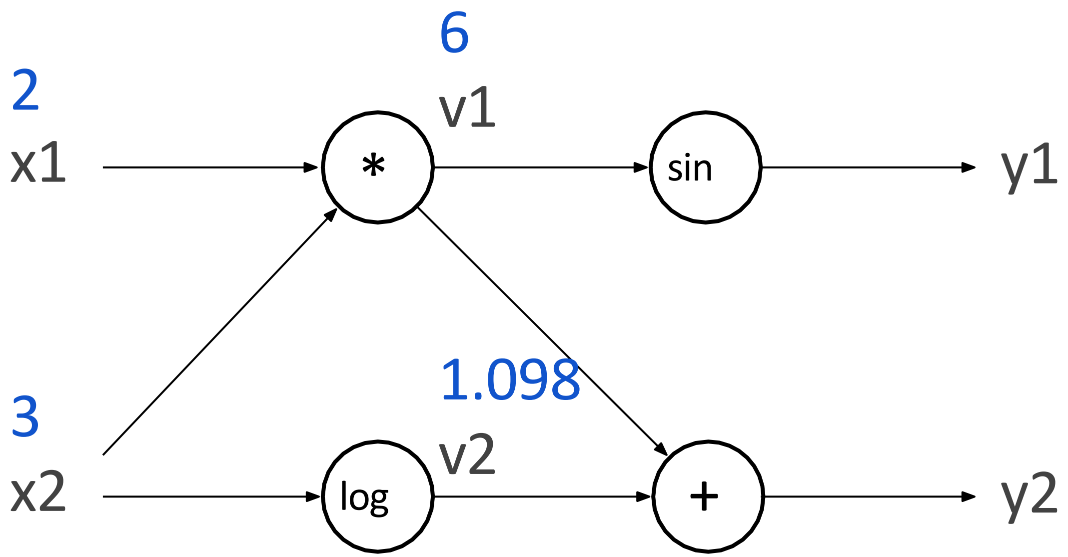
Primals

Derivatives (adjoints)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



Reverse mode

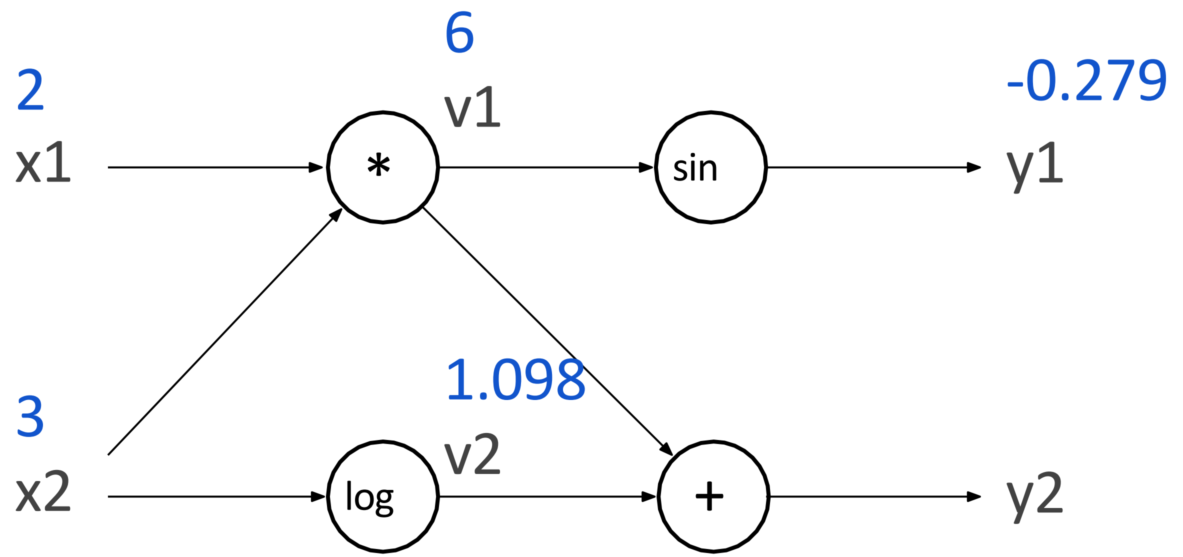
Primals

Derivatives (adjoints)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



Reverse mode

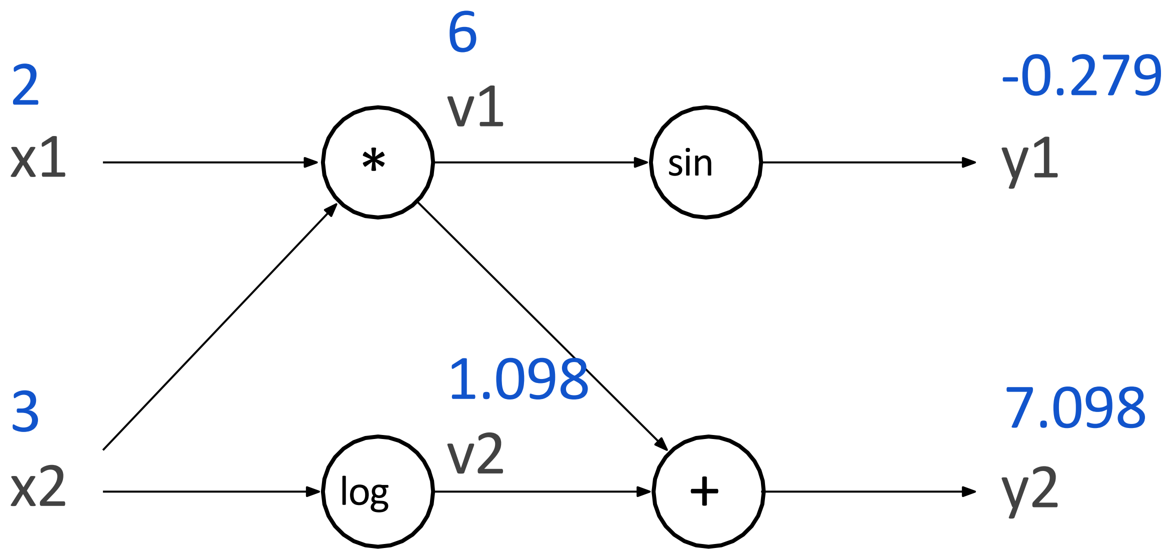
Primals

Derivatives (adjoints)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)

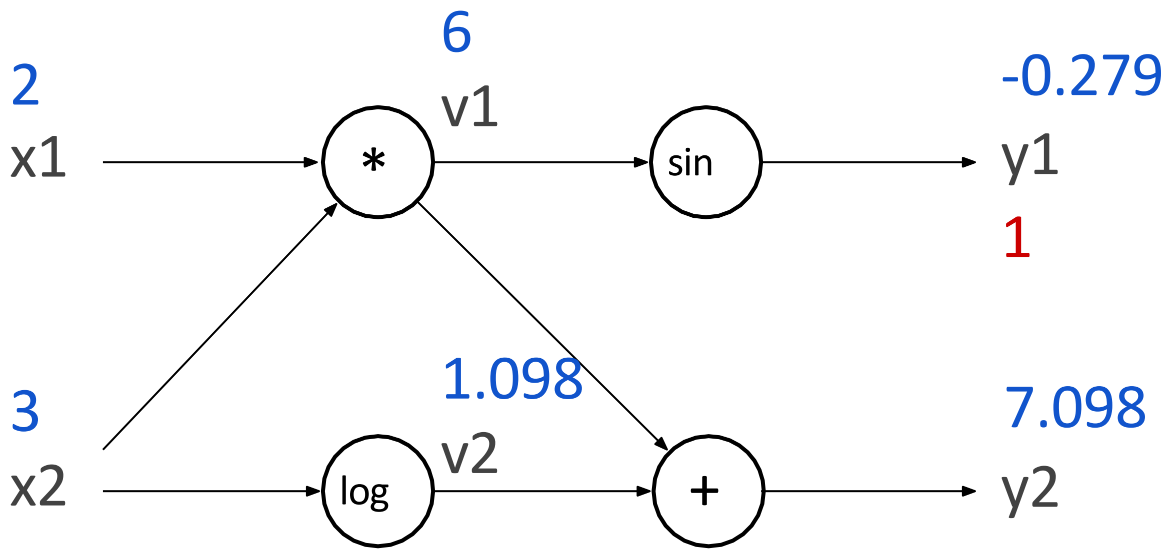


Reverse mode

$f : \mathbb{R}^2 \to \mathbb{R}^2$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



$$\frac{\partial y_1}{\partial y_1} = 1$$

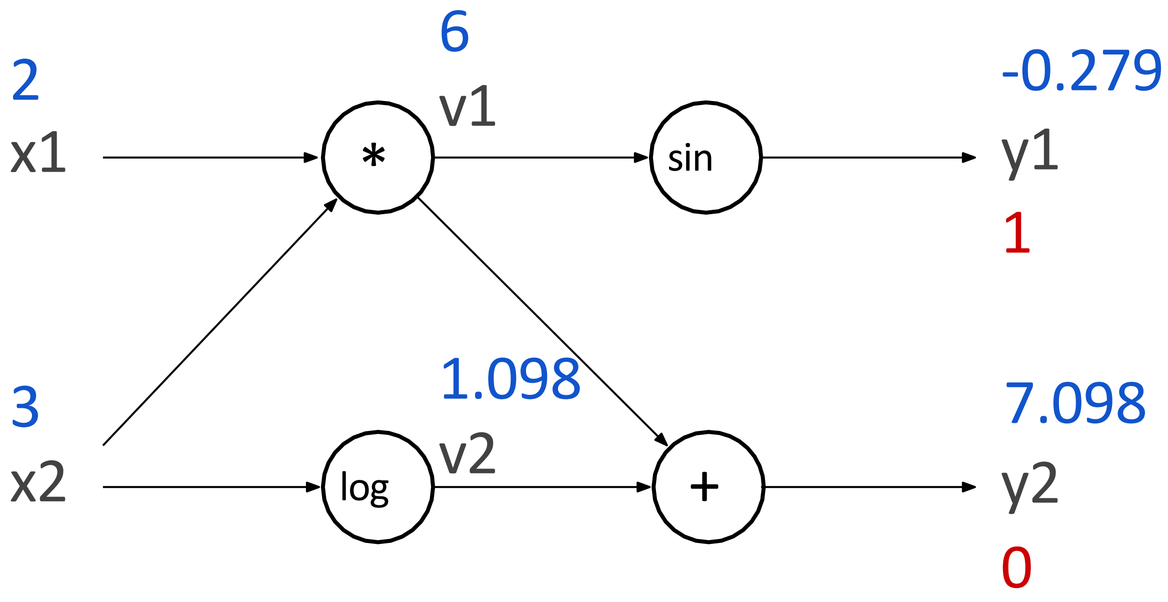
Primals
Derivatives (adjoints)

Reverse mode

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



$$\frac{\partial y_1}{\partial y_2} = 0$$

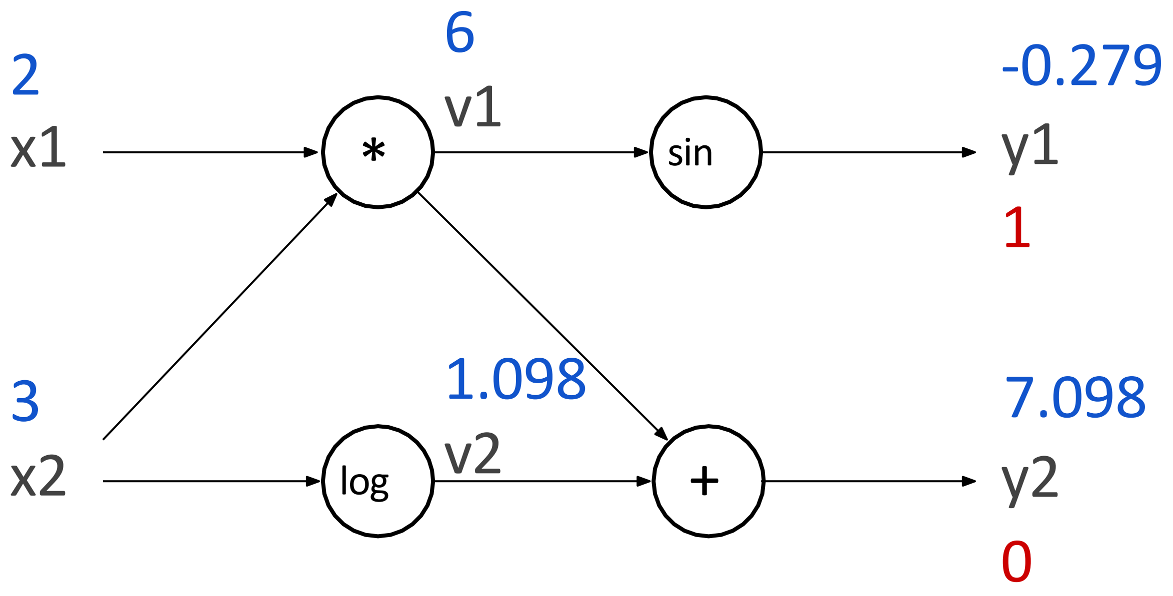
Reverse mode

Primals
Derivatives (adjoints)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



$$\frac{\partial y_1}{\partial v_1} =$$

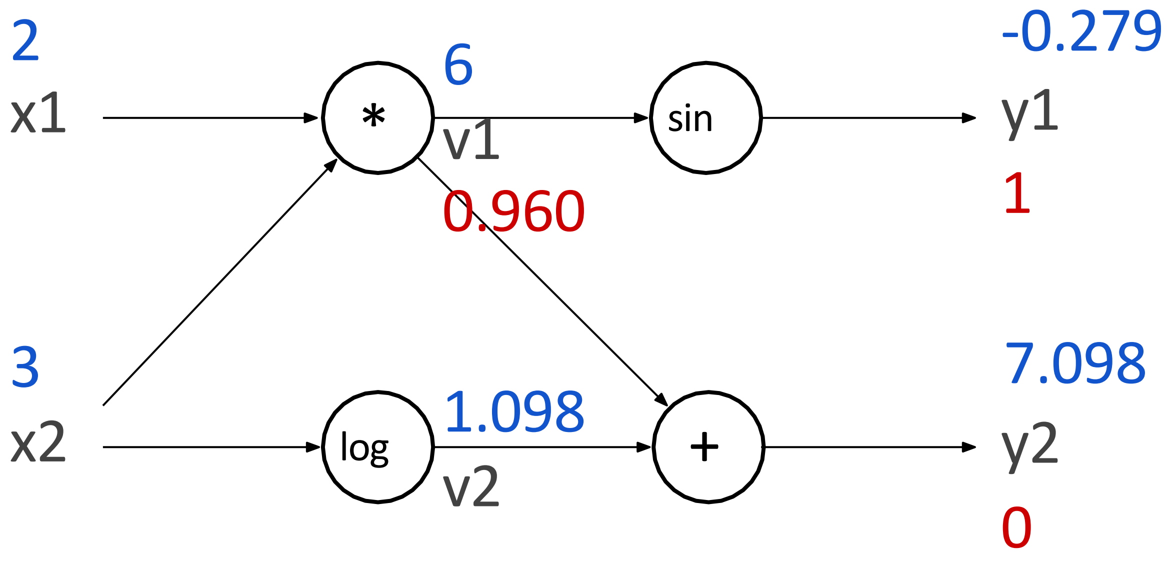
Reverse mode

Primals
Derivatives (adjoints)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



$$\frac{\partial y_1}{\partial v_1} = \cos(v1)\frac{\partial y_1}{\partial y_1}$$

Reverse mode

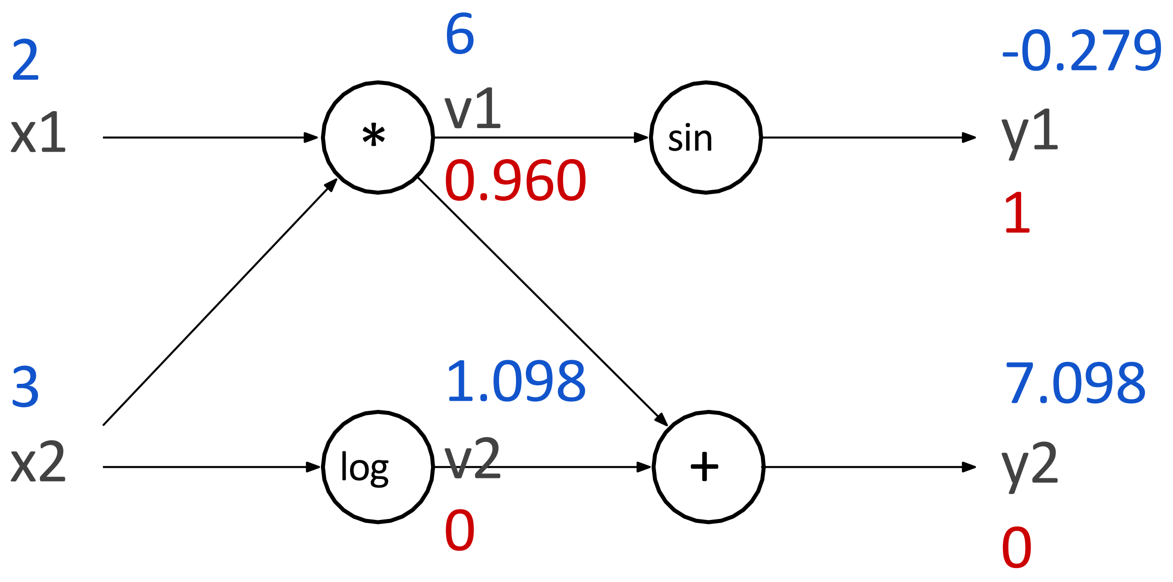
Primals

Derivatives (adjoints)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



$$\frac{\partial y_1}{\partial v_2} = 0$$

Reverse mode

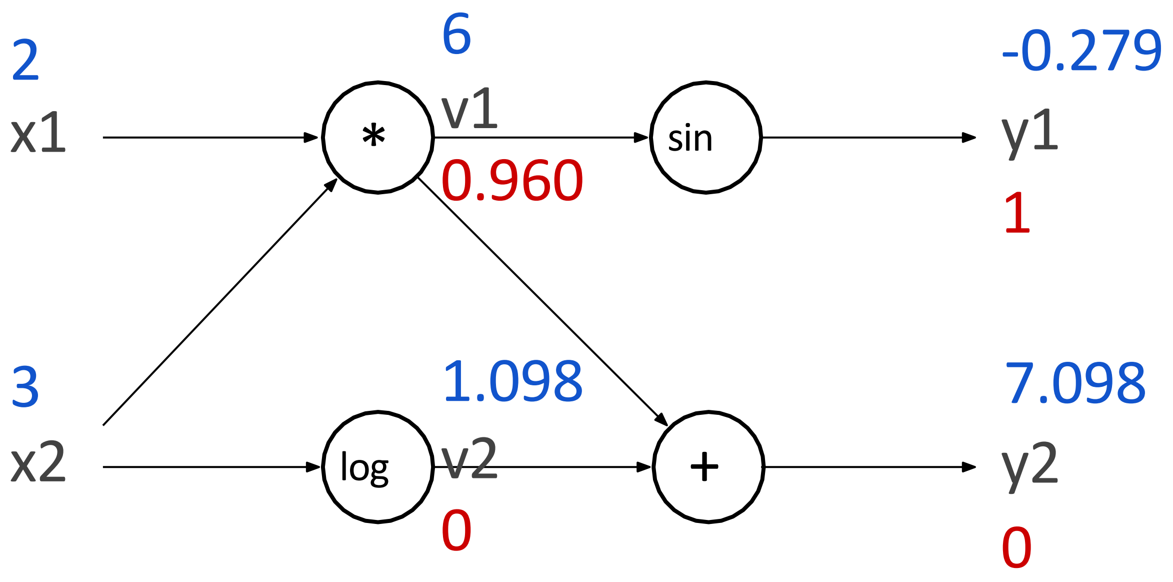
Primals

Derivatives (adjoints)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



$$\frac{\partial y_1}{\partial x_1} =$$

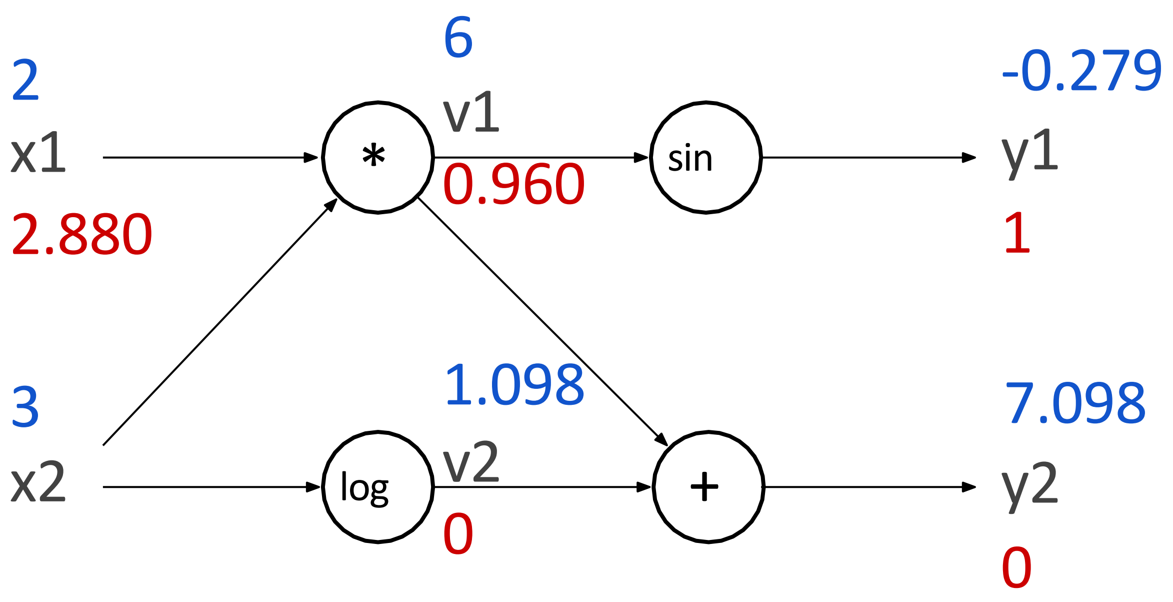
Reverse mode

Primals
Derivatives (adjoints)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



$$\frac{\partial y_1}{\partial x_1} = \frac{\partial v_1}{\partial x_1}\frac{\partial y_1}{\partial v_1} = x_2\frac{\partial y_1}{\partial v_1}$$

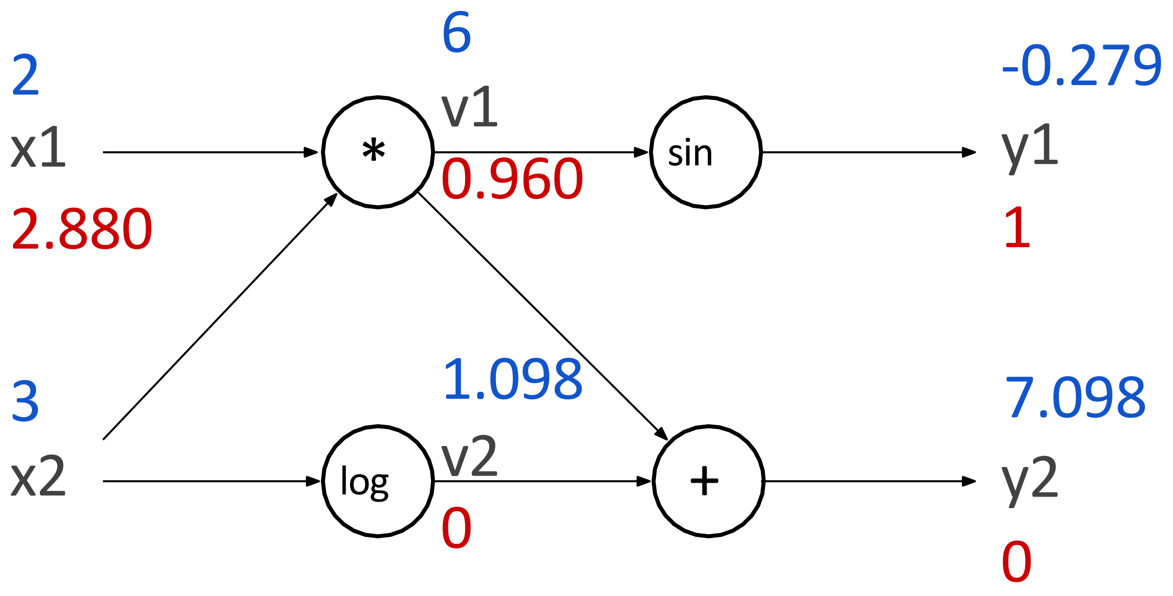
Reverse mode

Primals
Derivatives (adjoints)

$f : \mathbb{R}^2 \to \mathbb{R}^2$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



$$\frac{\partial y_1}{\partial x_2} =$$

Reverse mode

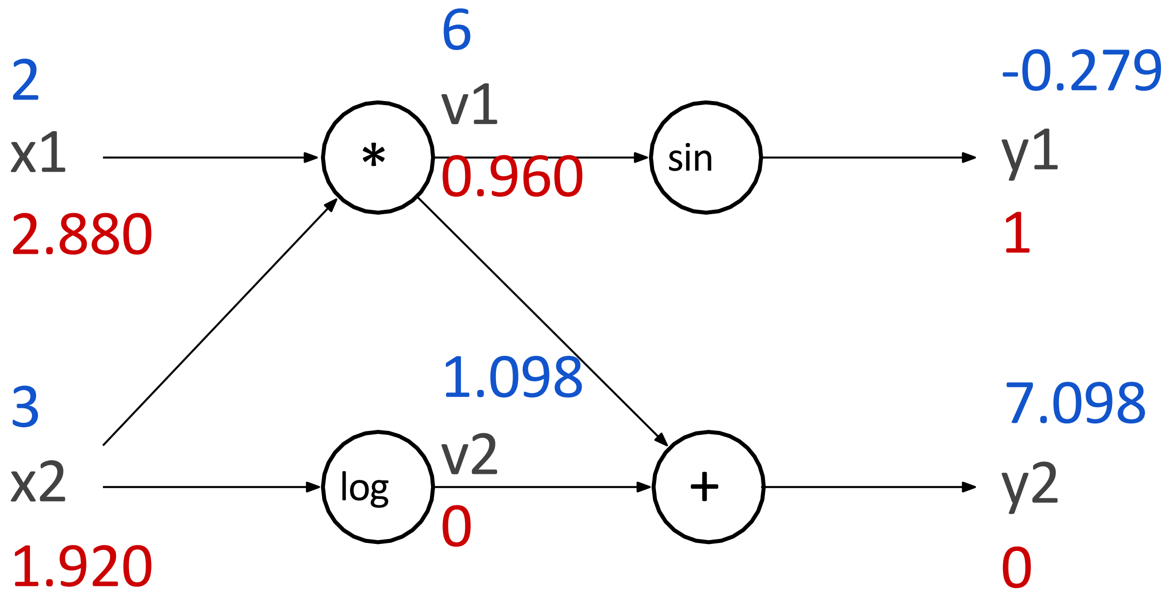
Primals

Derivatives (adjoints)

$$f : \mathbb{R}^2 \to \mathbb{R}^2$$

```
f(x1, x2):
    v1  =  x1 * x2
    v2  =  log(x2)
    y1  =  sin(v1)
    y2  =  v1 + v2
    return (y1, y2)
```

f(2, 3)



$$\frac{\partial y_1}{\partial x_2} = \frac{\partial v_1}{\partial x_2}\frac{\partial y_1}{\partial v_1} + \frac{\partial v_2}{\partial x_2}\frac{\partial y_1}{\partial v_2} = x_1 \frac{\partial y_1}{\partial v_1}$$

Forward vs reverse summary

In the extreme $\mathbf{f} : \mathbb{R} \to \mathbb{R}^m$
use forward mode to evaluate

$$(\frac{\partial f_1}{\partial x}, \cdots, \frac{\partial f_m}{\partial x})$$

In the extreme $f : \mathbb{R}^n \to \mathbb{R}$
use reverse mode to evaluate

$$\nabla f(\mathbf{x}) = (\frac{\partial f}{\partial x_1}, \cdots, \frac{\partial f}{\partial x_n})$$

In general $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ the Jacobian $\mathbf{J}_f(\mathbf{x}) \in \mathbb{R}^{m \times n}$ can be evaluated in

- $O(n\,\mathrm{time}(\mathbf{f}))$ with forward mode
- $O(m\,\mathrm{time}(\mathbf{f}))$ with reverse mode

Reverse performs better when $n \gg m$

Autograd

```python
import autograd.numpy as np
from autograd import grad
```

very sneaky!

```python
def sigmoid(x):
    return 0.5*(np.tanh(x) + 1)

def logistic_predictions(weights, inputs):
    # Outputs probability of a label being true according to logistic model.
    return sigmoid(np.dot(inputs, weights))

def training_loss(weights):
    # Training loss is the negative log-likelihood of the training labels.
    preds = logistic_predictions(weights, inputs)
    label_probabilities = preds * targets + (1 - preds) * (1 - targets)
    return -np.sum(np.log(label_probabilities))
```

... (load the data) ...

```python
# Define a function that returns gradients of training loss using Autograd.
training_gradient_fun = grad(training_loss)
```

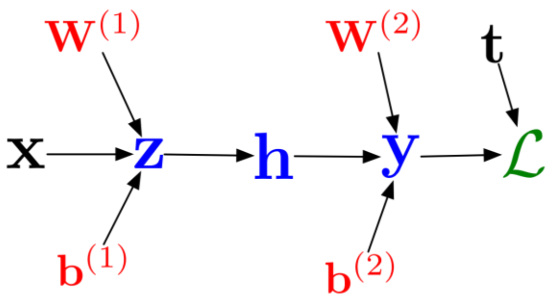
Autograd constructs a function for computing derivatives

```python
# Optimize weights using gradient descent.
weights = np.array([0.0, 0.0, 0.0])
print "Initial loss:", training_loss(weights)
for i in xrange(100):
    weights -= training_gradient_fun(weights) * 0.01

print  "Trained loss:", training_loss(weights)
```

- The rest of this lecture covers how Autograd is implemented.
- Source code for the original Autograd package:
  https://github.com/HIPS/autograd
- Autodidact, a pedagogical implementation of Autograd — you are encouraged to read the code.
  https://github.com/mattjj/autodidact

Building the Computation Graph

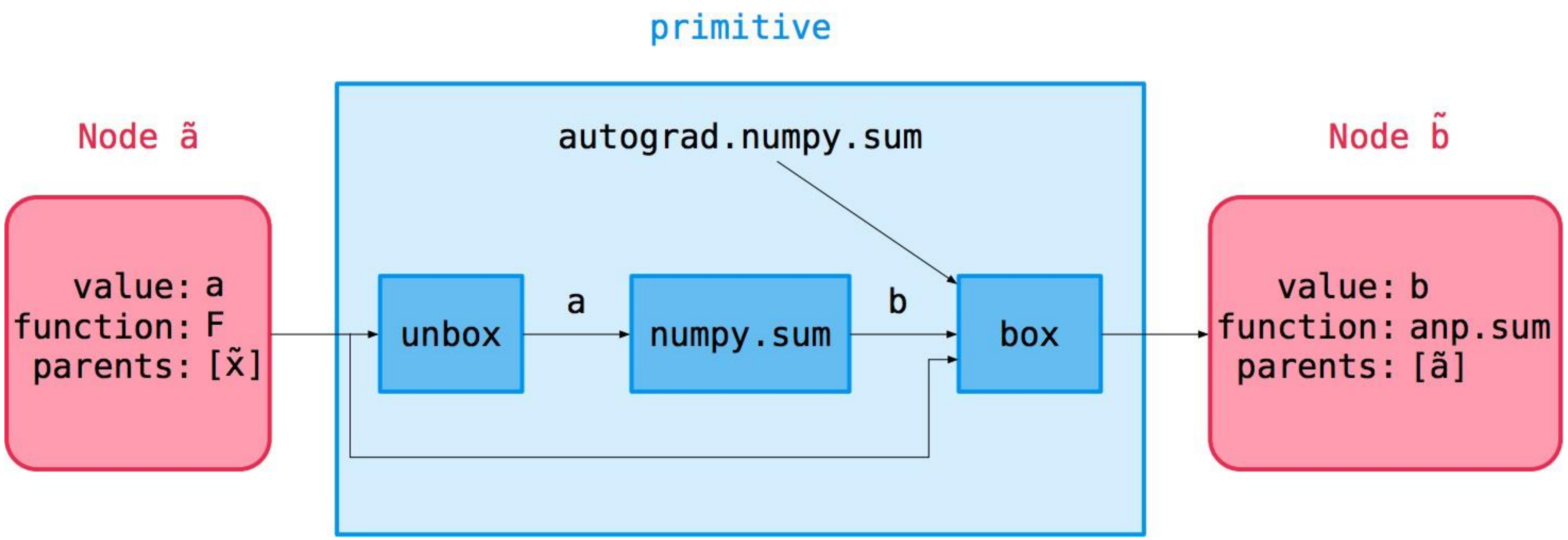


- Most autodiff systems, including Autograd, explicitly construct the computation graph.
  - Some frameworks like TensorFlow provide mini-languages for building computation graphs directly. Disadvantage: need to learn a totally new API.
  - Autograd instead builds them by tracing the forward pass computation, allowing for an interface nearly indistinguishable from NumPy.

- The Node class (defined in `tracer.py`) represents a node of the computation graph. It has attributes:
  - `value`, the actual value computed on a particular set of inputs
  - fun, the primitive operation defining the node
  - `args` and kwargs, the arguments the op was called with
  - `parents`, the parent Nodes

Building the Computation Graph

- Autograd's fake NumPy module provides primitive ops which look and feel like NumPy functions, but secretly build the computation graph.
- They wrap around NumPy functions:



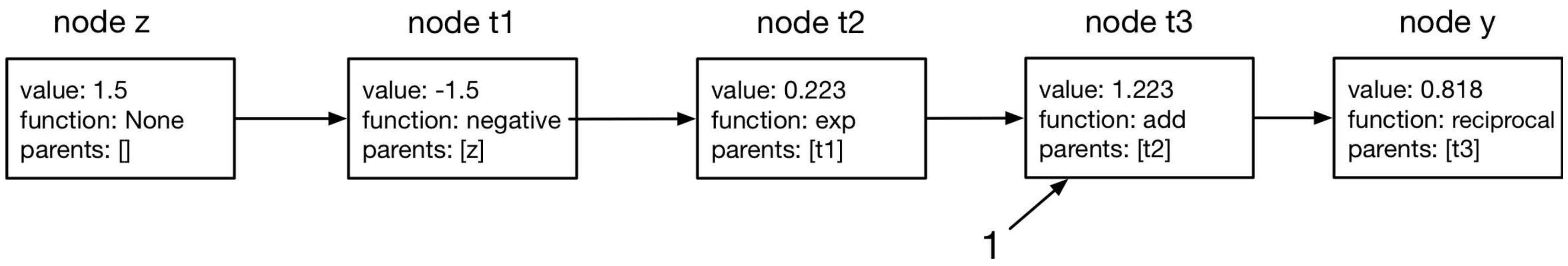
Building the Computation Graph

Example:

```python
def logistic(z):
    return 1. / (1. + np.exp(-z))

# that is equivalent to:
def logistic2(z):
    return np.reciprocal(np.add(1, np.exp(np.negative(z))))

z = 1.5
y = logistic(z)
```

Vector-Jacobian Products

- Implement the primitive operations in vectorized form.

- The Jacobian is the matrix of partial derivatives:

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

- The backprop equation (single child node) can be written as a vector-Jacobian product (VJP):

$$\overline{x_j} = \sum_i \overline{y_i} \frac{\partial y_i}{\partial x_j} \qquad \overline{\mathbf{x}} = \overline{\mathbf{y}}^T \mathbf{J}$$

- That gives a row vector 1 by n. We can treat it as a column vector by taking

$$\overline{\mathbf{x}} = \mathbf{J}^T \overline{\mathbf{y}}$$

Note: usually don't explicitly construct the Jacobian. It's simpler and more efficient to compute the VJP directly.

Vector-Jacobian Products

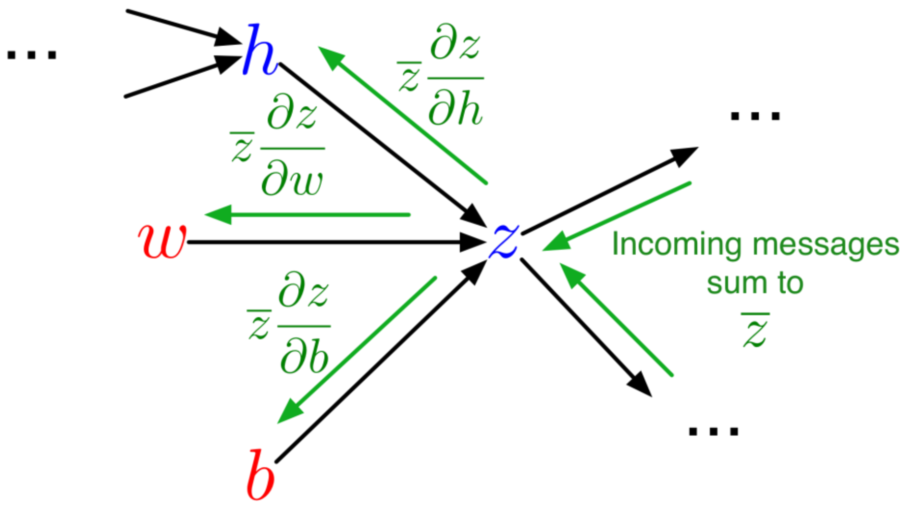
- For each primitive operation, we must specify VJPs for *each* of its arguments. Consider $y = \exp(x)$.
- This is a function which takes in the output gradient (i.e. $\bar{y}$), the answer (y), and the arguments (x), and returns the input gradient ($\bar{x}$)
- `defvjp` (defined in `core.py`) is a convenience routine for registering VJPs. It just adds them to a dict.
- Examples from numpy/numpy `vjps.py`

```
defvjp(negative, lambda g, ans, x: -g)
defvjp(exp,      lambda g, ans, x: ans * g)
defvjp(log,      lambda g, ans, x: g / x)

defvjp(add,         lambda g, ans, x, y : g,
                    lambda g, ans, x, y : g)
defvjp(multiply,    lambda g, ans, x, y : y * g,
                    lambda g, ans, x, y : x * g)
defvjp(subtract,    lambda g, ans, x, y : g,
                    lambda g, ans, x, y : -g)
```

Backward Pass

- Backprop computations are more modular if we view them as message passing.



The backwards pass is defined in `core.py`.

```
def backward_pass(g, end_node):
    outgrads = {end_node: g}
    for node in toposort(end_node):
        outgrad = outgrads.pop(node)
        fun, value, args, kwargs, argnums = node.recipe
        for argnum, parent in zip(argnums, node.parents):
            vjp = primitive_vjps[fun][argnum]
            parent_grad = vjp(outgrad, value, *args, **kwargs)
            outgrads[parent] = add_outgrads(outgrads.get(parent), parent_grad)
    return outgrad

def add_outgrads(prev_g, g):
    if prev_g is None:
        return g
    return prev_g + g
```

Summary

- We saw three main parts to the code:
  - tracing the forward pass to build the computation graph
  - vector-Jacobian products for primitive ops
  - the backwards pass
- Building the computation graph requires fancy NumPy gymnastics, but other two items are basically what I showed you.
- You're encouraged to read the full code (< 200 lines!) at:

  https://github.com/mattjj/autodidact/tree/master/autograd