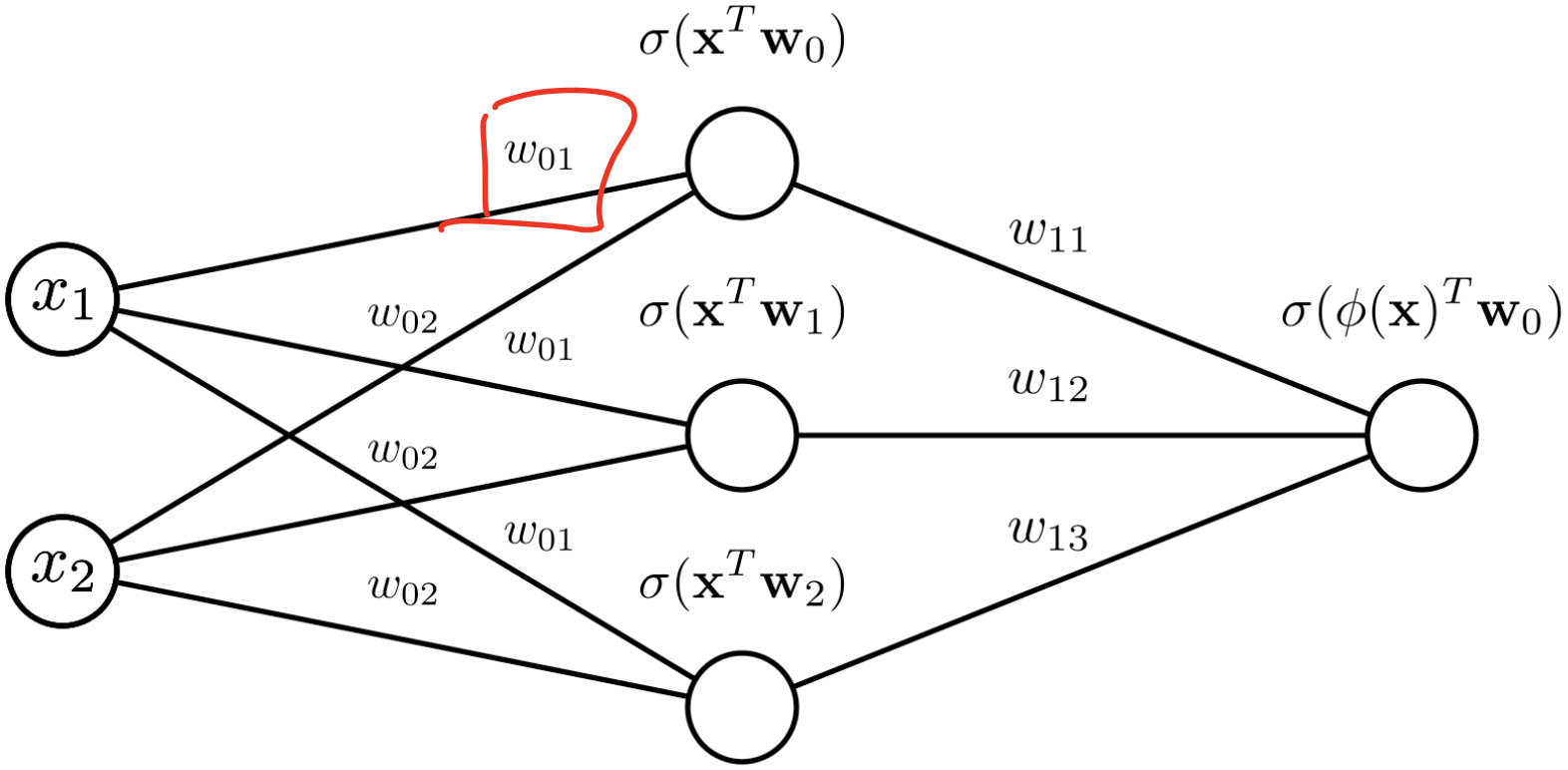
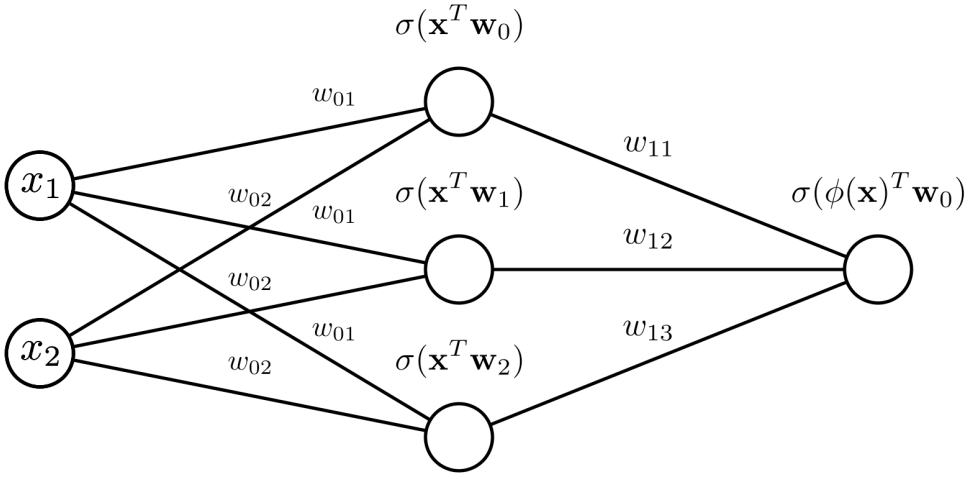
Learning Neural Networks

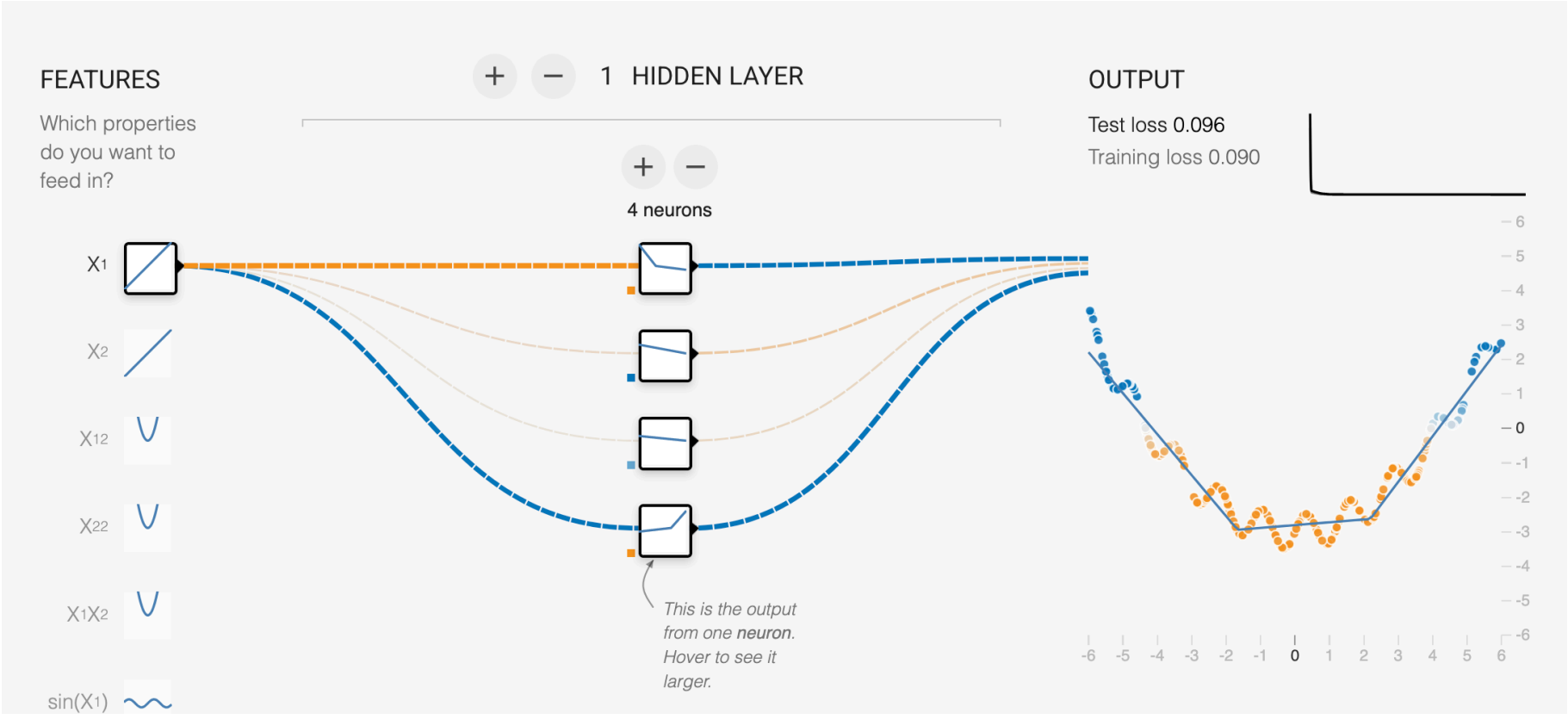
Last time: A Neural Network!



Last time: A Single Neuron



Last time: A Neural Network!

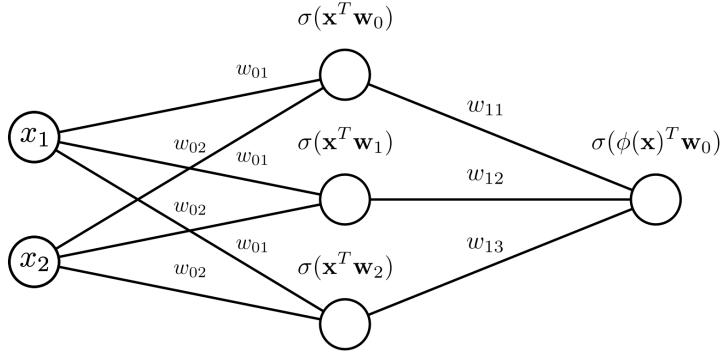


Last time: Neural networks with matrix notation

Our neural network

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \mathbf{w}_0 = \begin{bmatrix} w_{01} \\ w_{02} \end{bmatrix}$$

$$f(\mathbf{x}) = \phi(\mathbf{x})^T \mathbf{w}_0, \quad \phi(\mathbf{x}) = \begin{bmatrix} \sigma(\mathbf{x}^T \mathbf{w}_1) \\ \sigma(\mathbf{x}^T \mathbf{w}_2) \\ \sigma(\mathbf{x}^T \mathbf{w}_3) \end{bmatrix} = \begin{bmatrix} \sigma(x_1 w_{11} + x_2 w_{12}) \\ \sigma(x_1 w_{21} + x_2 w_{22}) \\ \sigma(x_1 w_{31} + x_2 w_{32}) \end{bmatrix}$$



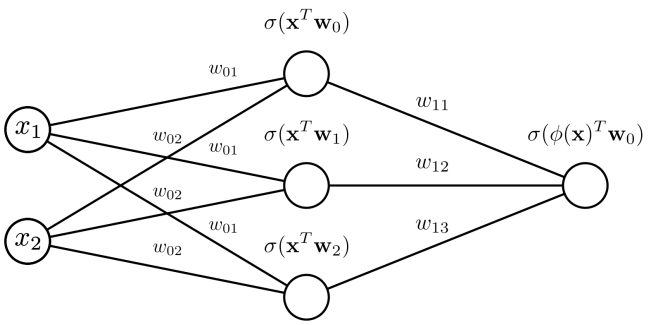
Write the linear function for each neuron as a matrix vector product

$$\begin{bmatrix} \mathbf{x}^T \mathbf{w}_1 \\ \mathbf{x}^T \mathbf{w}_2 \\ \mathbf{x}^T \mathbf{w}_3 \end{bmatrix} = \begin{bmatrix} x_1 w_{11} + x_2 w_{12} \\ x_1 w_{21} + x_2 w_{22} \\ x_1 w_{31} + x_2 w_{32} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \qquad \begin{bmatrix} \mathbf{x}^T \mathbf{w}_1 \\ \mathbf{x}^T \mathbf{w}_2 \\ \mathbf{x}^T \mathbf{w}_3 \end{bmatrix} = \mathbf{W}\mathbf{x} = (\mathbf{x}^T \mathbf{W}^T)^T$$

Last time: Neural networks with matrix notation

Write all the weights for a layer in a big matrix



$$\mathbf{W}_1 = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \mathbf{w}_3^T \\ \vdots \end{bmatrix}$$

$$f(\mathbf{x}) = \sigma(\mathbf{W}_1\mathbf{x})^T\mathbf{w_0}$$

Compact prediction function!

Can do the same for the full dataset

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \mathbf{x}_3^T \\ \vdots \end{bmatrix}$$

$$f(\mathbf{x}) = \sigma(\mathbf{X}\mathbf{W}_1)\mathbf{w_0}$$

Neural networks with matrix notation

$$f(\mathbf{x}) = \sigma(\mathbf{X}\mathbf{W}^T)\mathbf{w}_0$$

To summarize:

- $\mathbf{X}$: $N \times d$ matrix of observations
- $\mathbf{W}$: $h \times d$ matrix of network weights
- $\mathbf{w}_0$: $h\ (\times 1)$ vector of linear regression weights

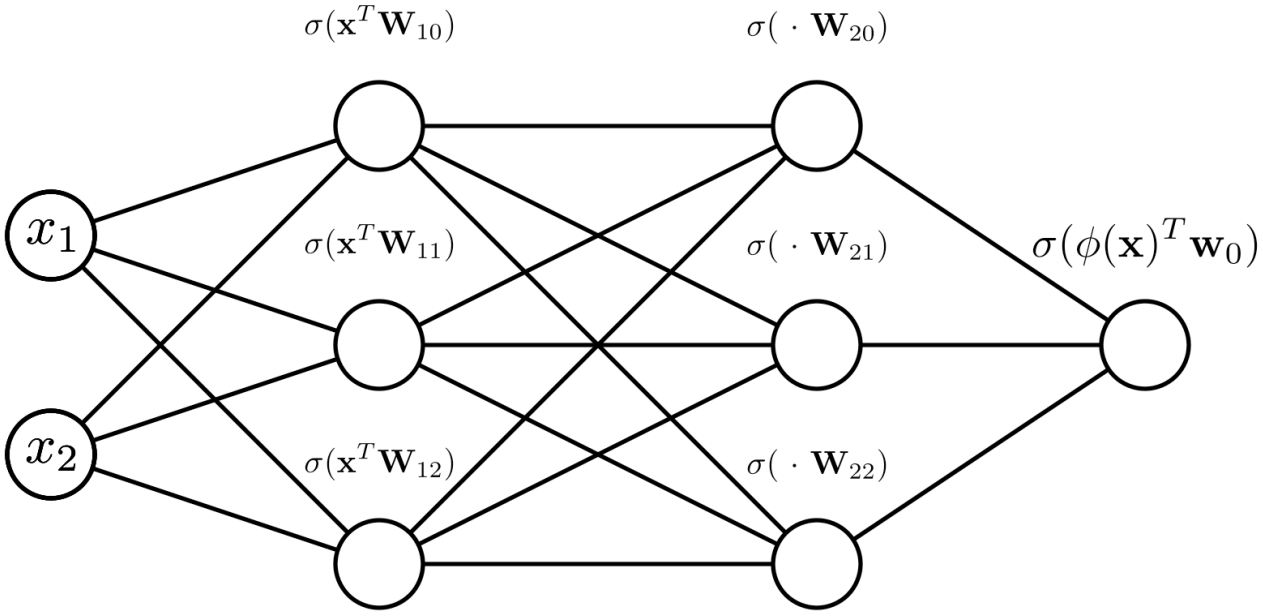
$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \mathbf{x}_3^T \\ \vdots \end{bmatrix} \quad \mathbf{W}_1 = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \mathbf{w}_3^T \\ \vdots \end{bmatrix} \quad \mathbf{w}_0 = \begin{bmatrix} w_{01} \\ w_{02} \end{bmatrix}$$

If we check that our dimensions work for matrix multiplication we see that we get the $N \times 1$ vector of predictions we are looking for!

$$(N \times d)(h \times d)^T(h \times 1) \to (N \times d)(d \times h)(h \times 1) \to (N \times h)(h \times 1)$$

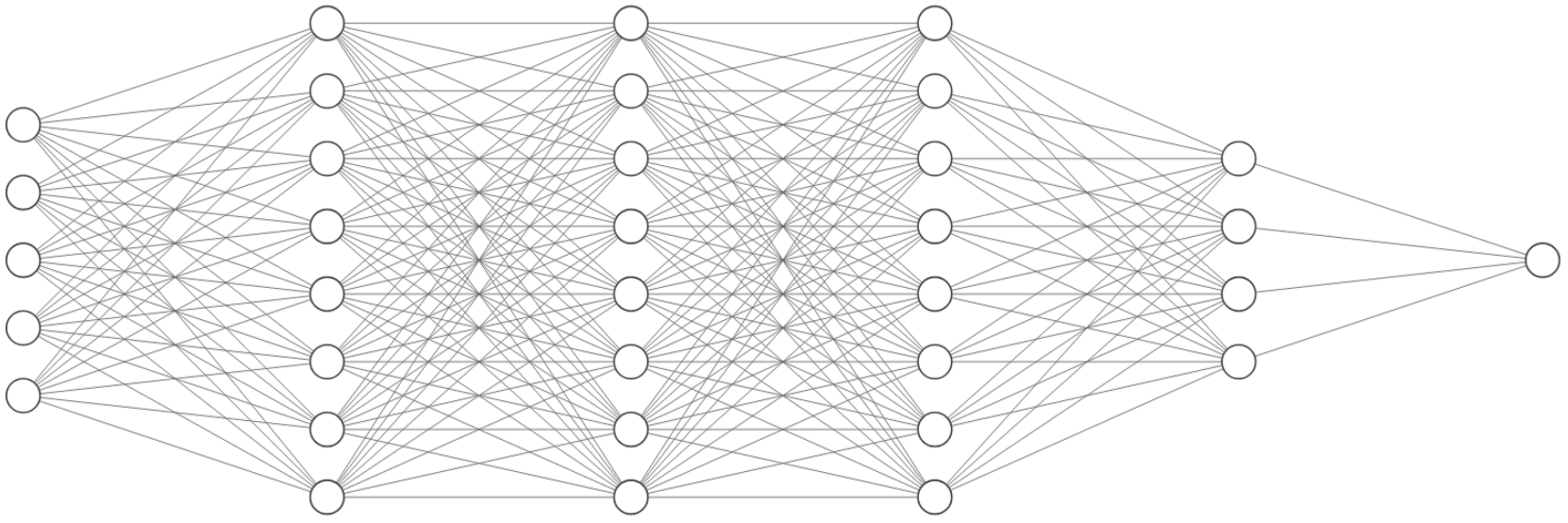
$$\longrightarrow (N \times 1)$$

Applying this recursively

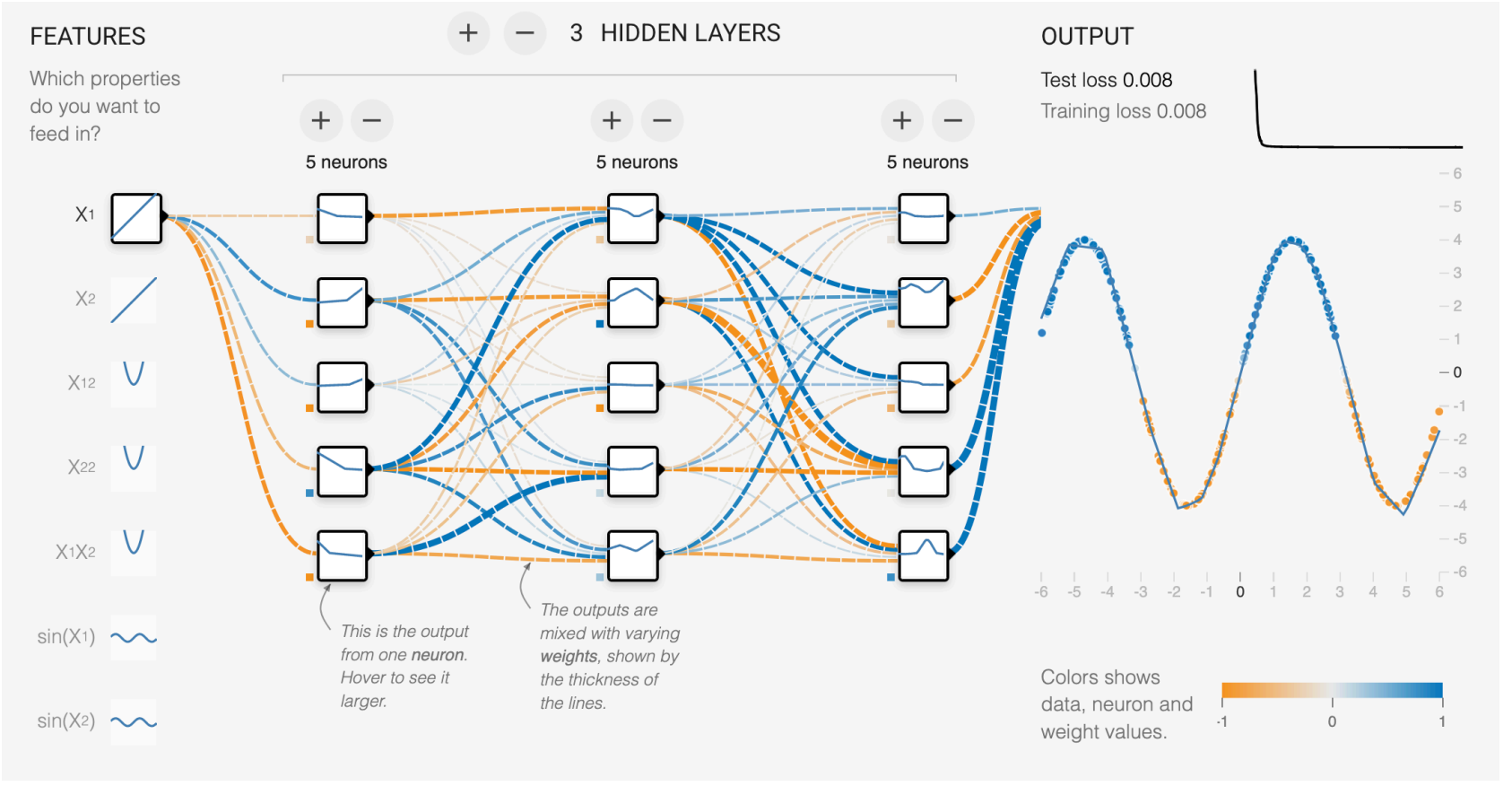


$$\phi(\mathbf{x})_i = \sigma(\mathbf{x}^T\mathbf{w}_i) \qquad \phi(\mathbf{x})_i = \sigma(\sigma(\mathbf{x}^T\mathbf{W}^T)\mathbf{w}_i)$$

Applying this recursively



Applying this recursively



Adding Biases

Loss function

$$\mathbf{NLL}(\mathbf{w}_0, \mathbf{W}, \mathbf{X}, \mathbf{y}) = -\sum_{i=1}^{N} \Big[y_i \log p(y = 1 \mid \mathbf{x}, \mathbf{w}_0, \mathbf{W}) + (1 - y_i) \log p(y = 0 \mid \mathbf{x}, \mathbf{w}_0, \mathbf{W}) \Big]$$

$$= -\sum_{i=1}^{N} \log \sigma\big((2y_i - 1)\phi(\mathbf{x}_i)^T \mathbf{w}\big)$$

Loss function

$$\mathbf{NLL}(\mathbf{w}_0, \mathbf{W}, \mathbf{X}, \mathbf{y}) = -\sum_{i=1}^{N} \Big[y_i \log p(y = 1 \mid \mathbf{x}, \mathbf{w}_0, \mathbf{W}) + (1 - y_i) \log p(y = 0 \mid \mathbf{x}, \mathbf{w}_0, \mathbf{W}) \Big]$$

$$= -\sum_{i=1}^{N} \log \sigma\big((2y_i - 1)\phi(\mathbf{x}_i)^T \mathbf{w}\big)$$

Substituting in our prediction function

$$p(y = 1 \mid \mathbf{x}, \mathbf{w}_0, \mathbf{W}) = \sigma(\phi(\mathbf{x})^T \mathbf{w}_0) = \sigma(\sigma(\mathbf{x}^T \mathbf{W}^T)\mathbf{w}_0), \quad \phi(\mathbf{x}) = \begin{bmatrix} \sigma(\mathbf{x}^T \mathbf{W}_1) \\ \sigma(\mathbf{x}^T \mathbf{W}_2) \\ \sigma(\mathbf{x}^T \mathbf{W}_3) \end{bmatrix}$$

$$= \sigma\big(w_{01} \cdot \sigma(x_1 W_{11} + x_2 W_{12}) + w_{02} \cdot \sigma(x_1 W_{21} + x_2 W_{22}) + w_{03} \cdot \sigma(x_1 W_{31} + x_2 W_{32})\big)$$

Gradient descent

Loss

$$\mathbf{NLL}(\mathbf{w}_0, \mathbf{W}, \mathbf{X}, \mathbf{y}) = -\sum_{i=1}^{N} \Big[y_i \log p(y = 1 \mid \mathbf{x}, \mathbf{w}_0, \mathbf{W}) + (1 - y_i) \log p(y = 0 \mid \mathbf{x}, \mathbf{w}_0, \mathbf{W}) \Big]$$

Updates

$$\mathbf{w}_0^{(k+1)} \longleftarrow \mathbf{w}_0^{(k)} - \alpha \nabla_{\mathbf{w}_0} \mathbf{NLL}(\mathbf{w}_0, \mathbf{W}, \mathbf{X}, \mathbf{y}), \quad \mathbf{W}^{(k+1)} \longleftarrow \mathbf{W}^{(k)} - \alpha \nabla_{\mathbf{W}} \mathbf{NLL}(\mathbf{w}_0, \mathbf{W}, \mathbf{X}, \mathbf{y})$$

Need:

$$\nabla_{\mathbf{w}_0} \mathbf{NLL}(\mathbf{w}_0, \mathbf{W}, \mathbf{X}, \mathbf{y}) = \ ? \qquad \nabla_{\mathbf{W}} \mathbf{NLL}(\mathbf{w}_0, \mathbf{W}, \mathbf{X}, \mathbf{y}) = \ ?$$

Gradient Shapes

$$\mathbf{NLL}(\mathbf{w}_0, \mathbf{W}, \mathbf{X}, \mathbf{y}) = -\sum_{i=1}^{N} \Big[y_i \log p(y = 1 \mid \mathbf{x}, \mathbf{w}_0, \mathbf{W}) + (1 - y_i) \log p(y = 0 \mid \mathbf{x}, \mathbf{w}_0, \mathbf{W}) \Big]$$

$$\nabla_{\mathbf{w}_0} \mathbf{NLL}(\mathbf{w}_0, \mathbf{W}, \mathbf{X}, \mathbf{y}) = \begin{bmatrix} \frac{\partial \mathbf{NLL}}{\partial w_{01}} \\ \frac{\partial \mathbf{NLL}}{\partial w_{02}} \\ \frac{\partial \mathbf{NLL}}{\partial w_{03}} \\ \vdots \end{bmatrix}, \quad \nabla_{\mathbf{W}} \mathbf{NLL}(\mathbf{w}_0, \mathbf{W}, \mathbf{X}, \mathbf{y}) = \begin{bmatrix} \frac{\partial \mathbf{NLL}}{\partial W_{11}} & \frac{\partial \mathbf{NLL}}{\partial W_{12}} & \cdots & \frac{\partial \mathbf{NLL}}{\partial W_{1d}} \\ \frac{\partial \mathbf{NLL}}{\partial W_{21}} & \frac{\partial \mathbf{NLL}}{\partial W_{22}} & \cdots & \frac{\partial \mathbf{NLL}}{\partial W_{2d}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{NLL}}{\partial W_{h1}} & \frac{\partial \mathbf{NLL}}{\partial W_{h2}} & \cdots & \frac{\partial \mathbf{NLL}}{\partial W_{hd}} \end{bmatrix}$$

Automatic differentiation

Automatic differentiation review

$$L = -\log \sigma(wx^2)$$

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

$g = g(x)$ $\quad \frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$

$f = f(g)$

3 fundamental steps

1: Break up computation into individual operations

$$a = x^2$$
$$b = wa$$
$$c = \sigma(b)$$
$$g = \log c$$
$$L = -g$$

2: Compute derivative of each step

$$\frac{da}{dx} = 2x \qquad \frac{db}{da} = w$$

$$\frac{dc}{db} = \sigma(b)(1 - \sigma(b))$$

$$\frac{dg}{dc} = \frac{1}{c} \qquad \frac{dL}{dg} = -1$$

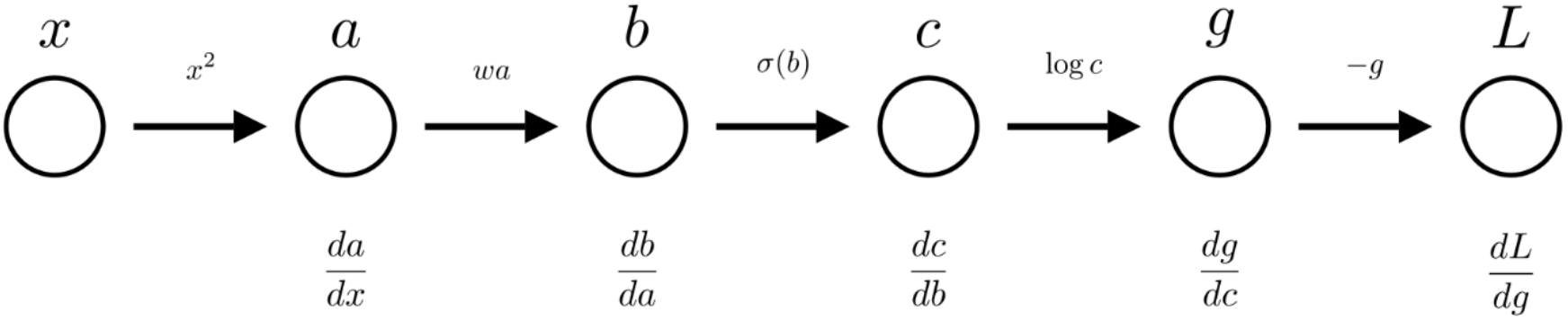
3: Multiply!

$$\frac{dL}{dx} = \frac{dL}{dg}\frac{dg}{dc}\frac{dc}{db}\frac{db}{da}\frac{da}{dx}$$

Computational graph

Nodes with value and derivative for each step

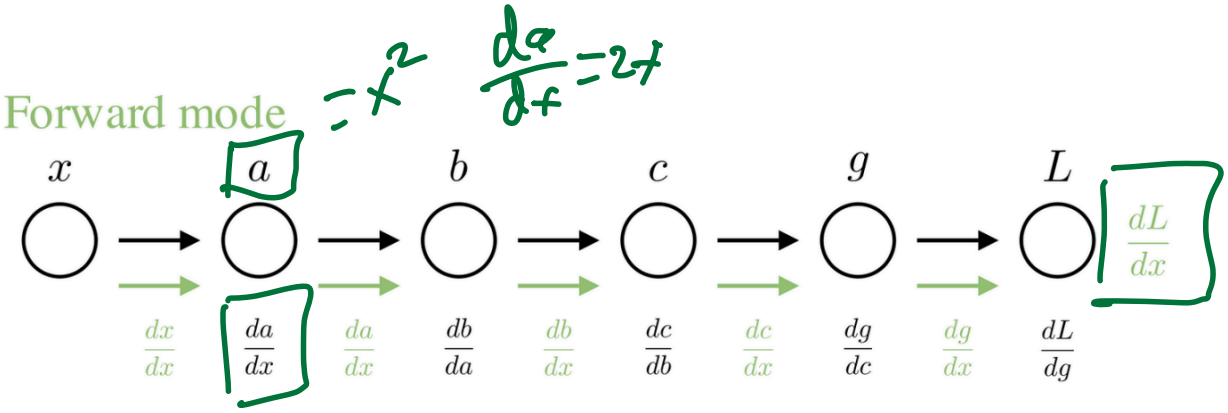
$$L = -\log \sigma\left(wx^2\right)$$



Forward-mode autodiff

Keep track of derivative with respect to input as we go!

$$L = -\log \sigma(wx^2)$$



E.g.: At node *b*

Inputs: a, $\frac{da}{dx}$

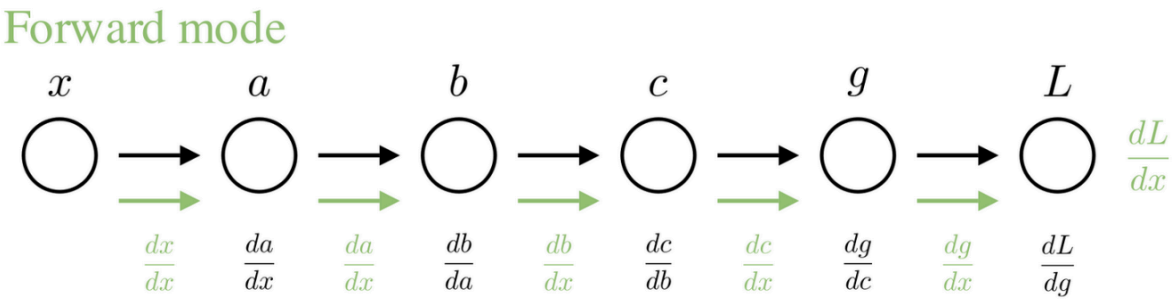
Compute: $\frac{db}{da} = w$

$\frac{db}{dx} = \frac{db}{da} \cdot \frac{da}{dx} = w(2x) = 2wx$

Forward-mode autodiff

Keep track of derivative with respect to input as we go!

$$L = -\log \sigma(wx^2)$$



E.g.: At node *b*

Inputs: $a, \quad \frac{da}{dx}$

Compute:

No need to keep!

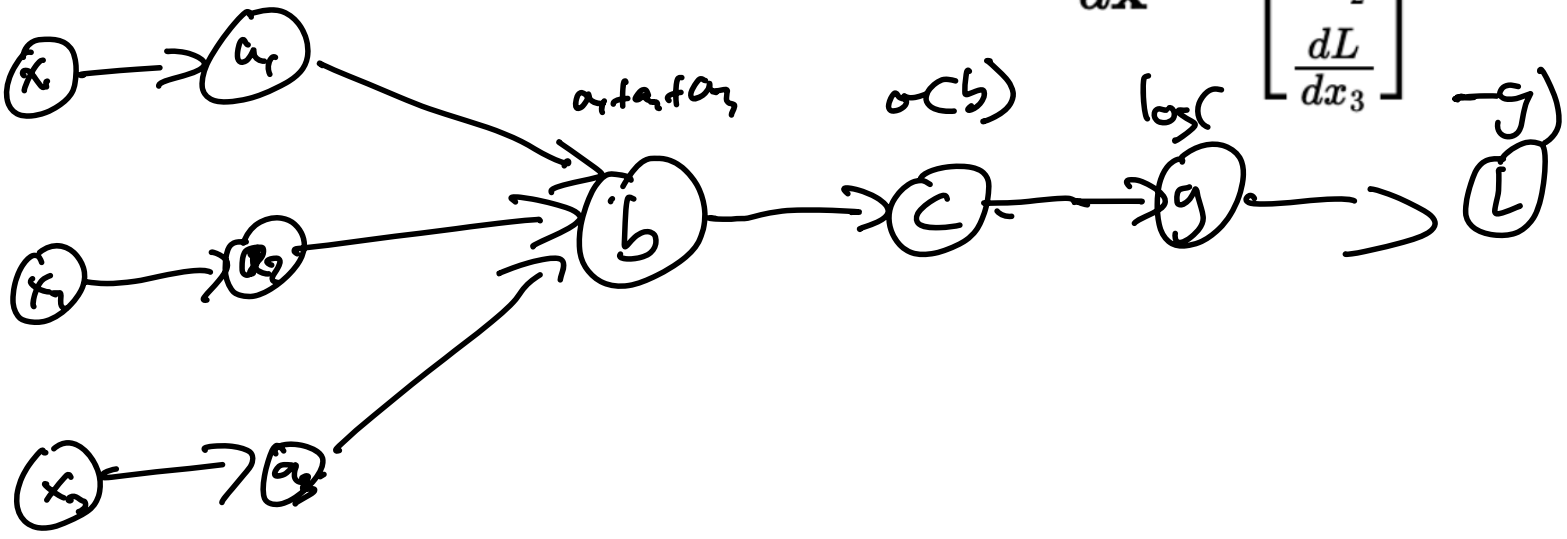
$$b = wa \qquad \frac{db}{da} = w$$

$$\frac{db}{dx} = \frac{db}{da}\frac{da}{dx}$$

Automatic differentiation with multiple inputs

$$-\log \sigma(w_1 x_1 + w_2 x_2 + w_3 x_3)$$

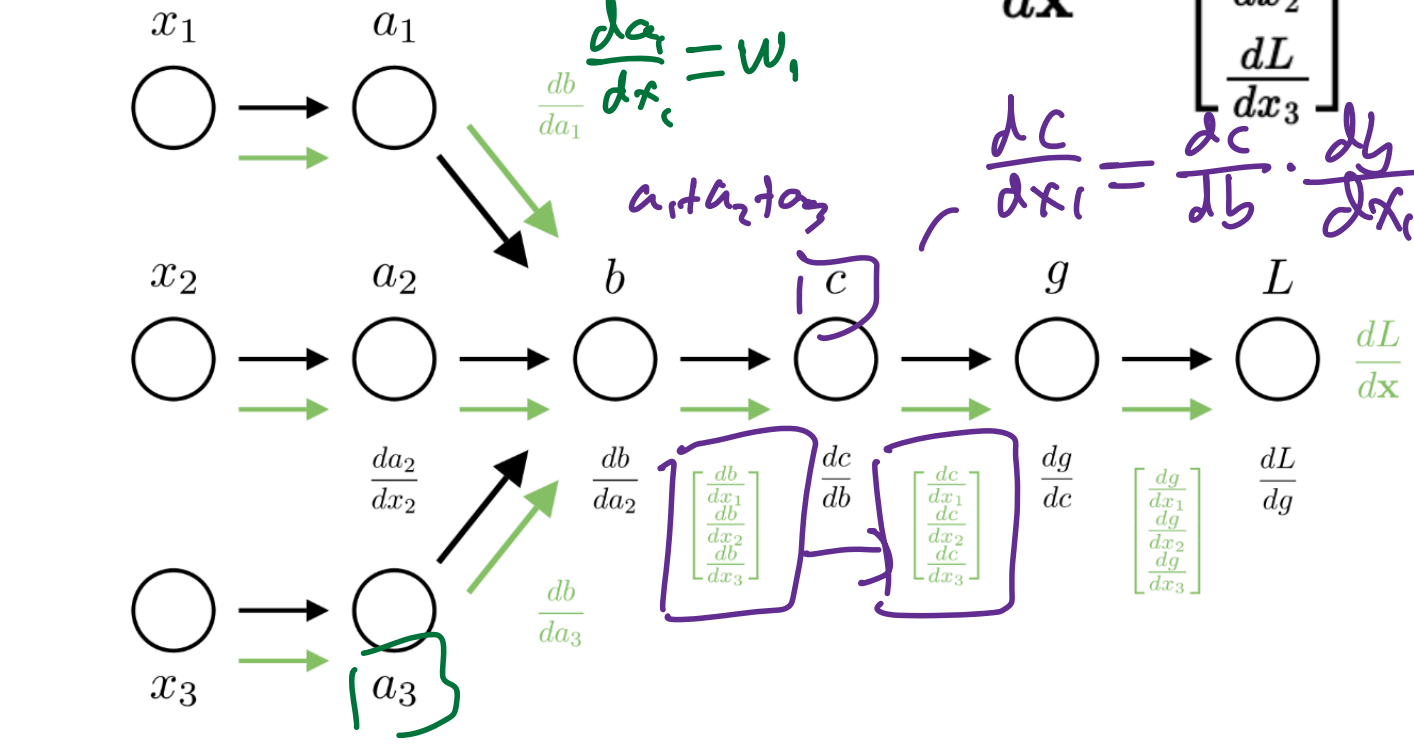
$$\frac{dL}{d\mathbf{x}} = \begin{bmatrix} \frac{dL}{dx_1} \\ \frac{dL}{dx_2} \\ \frac{dL}{dx_3} \end{bmatrix}$$



Automatic differentiation with multiple inputs

$$-\log \sigma(w_1 x_1 + w_2 x_2 + w_3 x_3)$$

$$\frac{dL}{d\mathbf{x}} = \begin{bmatrix} \frac{dL}{dx_1} \\ \frac{dL}{dx_2} \\ \frac{dL}{dx_3} \end{bmatrix}$$



Reusing values

```
def loss(x):
    a = x ** 2
    b = 5 * a
    c = log(a)
    g = b * c
    L = -g
    return L
```

$$a = x^2$$

$$b = 5a$$

$$c = \log a$$

$$g = bc$$

$$L = -b$$

Reusing values

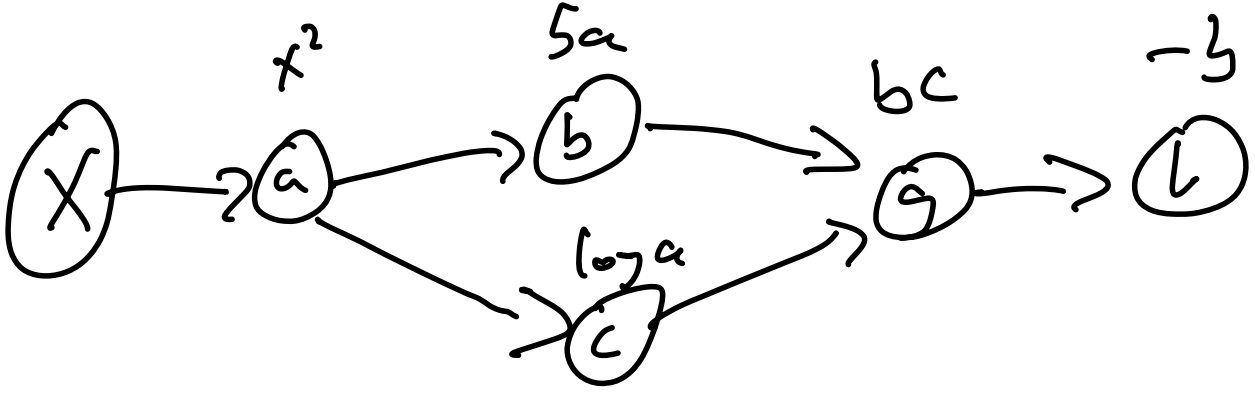
$$a = x^2$$

$$b = 5a$$

$$c = \log a$$

$$g = bc$$

$$L = -y$$



b:

$$\frac{dg}{dx} = \frac{dg}{db} \cdot \frac{db}{dx}$$

c:

$$\frac{dg}{dx} = \frac{dg}{dc} \cdot \frac{dc}{dx}$$

Reusing values

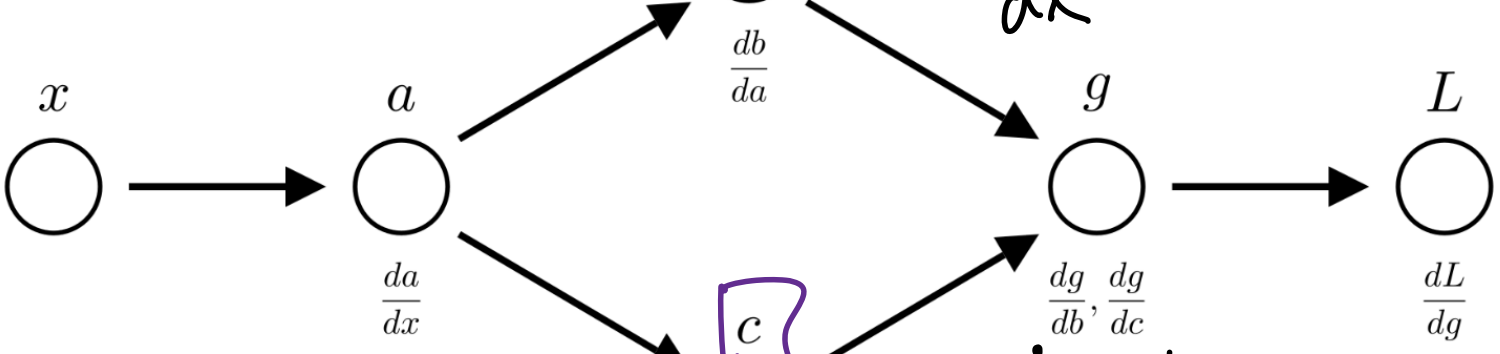
$$a = x^2$$

$$b = 5a$$

$$c = \log a$$

$$g = bc$$

$$L = -b$$



$$\frac{dg}{dx} = \frac{dg}{db} \cdot \frac{db}{dx} + \frac{dg}{dc} \cdot \frac{dc}{dx}$$

$$\frac{db}{dx} = \frac{db}{da} \cdot \frac{da}{dx}$$

$$\frac{dg}{db} = c$$

$$\frac{dg}{dc} = b$$

$$\frac{dg}{dx} =$$

$$\frac{dc}{da} \cdot \frac{db}{dx} \cdot \frac{dg}{db} \cdot \frac{da}{dx}$$

$L = -(5x^2)x^2 = -5x^4$

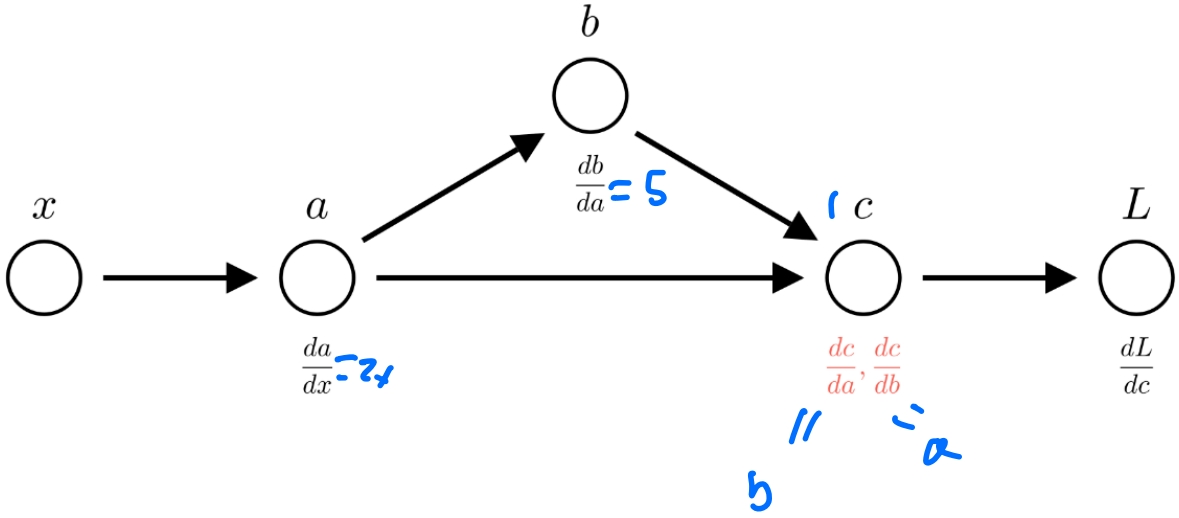
Partial derivatives revisited

$$a = x^2$$

$$b = 5a$$

$$c = ab$$

$$L = -c$$



$$\frac{dc}{da} =$$

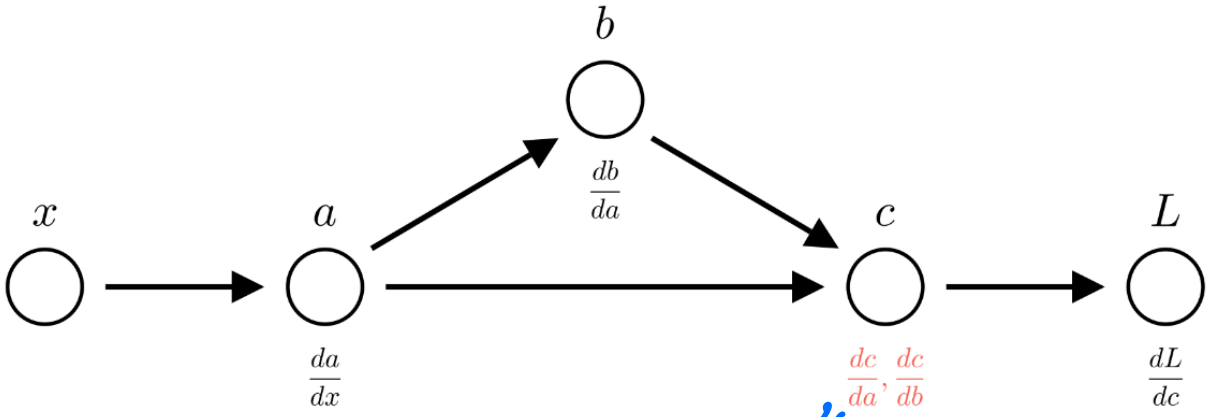
Partial derivatives revisited

$$a = x^2$$

$$b = 5a$$

$$c = ab$$

$$L = -c$$



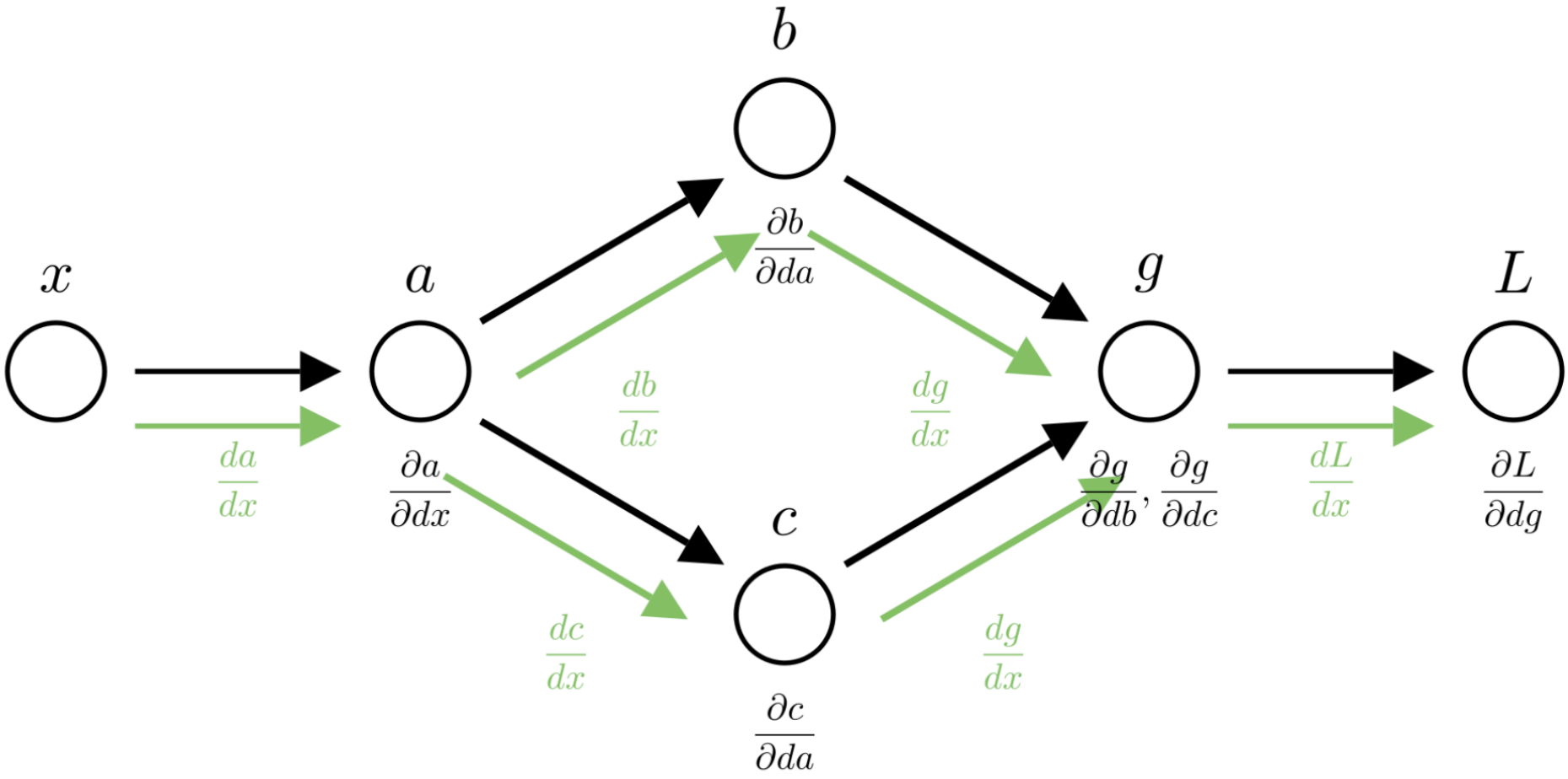
a 5

$\|$ $\|$

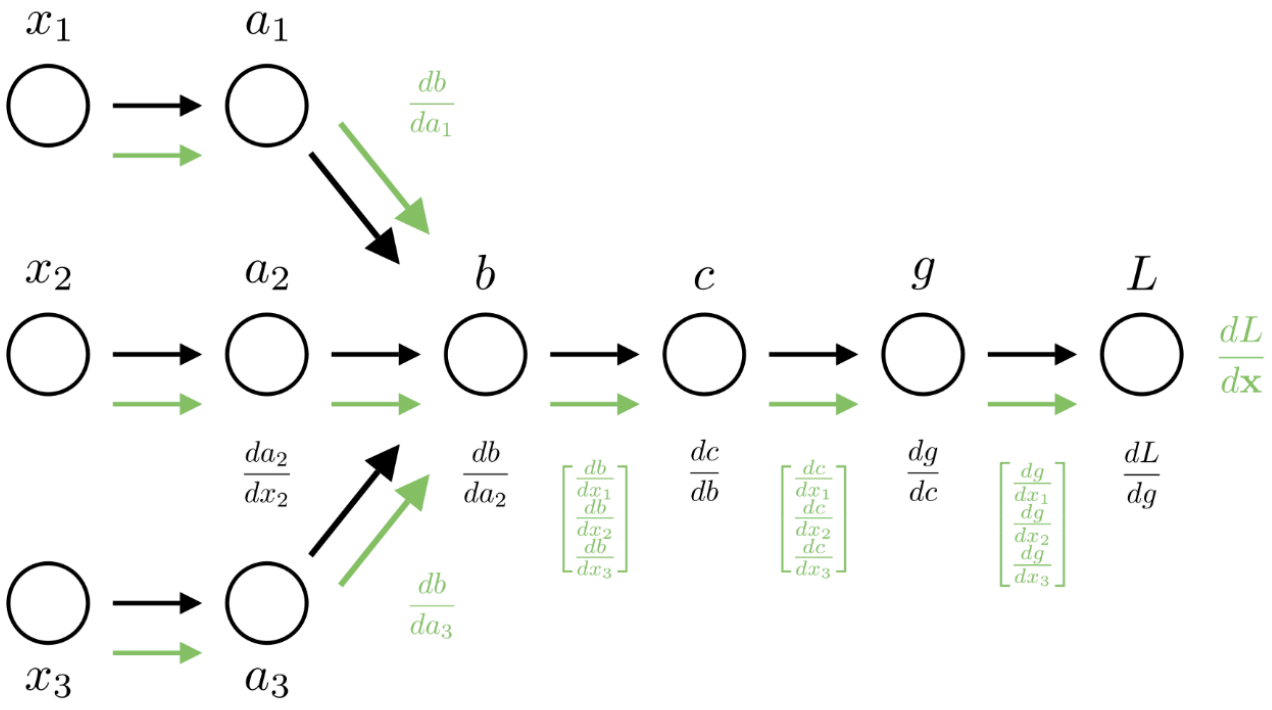
$5a$

$$\frac{dc}{da} = \frac{\partial c}{\partial a} + \frac{\partial c}{\partial b}\frac{\partial b}{\partial a} = 5a + 5a = 10a$$

Partial derivatives revisited

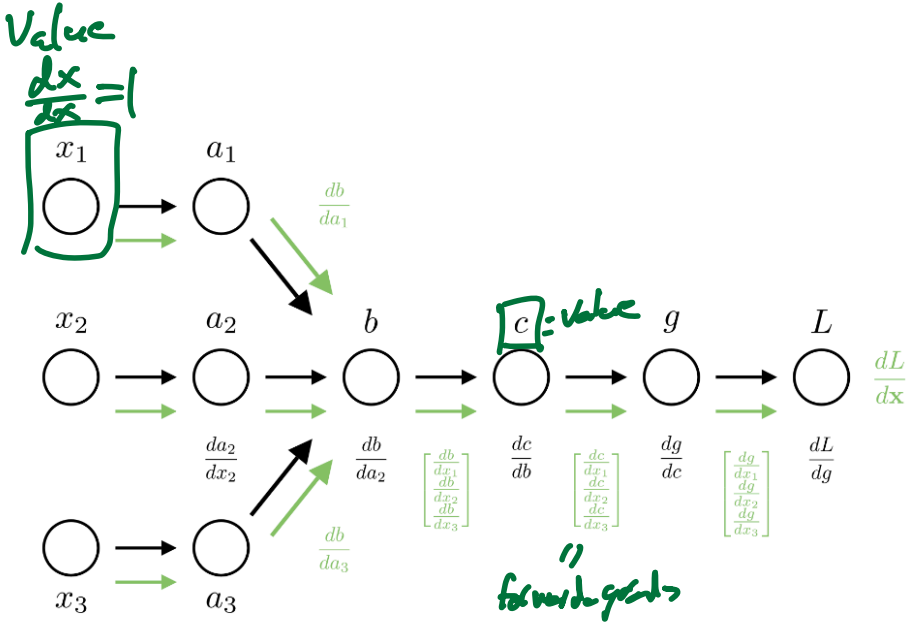


Implementing automatic differentiation



Implementing automatic differentiation

```python
class ForwardValue:
    '''
    Base class for automatic differentiation operations. Represents variable delcaration.
    Subclasses will overwrite func and grads to define new operations.

    Properties:
        parent_values    (list): A list of raw values of each input (as floats)
        value   (float): The value of the result of this operation
        forward_grads (dict): A dictionary mapping inputs to gradients
    '''

    def __init__(self, *args):
        self.parent_values = [arg.value if isinstance(arg, ForwardValue) else arg for arg in args]
        self.value = self.forward_pass(args)

        if len(self.forward_grads.keys()) == 0:
            self.forward_grads = {self: 1}


    def func(self, input):
        '''
        Compute the value of the operation given the inputs.
        For declaring a variable, this is just the identity function (return the input).

        Args:
            input (float): The input to the operation
        Returns:
            value (float): The result of the operation
        '''
        return input

    def grads(self, *args):
        '''
        Compute the derivative of the operation with respect to each input.
        In the base case the derivative of the identity function is just 1. (da/da = 1).

        Args:
            input (float): The input to the operation
        Returns:
            grads (tuple): The derivative of the operation with respect to each input
                           Here there is only a single input, so we return a length-1 tuple.
        '''
        return (1,)

    def forward_pass(self, args):
        # Calls func to compute the value of this operation
        self.forward_grads = {}
        return self.func(*self.parent_values)
```

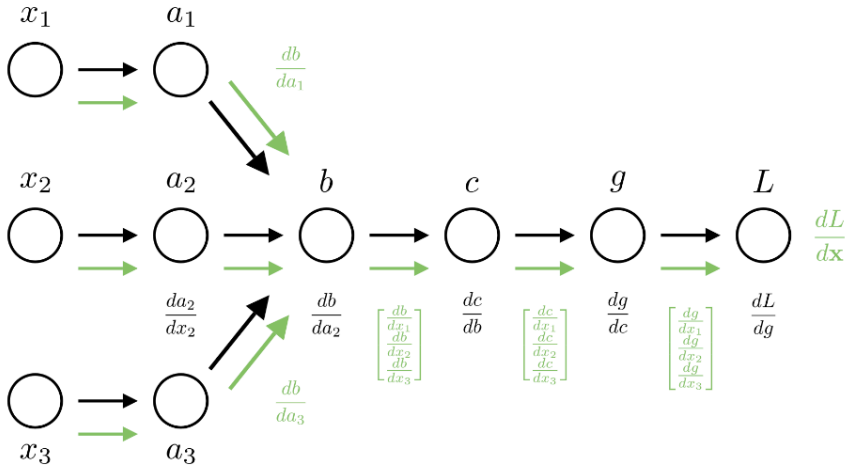


Implementing automatic differentiation

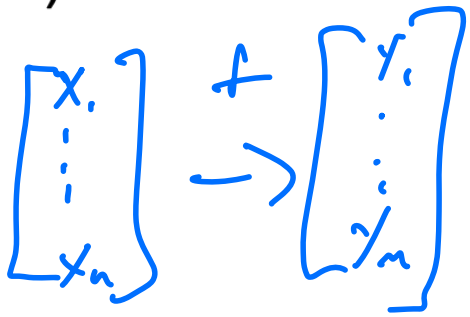
```python
class _add(ForwardValue):
    # Addition operator (a + b)
    def func(self, a, b):
        return a + b

    def grads(self, a, b):
        return 1., 1.
```

```python
class _neg(ForwardValue):
    # Negation operator (-a)
    def func(self, a):
        return -a

    def grads(self, a):
        return (-1.,)
```

```python
class _sub(ForwardValue):
    # Subtraction operator (a - b)
    def func(self, a, b):
        # Your code here

    def grads(self, a, b):
        # Your code here
```



Calculus for vectors (Jacobian)



Now let's consider a vector-valued function $f: \mathbb{R}^n \to \mathbb{R}^m$

e.g. $f(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_m)$

We define the instantaneous rate of change via the **Jacobian matrix**, whose entries are **partial derivatives**

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

The Jacobian is also called the **derivative** of f. It is a $m \times n$ matrix

Each partial derivative fixes the other inputs and treats f as scalar-input scalar-output

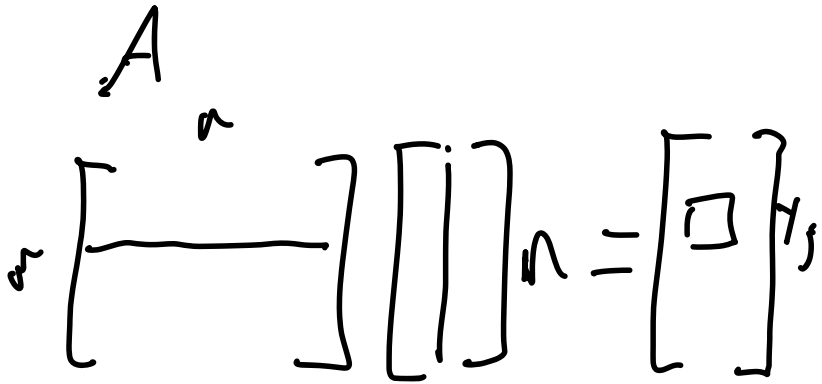
Calculus for vectors (Jacobian)

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = A$$

$$\frac{\partial y_j}{\partial x_i} =$$

$$\frac{\partial}{\partial x_i} y_j = \sum_{b=1}^{n} x_b A_{jb} = \frac{\partial}{\partial x_i} x_i A_{ji} = A_{ji}$$



$x \in \mathbb{R}^n$

Calculus for vectors (Jacobian)

$$\mathbf{y} = \mathbf{x}^2$$

$$y = \begin{bmatrix} x_1^2 \\ x_2^2 \\ x_3^2 \\ \vdots \end{bmatrix} \qquad y_i = x_i^2$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = n \begin{bmatrix} 2x_1 & & & \\ & 2x_2 & & \\ & & 2x_3 & \\ & & & \ddots \end{bmatrix}$$

$$\frac{dy_i}{dx_i} = 2x_i$$

$$\frac{dy_j}{dx_i} = 0$$

$j \neq i$

Jacobians and gradients

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

When $m = 1$, (scalar-valued output) the Jacobian is called the **gradient**

$$\frac{\partial f}{\partial x} = \left[\frac{\partial y}{\partial x_1} \quad \frac{\partial y}{\partial x_2} \quad \cdots \quad \frac{\partial y}{\partial x_n}\right] = \nabla_{\vec{x}} f$$

Notes:

- The Jacobian of a scalar-valued function is a row vector.
- The gradient is often described as a column vector

Chain rule for Jacobians

An easy extension of the chain rule exists:

Given $f: \mathbb{R}^k \to \mathbb{R}^m$ and $g: \mathbb{R}^n \to \mathbb{R}^k$

$$\underbrace{\frac{\partial f(g(x))}{\partial x}}_{m \times n} = \underbrace{\frac{\partial f(y)}{\partial y}}_{m \times k} \cdot \underbrace{\frac{\partial g(x)}{\partial x}}_{k \times n}$$

Where $y = g(x)$.

Forward-mode with vectors

$$\mathbf{a} = \mathbf{X}\mathbf{w}$$

$$\mathbf{b} = \mathbf{y} - \mathbf{a}$$

$$\mathbf{c} = \mathbf{b}^2$$

$$g = \sum_{i=1}^{N} c_i$$

$$L = \frac{1}{N} g$$

Forward-mode with vectors

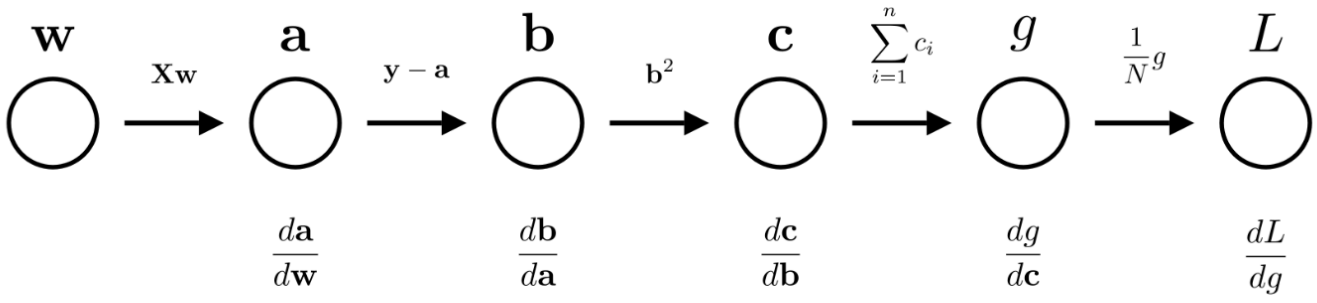
$$\mathbf{a} = \mathbf{X}\mathbf{w}$$

$$\mathbf{b} = \mathbf{y} - \mathbf{a}$$

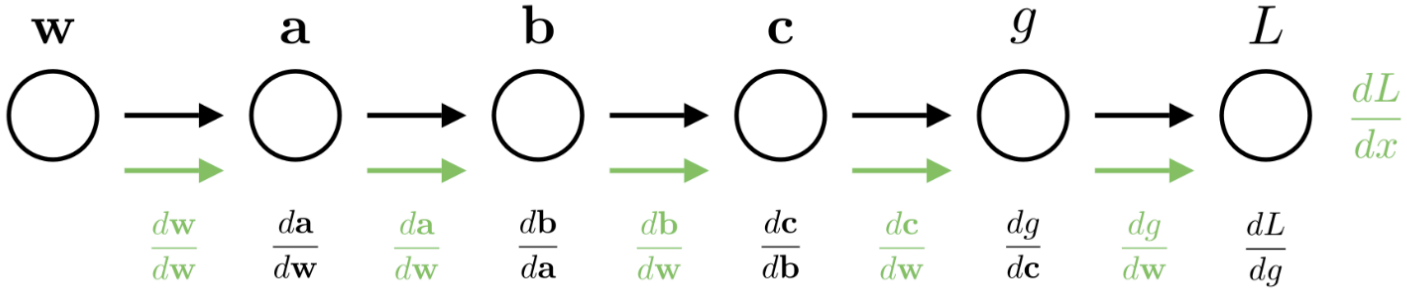
$$\mathbf{c} = \mathbf{b}^2$$

$$g = \sum_{i=1}^{N} c_i$$

$$L = \frac{1}{N} g$$



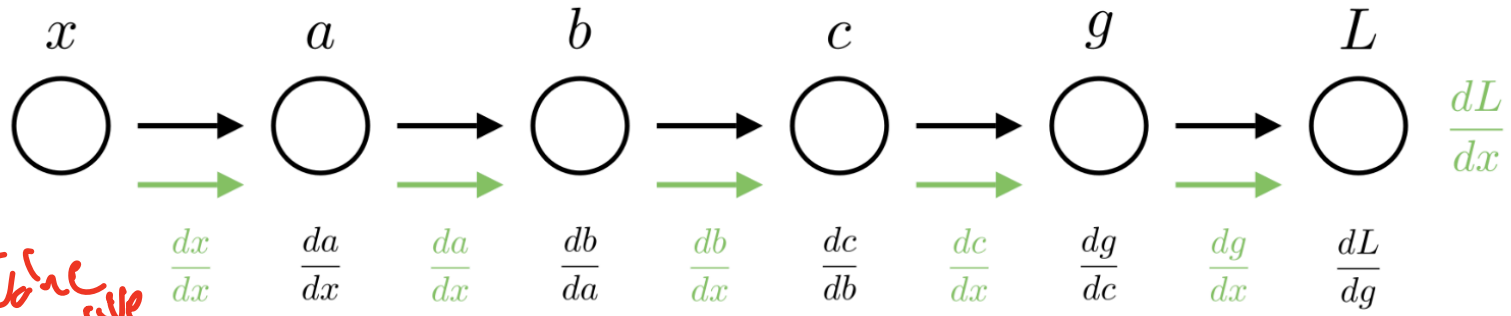
Forward mode



Reverse-mode AD

1-pass

Forward mode

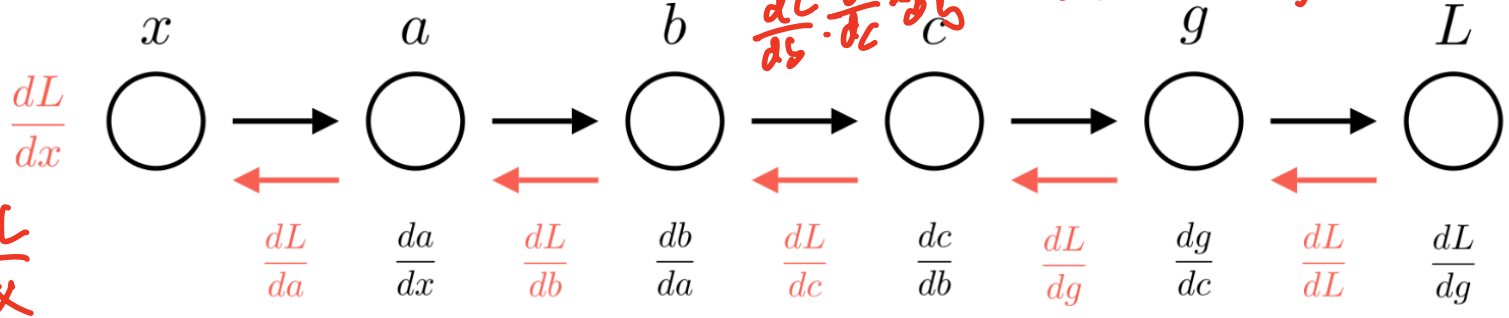


2-pass
1 computable
2 updated derivative

Reverse mode – Backpropagation

$\frac{dL}{dc} = \frac{dL}{dg} \cdot \frac{dg}{dc}$

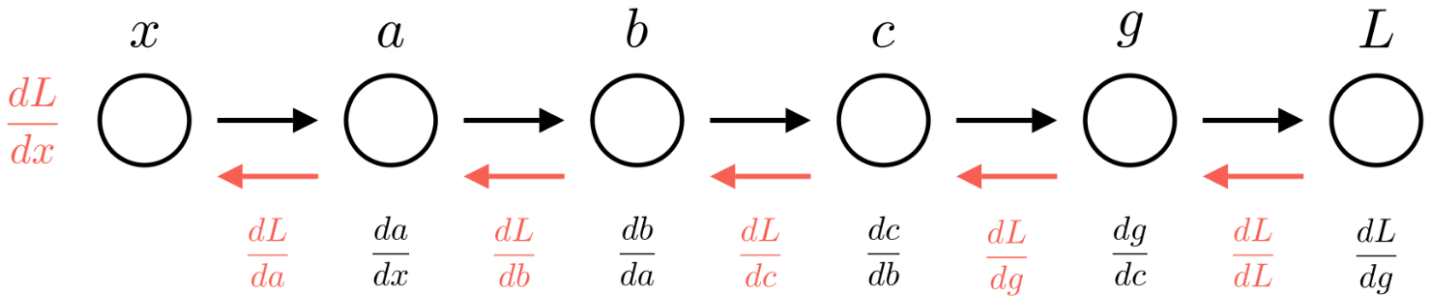
$\frac{dL}{db} \cdot \frac{db}{dc} \cdot \frac{dc}{db}$



$\frac{dL}{dx}$

Reverse mode

Reverse-mode AD



Define inputs

```
a = AutogradValue(5)
b = AutogradValue(2)
```

Compute operations

```
c = a + b
L = log(c)
```

AV

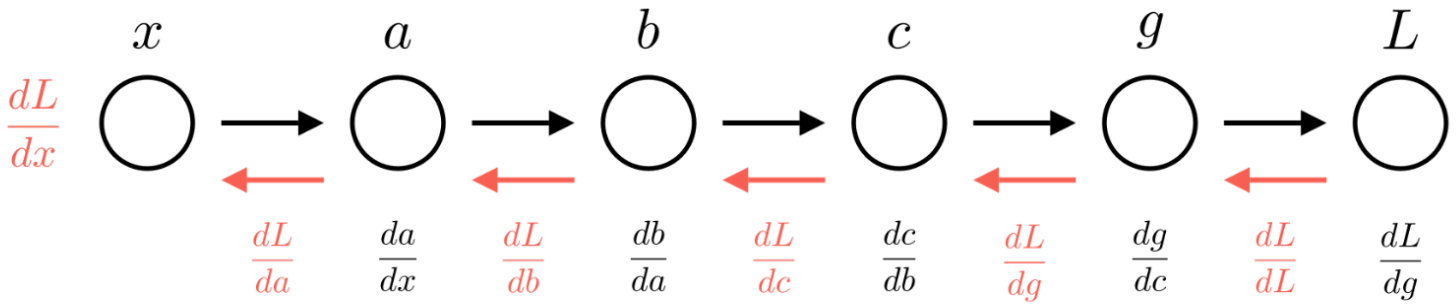
Get derivatives

```
L.backward()
dL_da = a.grad
```

Store

Reverse mode

Reverse-mode AD



Define inputs

```
a = AutogradValue(5)
b = AutogradValue(2)
```

Compute operations

```
c = a + b
L = log(c)
```

Get derivatives

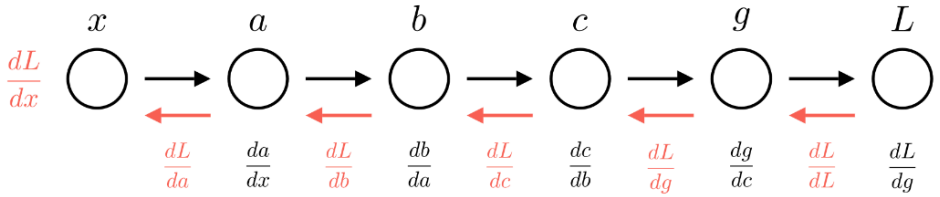
```
L.backward()
dL_da = a.grad
```

Operations with non-Autograd values

```
s = 4
L = s * a
dL_da = a.grad # Will work because a is an AutogradValue
dL_ds = s.grad # Will give an error because s is not an AutogradValue
```

Reverse mode

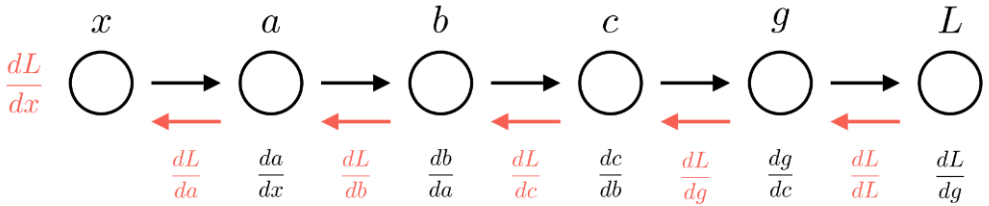
Reverse-mode AD




```python
class AutogradValue:
    '''
    Base class for automatic differentiation operations.
    Represents variable delcaration. Subclasses will overwrite
    func and grads to define new operations.

    Properties:
        parents (list):  A list of the inputs to the operation,
                         may be AutogradValue or float
        args    (list):  A list of raw values of each
                         input (as floats)
        grad    (float): The derivative of the final loss with
                         respect to this value (dL/da)
        value   (float): The value of the result of this operation
    '''

    def __init__(self, *args):
        self.parents = list(args)
        self.args = [arg.value if isinstance(arg, AutogradValue)
                     else arg
                     for arg in self.parents]
        self.grad = 0.
        self.value = self.forward_pass()

    def forward_pass(self):
        # Calls func to compute the value of this operation
        return self.func(*self.args)
```

Reverse mode

Reverse-mode AD




```
class AutogradValue:
    '''
    Base class for automatic differentiation operations.
    Represents variable delcaration. Subclasses will overwrite
    func and grads to define new operations.

    Properties:
        parents (list):  A list of the inputs to the operation,
                         may be AutogradValue or float
        args    (list):  A list of raw values of each
                         input (as floats)
        grad    (float): The derivative of the final loss with
                         respect to this value (dL/da)
        value   (float): The value of the result of this operation
    '''

    def __init__(self, *args):
        self.parents = list(args)
        self.args = [arg.value if isinstance(arg, AutogradValue)
                     else arg
                     for arg in self.parents]
        self.grad = 0.
        self.value = self.forward_pass()

    def forward_pass(self):
        # Calls func to compute the value of this operation
        return self.func(*self.args)
```

Base class (input)

```
class AutogradValue:
    def func(self, input):
        '''
        Compute the value of the operation given the inputs.
        For declaring a variable, this is just the identity
        function (return the input).

        Args:
            input (float): The input to the operation
        Returns:
            value (float): The result of the operation
        '''
        return input
```

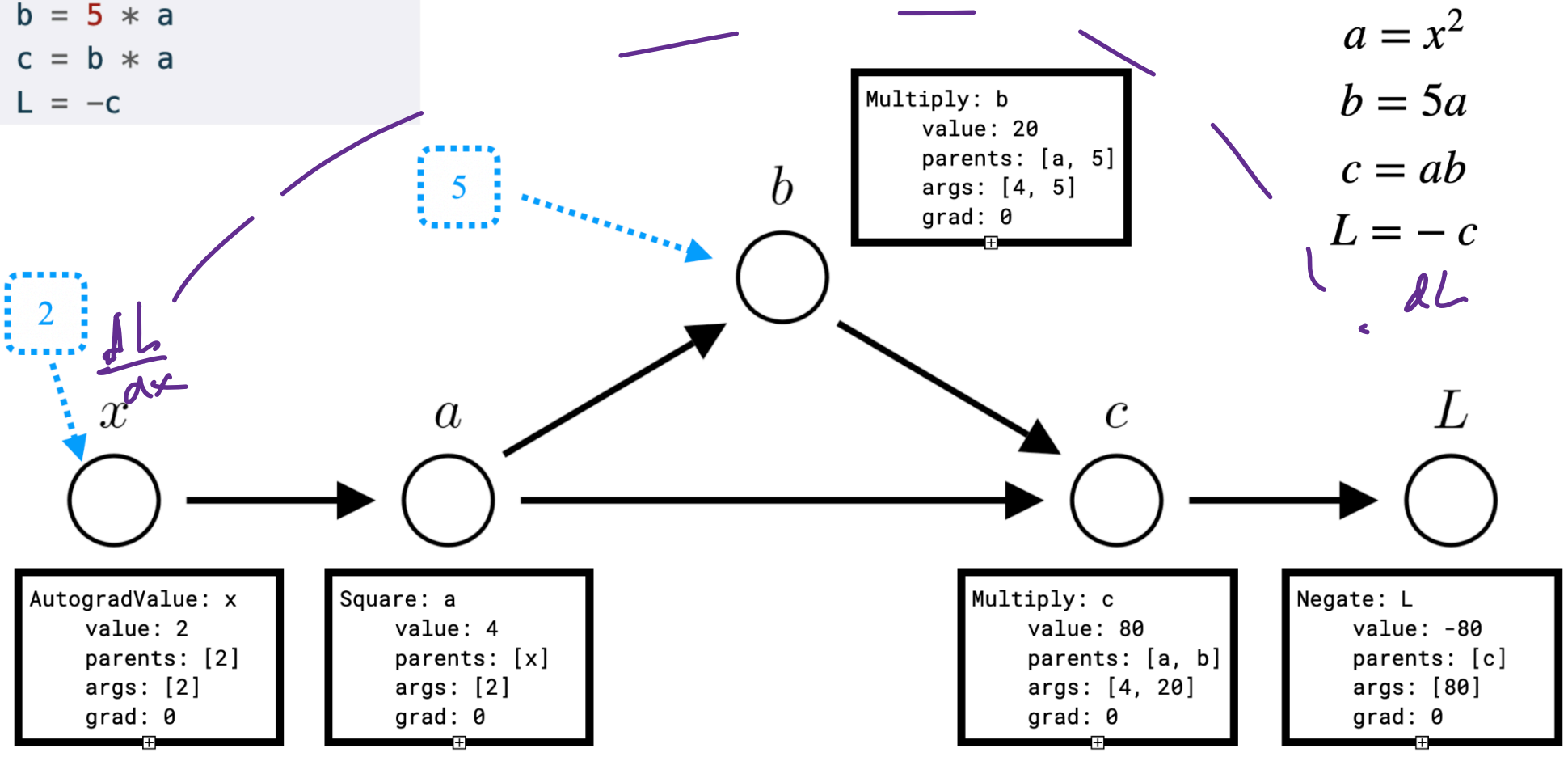
Sub-class (operation)

```
class _square(AutogradValue):
    # Square operator (a ** 2)
    def func(self, a):
        return a ** 2
```

```
x = AutogradValue(2)
a = x ** 2
b = 5 * a
c = b * a
L = -c
```

Forward pass

$$x = 2$$
$$a = x^2$$
$$b = 5a$$
$$c = ab$$
$$L = -c$$

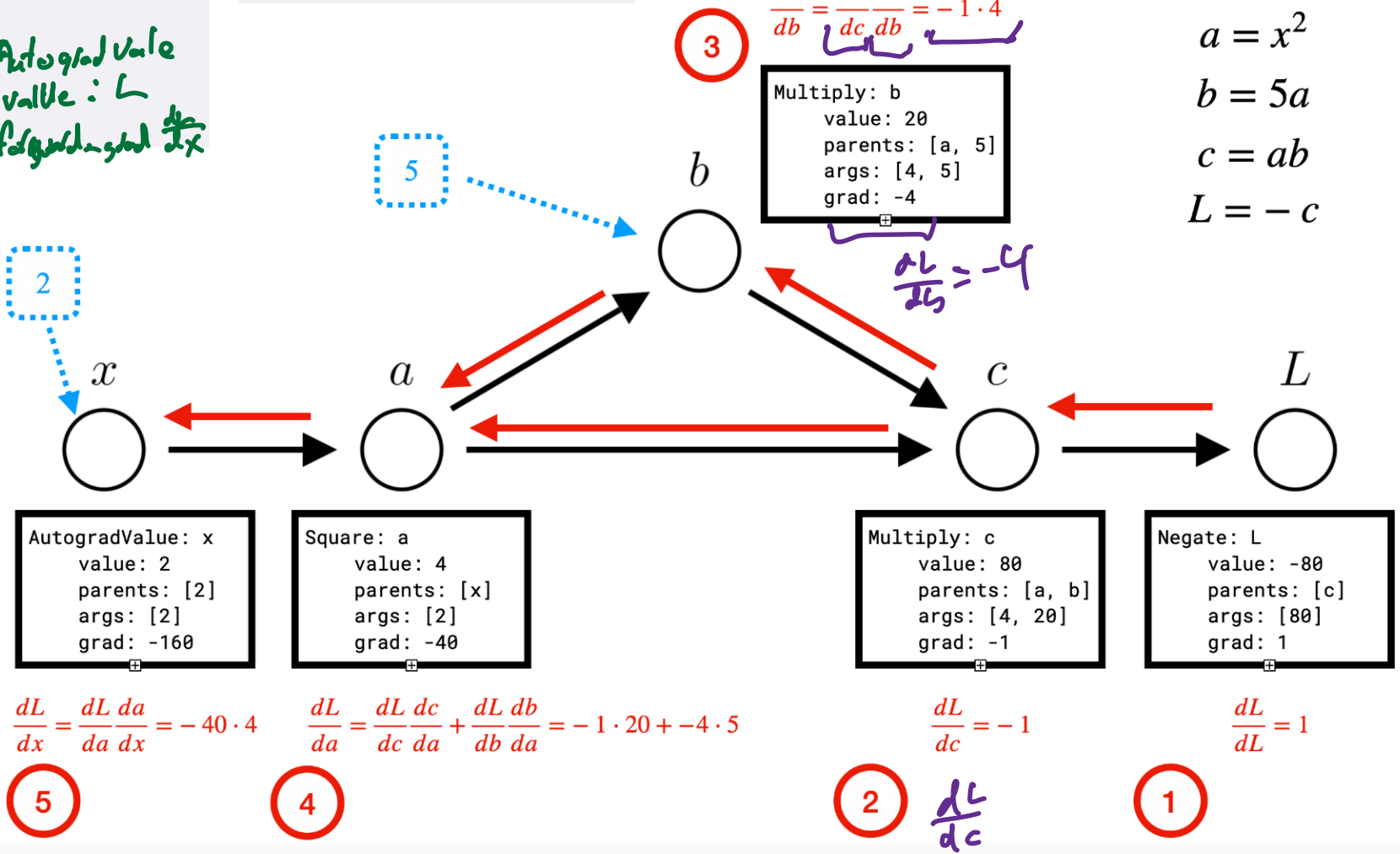


```
x = AutogradValue(2)
a = x ** 2
b = 5 * a
c = b * a
L = -c
```

```
L.backward()
print('dL_dx', x.grad)
```

Backward pass

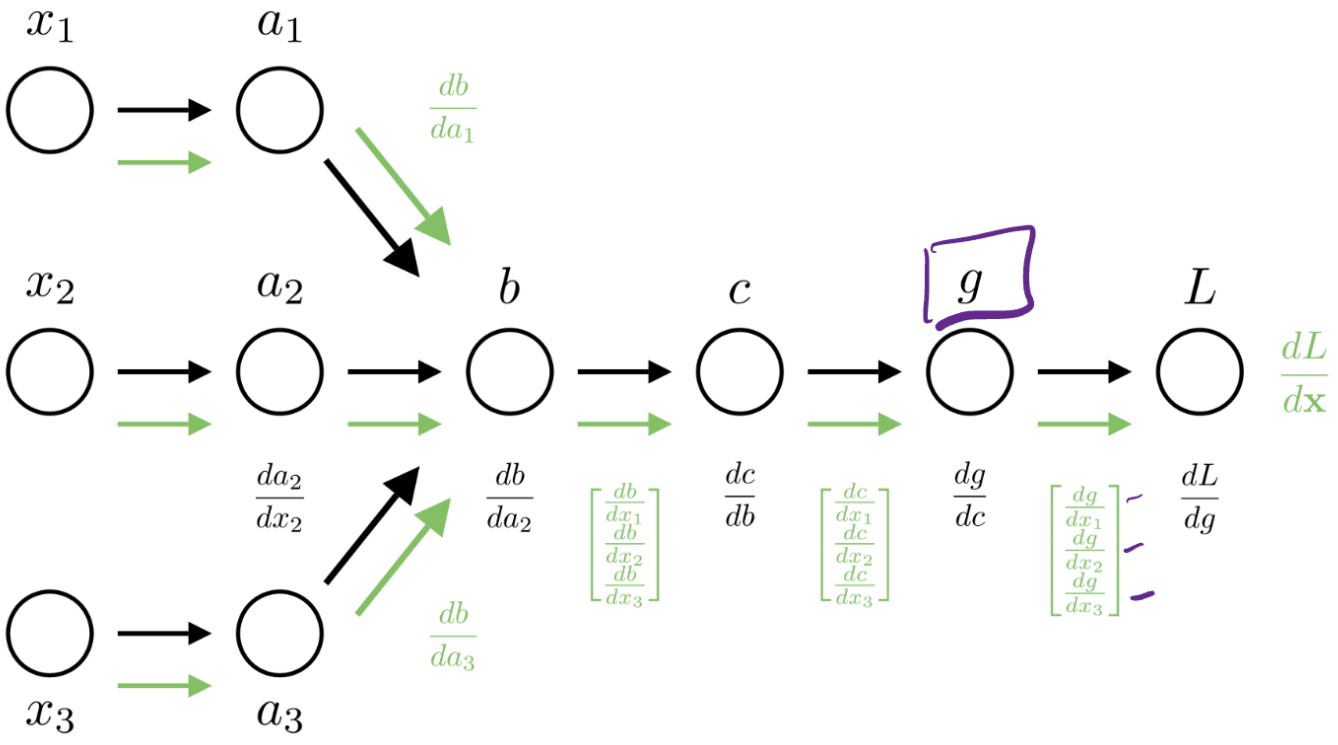
$$x = 2$$
$$a = x^2$$
$$b = 5a$$
$$c = ab$$
$$L = -c$$



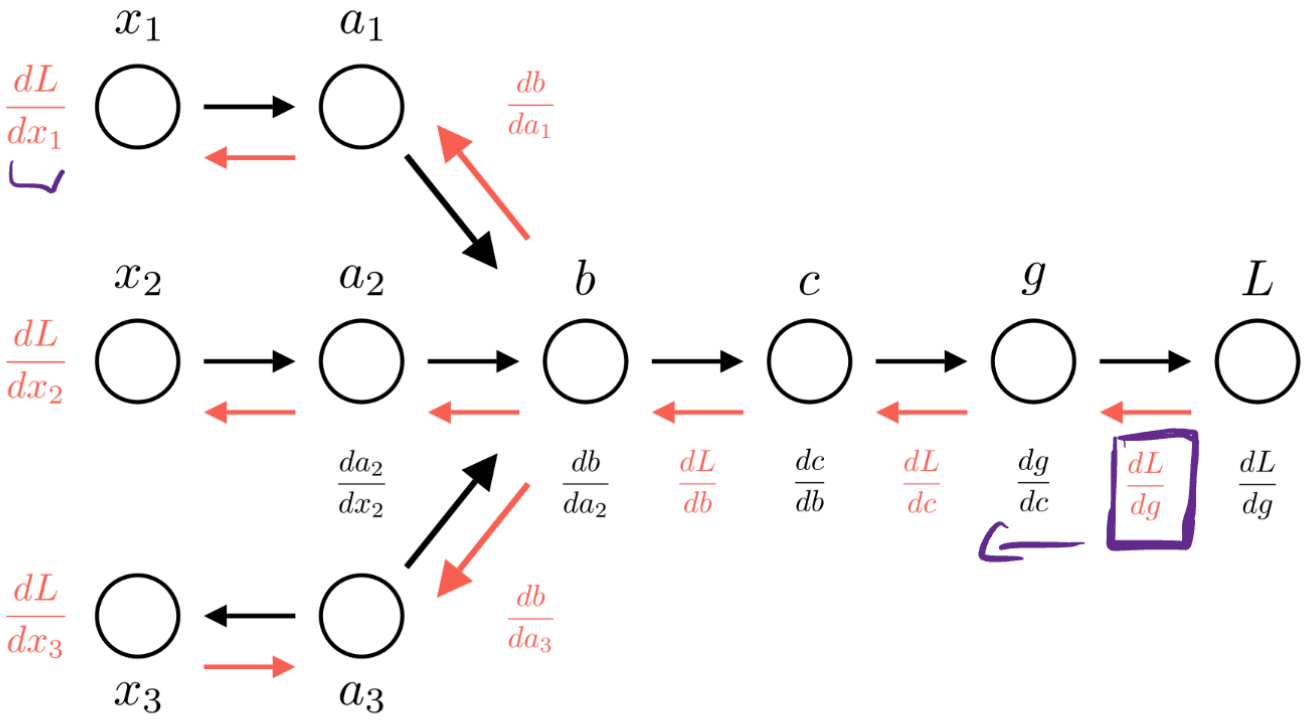
Backward pass

```
class AutogradValue:
        def grads(self, *args):
        '''
        Compute the derivative of the operation with respect to each input.
        In the base case the derivative of the identity function is just 1. (da/da = 1).

        Args:
            input (float): The input to the operation
        Returns:
            grads (tuple): The derivative of the operation with respect to each input
                           Here there is only a single input, so we return a length-1 t
        '''
        return (1,)

class _square(AutogradValue):
    # Square operator (a ** 2)
    def grads(self, a):
        return (2 * a,)
```

Why reverse mode?



Why reverse mode?



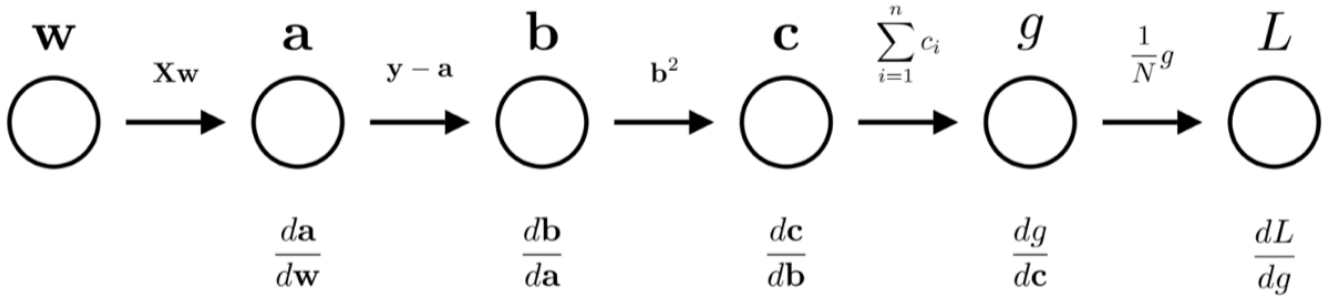
$$\mathbf{a} = \mathbf{X}\mathbf{w}$$

$$\mathbf{b} = \mathbf{y} - \mathbf{a}$$

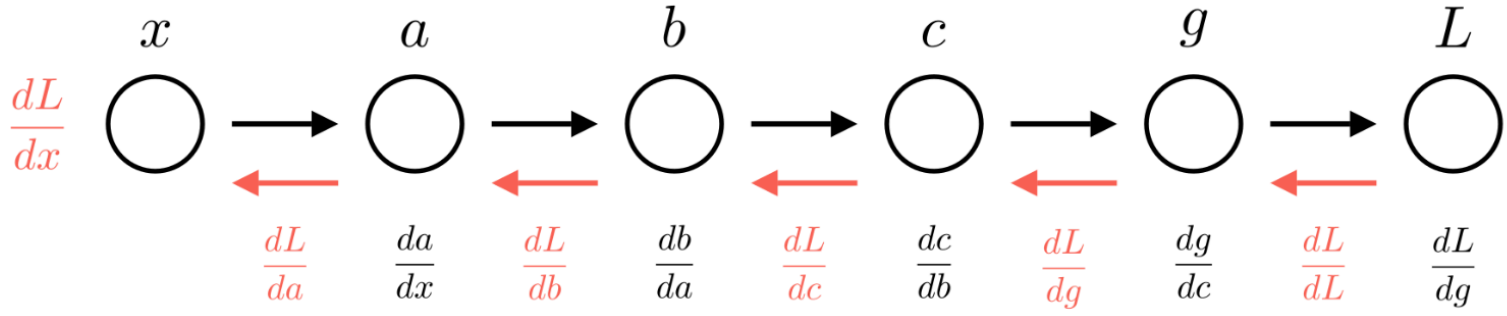
$$\mathbf{c} = \mathbf{b}^2$$

$$g = \sum_{i=1}^{N} c_i$$

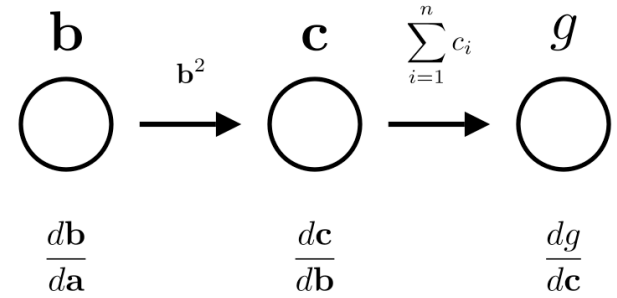
$$L = \frac{1}{N} g$$



Reverse mode

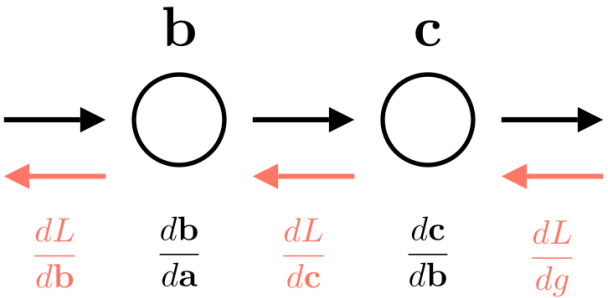


Jacobian computation



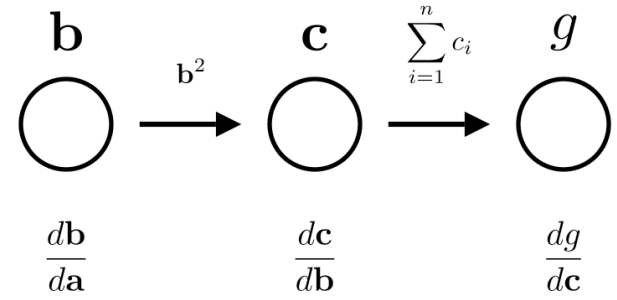
$$\mathbf{c} = \mathbf{b}^2, \quad \mathbb{R}^n \to \mathbb{R}^n$$

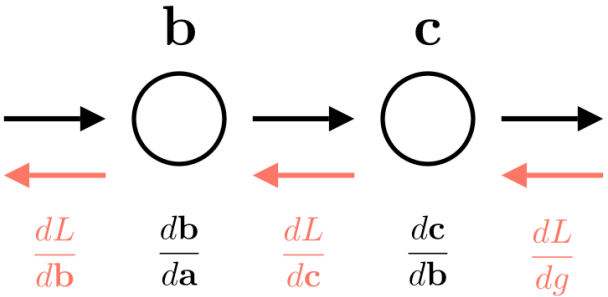
$$\frac{d\mathbf{c}}{d\mathbf{b}} =$$



$$\frac{dL}{d\mathbf{b}} = \frac{dL}{d\mathbf{c}}\frac{d\mathbf{c}}{d\mathbf{b}} =$$

Jacobian computation



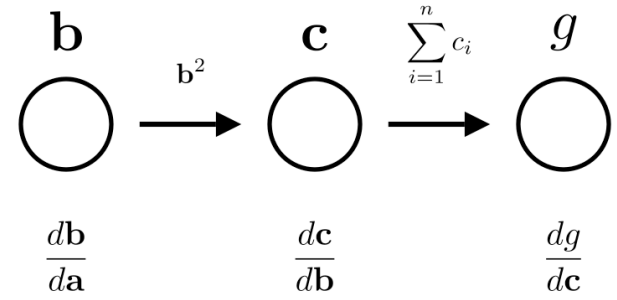


$$\mathbf{c} = \mathbf{b}^2, \quad \mathbb{R}^n \to \mathbb{R}^n$$

$$\frac{d\mathbf{c}}{d\mathbf{b}} = \begin{bmatrix} \frac{\partial c_1}{\partial b_1} & 0 & \dots & 0 \\ 0 & \frac{\partial c_2}{\partial b_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{\partial c_N}{\partial b_d} \end{bmatrix}$$

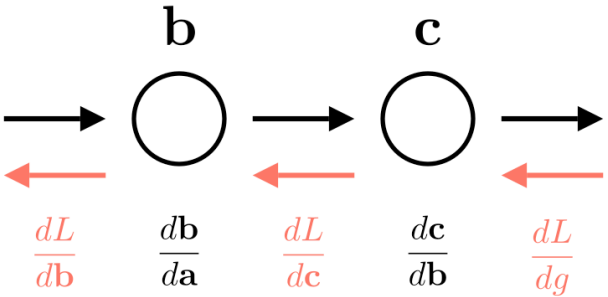
$$\frac{dL}{d\mathbf{b}} = \frac{dL}{d\mathbf{c}} \frac{d\mathbf{c}}{d\mathbf{b}} =$$

Vector-Jacobian Product



$$\mathbf{c} = \mathbf{b}^2, \quad \mathbb{R}^n \to \mathbb{R}^n$$

$$\frac{d\mathbf{c}}{d\mathbf{b}} = \begin{bmatrix} \frac{\partial c_1}{\partial b_1} & 0 & \dots & 0 \\ 0 & \frac{\partial c_2}{\partial b_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{\partial c_N}{\partial b_d} \end{bmatrix}$$



$$\frac{dL}{d\mathbf{b}} = \frac{dL}{d\mathbf{c}}\frac{d\mathbf{c}}{d\mathbf{b}} = \begin{bmatrix} \frac{dL}{dc_1}\frac{dc_1}{db_1} \\ \frac{dL}{dc_2}\frac{dc_2}{db_2} \\ \vdots \\ \frac{dL}{dc_N}\frac{dc_N}{db_N} \end{bmatrix}$$

Jacobian computation

$$\mathbf{a} = \mathbf{X}\mathbf{w}, \qquad \frac{d\mathbf{a}}{d\mathbf{X}}$$

$$\frac{dL}{d\mathbf{X}} = \frac{dL}{d\mathbf{a}}^{T} \frac{d\mathbf{a}}{d\mathbf{X}}$$

$$\frac{dL}{dX_{jk}} = \sum_{i=1}^{N} \frac{\partial L}{\partial a_i} \frac{\partial a_i}{\partial X_{jk}}$$

Jacobian computation

$$\mathbf{a} = \mathbf{X}\mathbf{w}, \qquad \frac{d\mathbf{a}}{d\mathbf{X}}$$

$$\frac{dL}{d\mathbf{X}} = \frac{dL}{d\mathbf{a}}^{T} \frac{d\mathbf{a}}{d\mathbf{X}}$$

$$\frac{dL}{dX_{jk}} = \sum_{i=1}^{N} \frac{\partial L}{\partial a_i} \frac{\partial a_i}{\partial X_{jk}}$$

```
dL_da, da_dx  # The computed gradients/jacobians
da_dx = da_dx.reshape((da_dx.shape[0], -1))
dL_dx = np.dot(dL_da, da_dx)
dL_dx = dL_dx.reshape(x.shape)
```

Jacobian computation

$$\mathbf{a} = \mathbf{X}\mathbf{w}, \quad \frac{d\mathbf{a}}{d\mathbf{X}}$$

$$\frac{dL}{d\mathbf{X}} = \frac{dL}{d\mathbf{a}}^T \frac{d\mathbf{a}}{d\mathbf{X}}$$

$$\frac{dL}{dX_{jk}} = \sum_{i=1}^{N} \frac{\partial L}{\partial a_i} \frac{\partial a_i}{\partial X_{jk}}$$

$$\mathbf{a} = \mathbf{X}\mathbf{w}, \quad \frac{d\mathbf{a}}{d\mathbf{X}}$$

$$a_i = \sum_{k=1}^{d} X_{ik} w_k$$

$$\frac{\partial a_i}{\partial X_{jk}} = \frac{\partial}{\partial X_{jk}} \sum_{k=1}^{d} X_{ik} w_k = \mathbb{I}(i=j) w_k$$

$$\frac{\partial L}{\partial X_{ik}} = \frac{\partial L}{\partial a_i} \frac{\partial a_i}{\partial X_{ik}} = \frac{\partial L}{\partial a_i} w_k$$

Jacobian computation

```
w_mat = w.reshape((1, -1)) # reshape w to 1 x d
dL_da_mat = dL_da.reshape((-1, 1)) # reshape dL_da to N x 1
dL_dx = w_mat * dL_da_mat # dL_dx becomes N x d, entry ik = dL_da_i * w_k
```

```
dL_da, da_dx  # The computed gradients/jacobians
da_dx = da_dx.reshape((da_dx.shape[0], -1))
dL_dx = np.dot(dL_da, da_dx)
dL_dx = dL_dx.reshape(x.shape)
```

$$\mathbf{a} = \mathbf{X}\mathbf{w}, \qquad \frac{d\mathbf{a}}{d\mathbf{X}}$$

$$a_i = \sum_{k=1}^{d} X_{ik} w_k$$

$$\frac{\partial a_i}{\partial X_{jk}} = \frac{\partial}{\partial X_{jk}} \sum_{k=1}^{d} X_{ik} w_k = \mathbb{I}(i=j) w_k$$

$$\frac{\partial L}{\partial X_{ik}} = \frac{\partial L}{\partial a_i} \frac{\partial a_i}{\partial X_{ik}} = \frac{\partial L}{\partial a_i} w_k$$

Vector-Jacobian products

$$f(\mathbf{x}) = \mathbf{x} + \mathbf{a}$$

$$\mathbf{v}^T \frac{d\mathbf{f}}{d\mathbf{x}} =$$

$$f(\mathbf{x}) = \log \mathbf{x}$$

$$\mathbf{v}^T \frac{d\mathbf{f}}{d\mathbf{x}} =$$