# Computation Graphs + Backpropagation

LING 575K Deep Learning for NLP

Shane Steinert-Threlkeld

April 12 2021

# Today's Plan

- Computation graph abstraction
- Backpropagation
  - "Calculus on computation graphs"
- Forward/backward API

# Announcements

- Thanks for helping iron out all the little kinks :)
- Updated absolute path for default argument for training_data
- Added some minimal unit tests for individual methods
  - hw2/test_all.py
  - Run `pytest` from your local directory, with environment activated
- In hw2, log = ln (natural logarithm), pdf updated
- vocabulary.py in hw1/ref
  - You can symbolic link to it from your directory to use:
  - `ln -s /dropbox/20-21/575k/hw1/ref/vocabulary.py vocabulary.py`

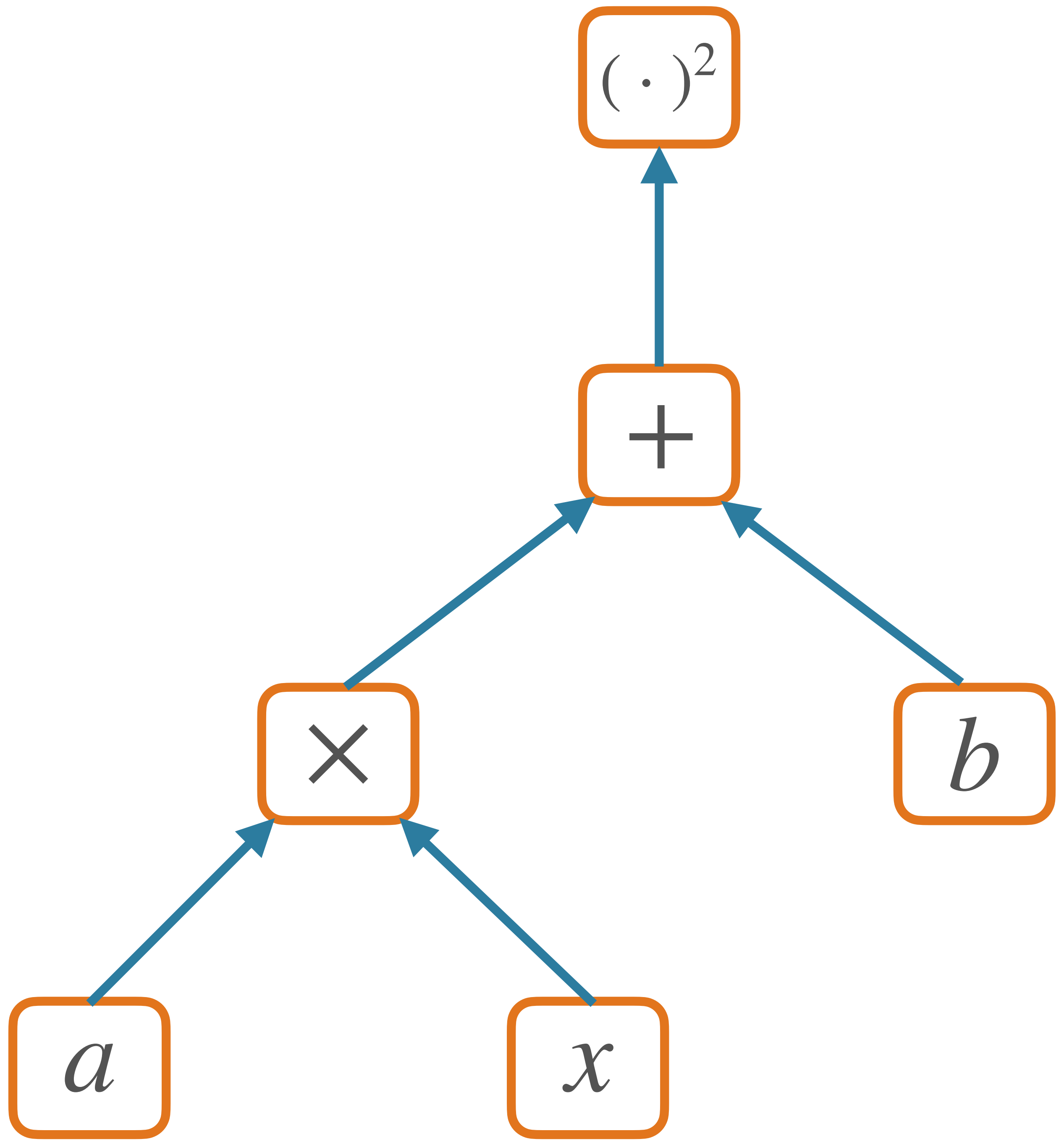
# Computation Graphs

# What is a computation graph?

- The “descriptive” language of deep learning frameworks
  - e.g. TensorFlow, PyTorch
- Essentially, “parse trees” of mathematical expressions
  - Captures dependence between
- Two types of computation:
  - Forward: compute outputs given inputs
  - Backward: compute gradients

# Computation Graph Example

$$f(x; a, b) = (ax + b)^2$$



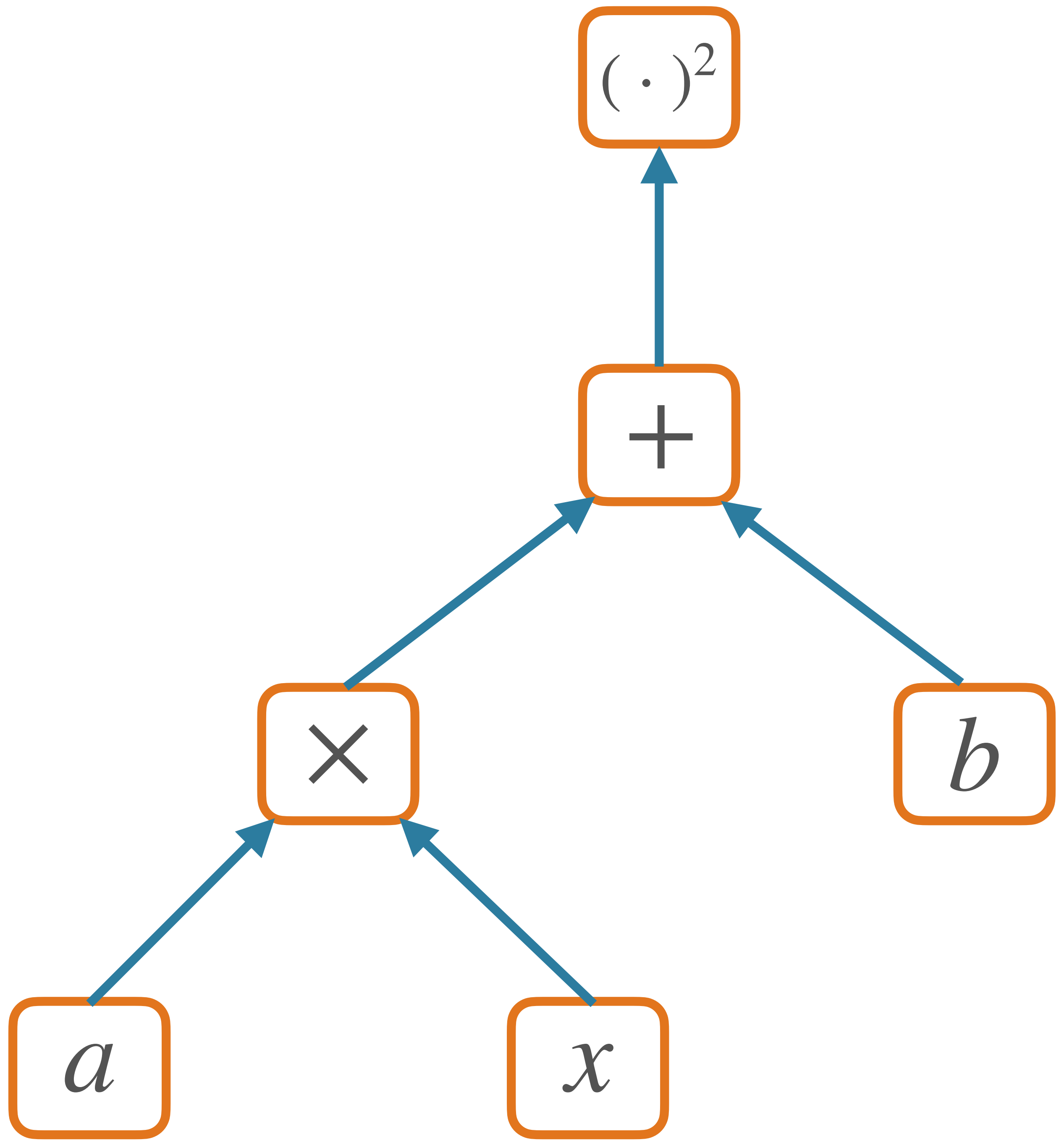
# Forward Pass

- Compute output(s) given inputs
  - Inputs: leaf nodes; need values
  - Outputs: those with no children

- Forward computation:
  - Loop over nodes in topological order [i.e. children after parents]
    - Compute value of a node given values of its parent nodes

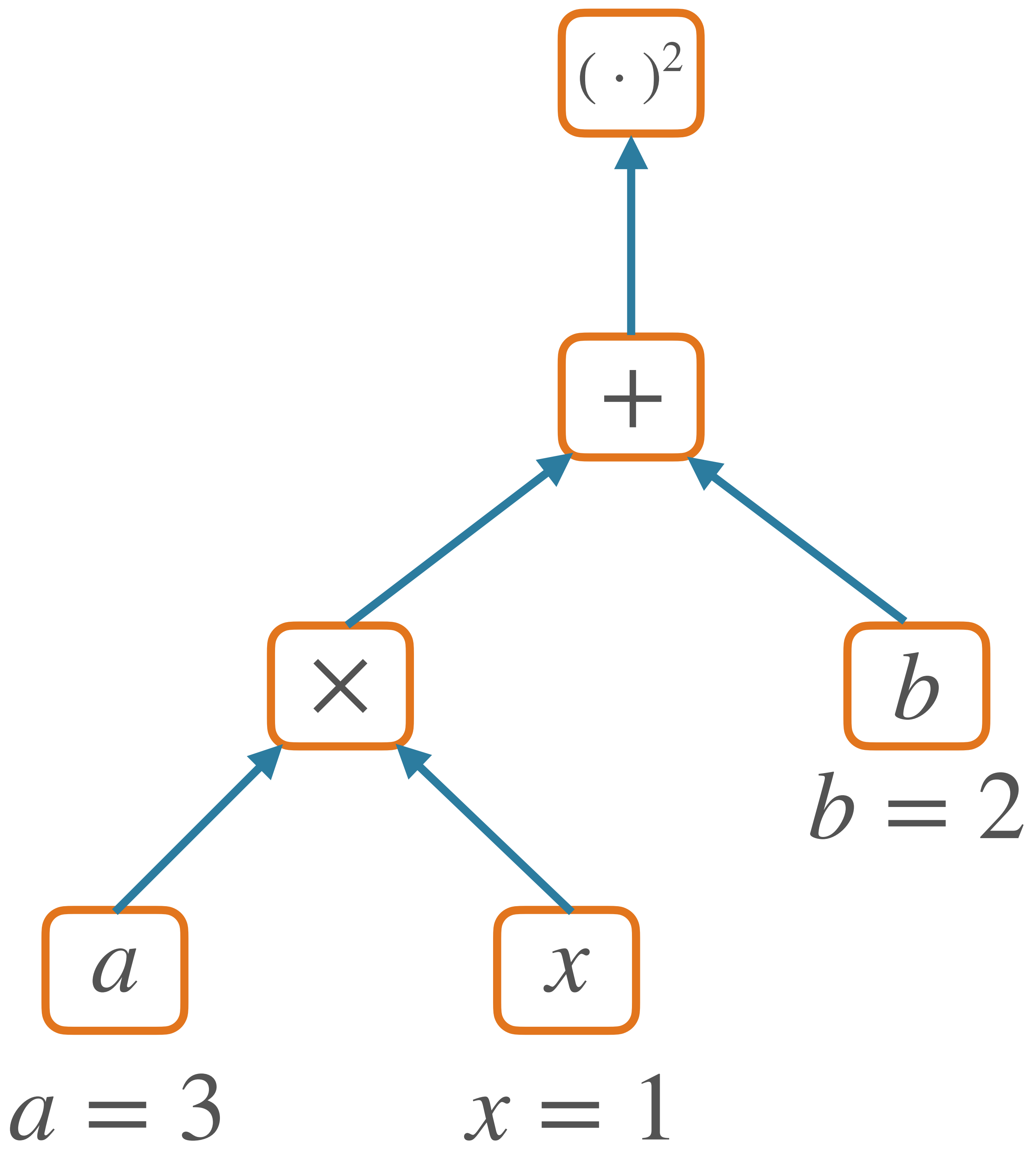
# Computation Graph Example

$$f(x; a, b) = (ax + b)^2$$



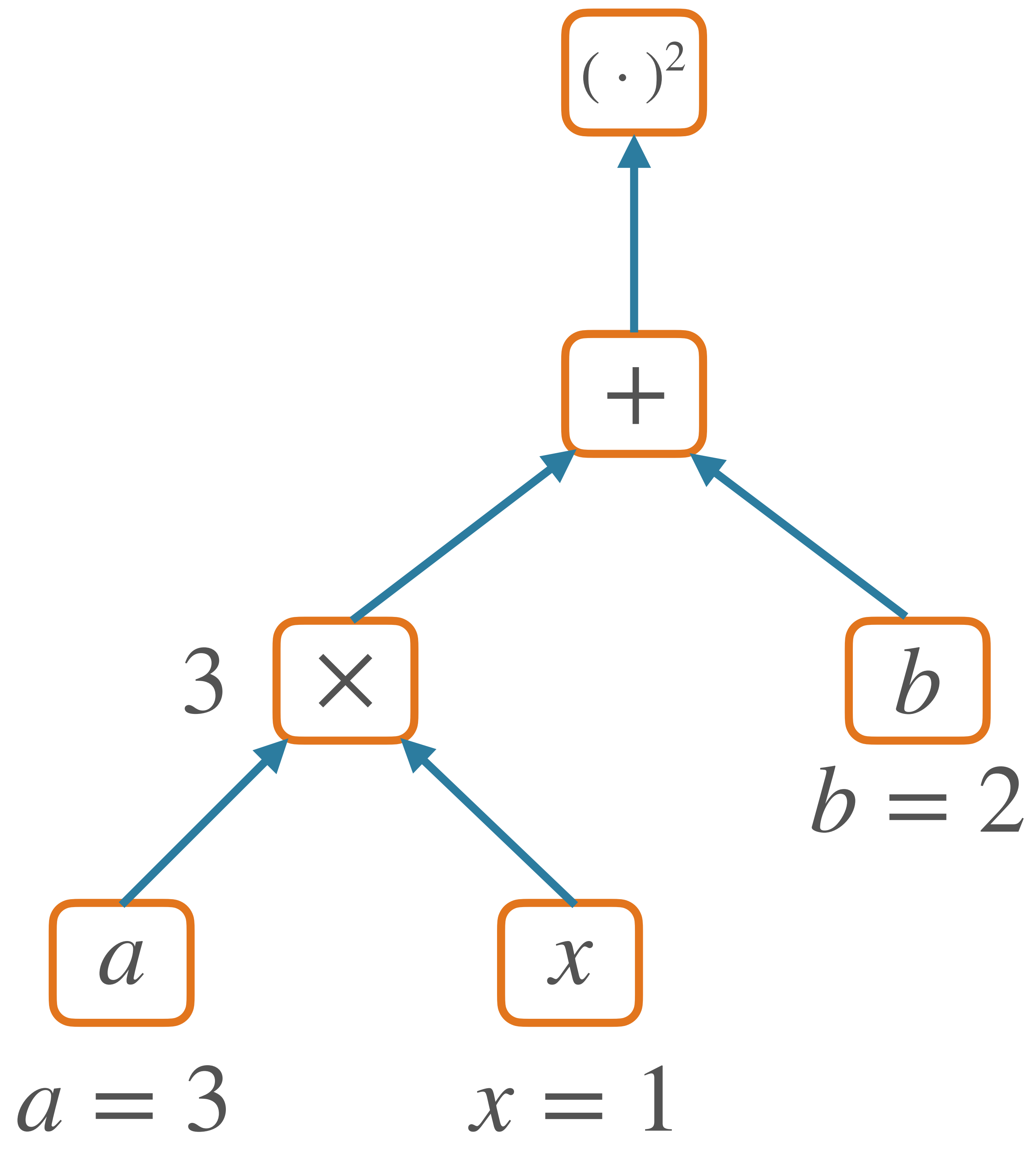
# Computation Graph Example

$$f(x; a, b) = (ax + b)^2$$



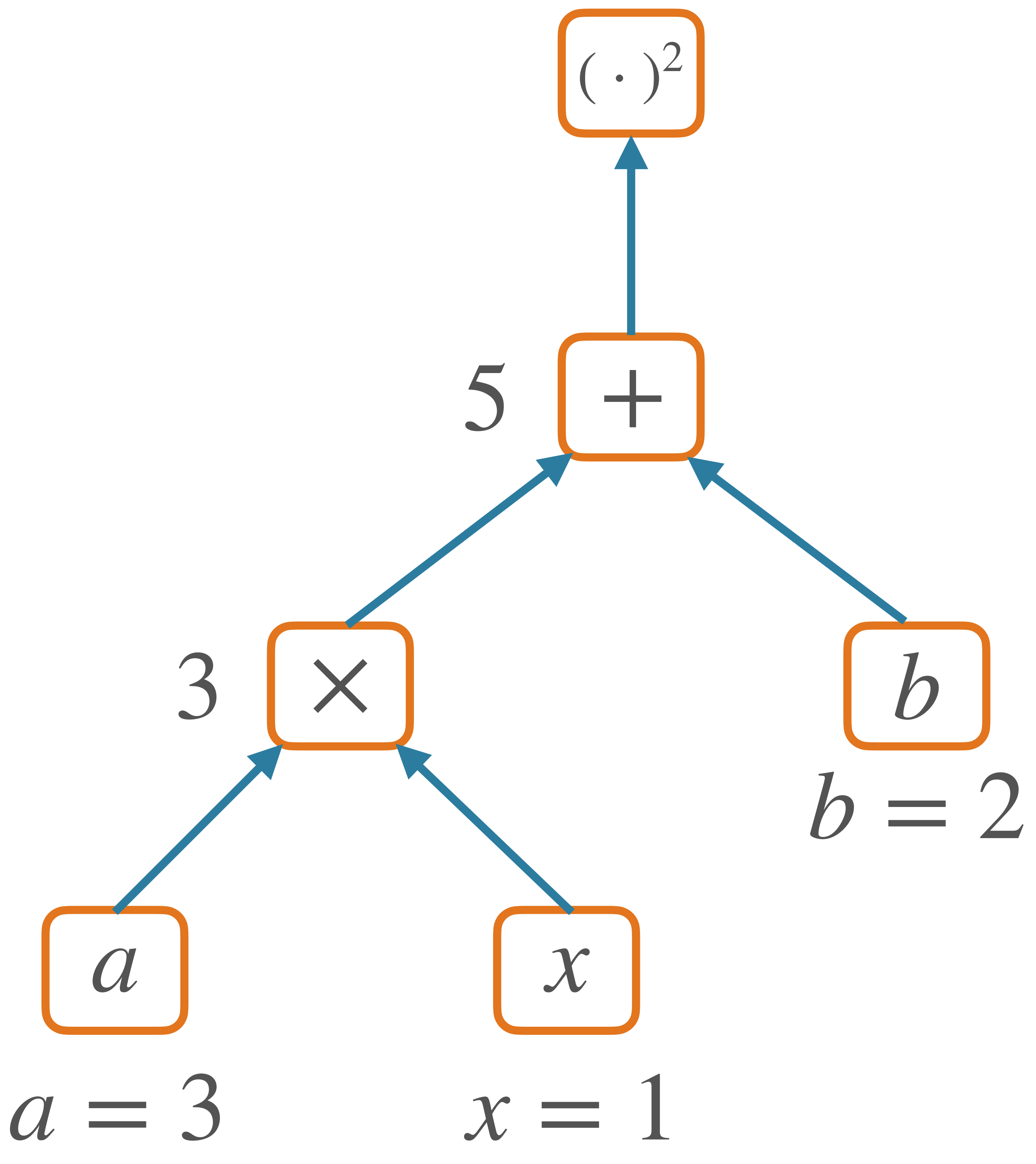
# Computation Graph Example

$$f(x; a, b) = (ax + b)^2$$



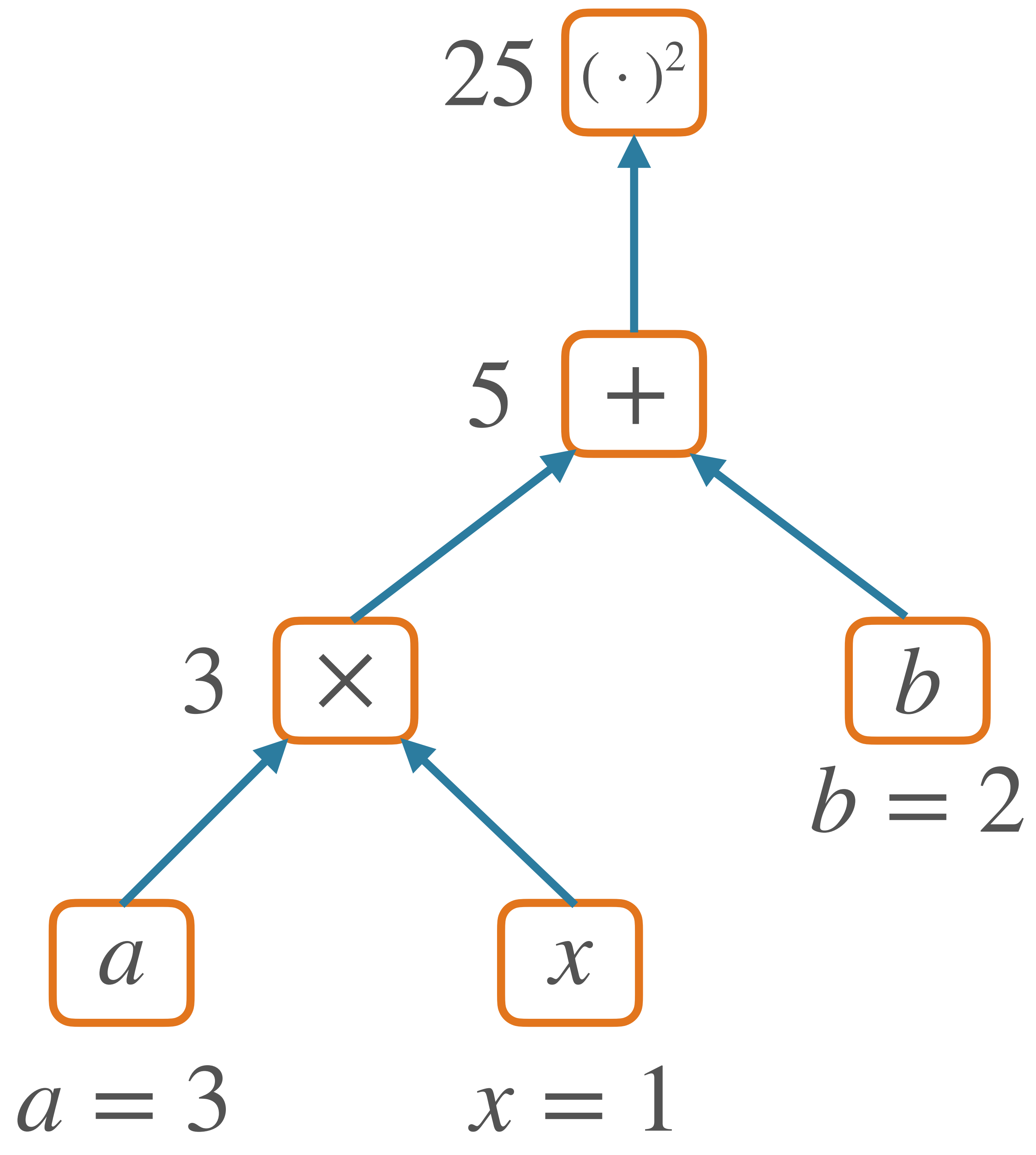
# Computation Graph Example

$$f(x; a, b) = (ax + b)^2$$



# Computation Graph Example

$$f(x; a, b) = (ax + b)^2$$

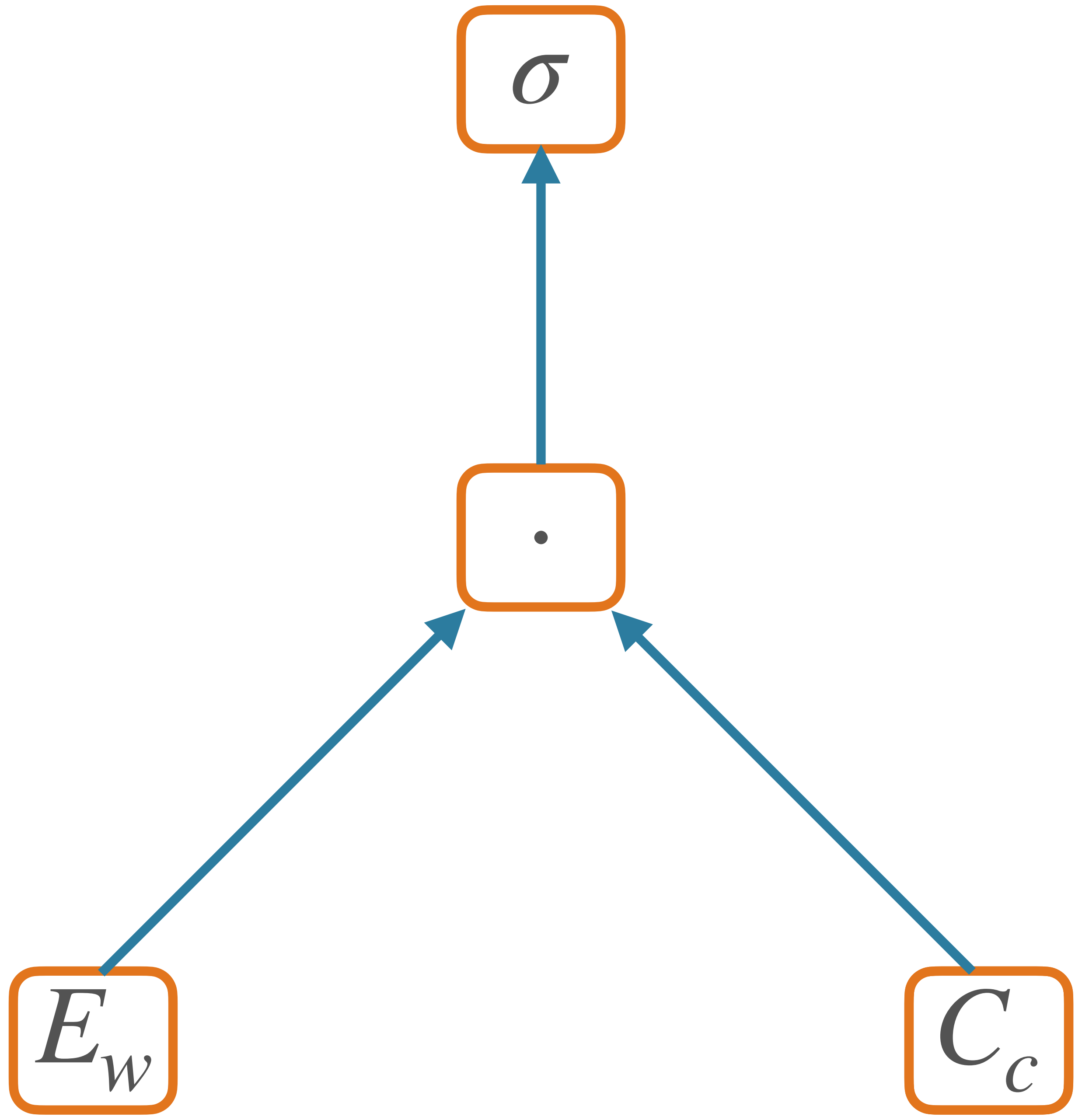


# Nodes in a Graph

- Node: a Tensor value
  - e.g. numpy ndarray; n-dimensional array of values
  - Scalar, vector, matrix, …
- Edge: function argument
  - The value of a node is a function of the values of its parents
- For **forward**: node computes its value based on its parents' values

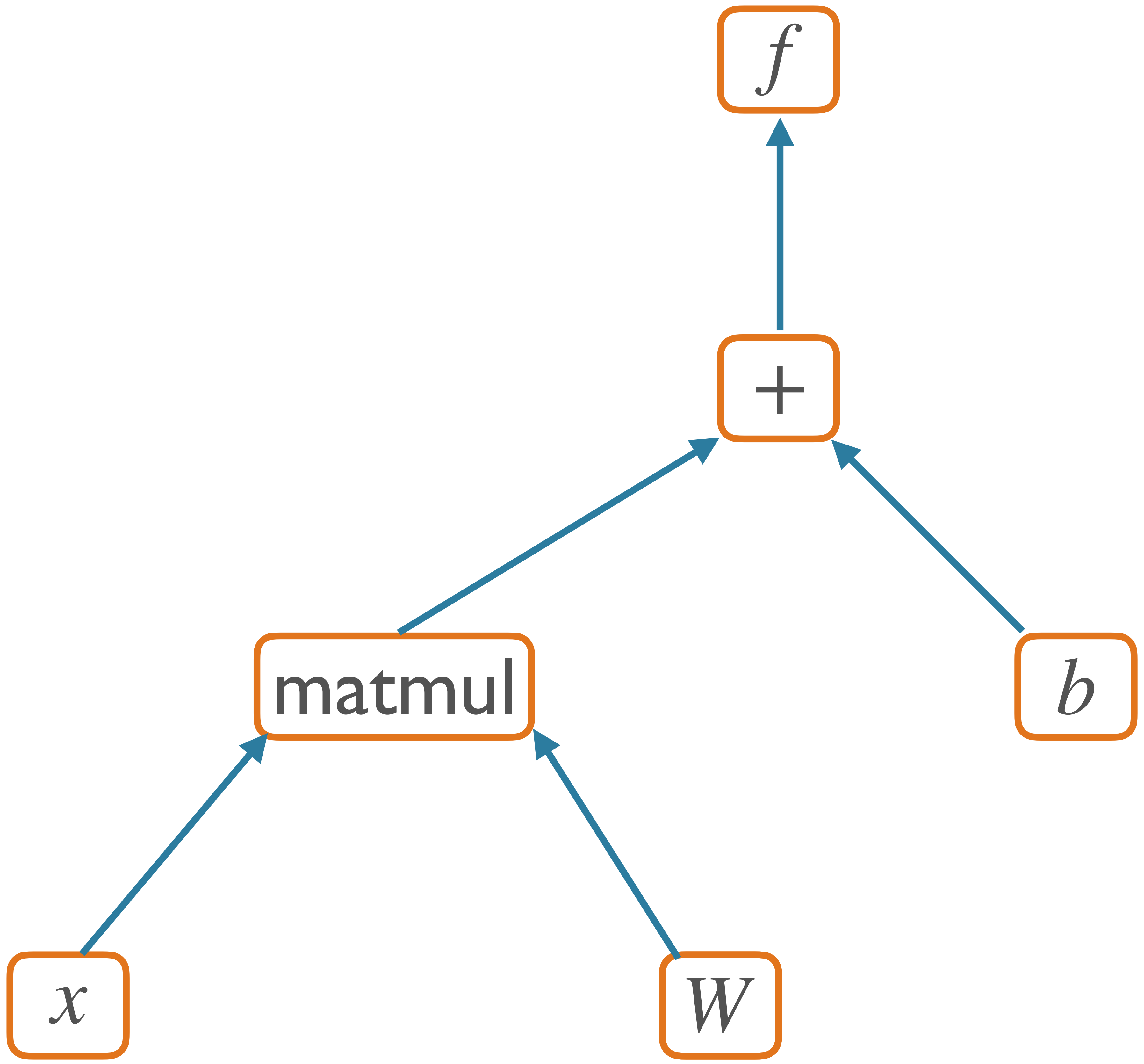
# SGNS as a Graph

$$P(1 \mid w, c) = \sigma\left(E_w \cdot C_c\right)$$



# Hidden Layer Graph

$$\hat{y} = f(xW + b)$$



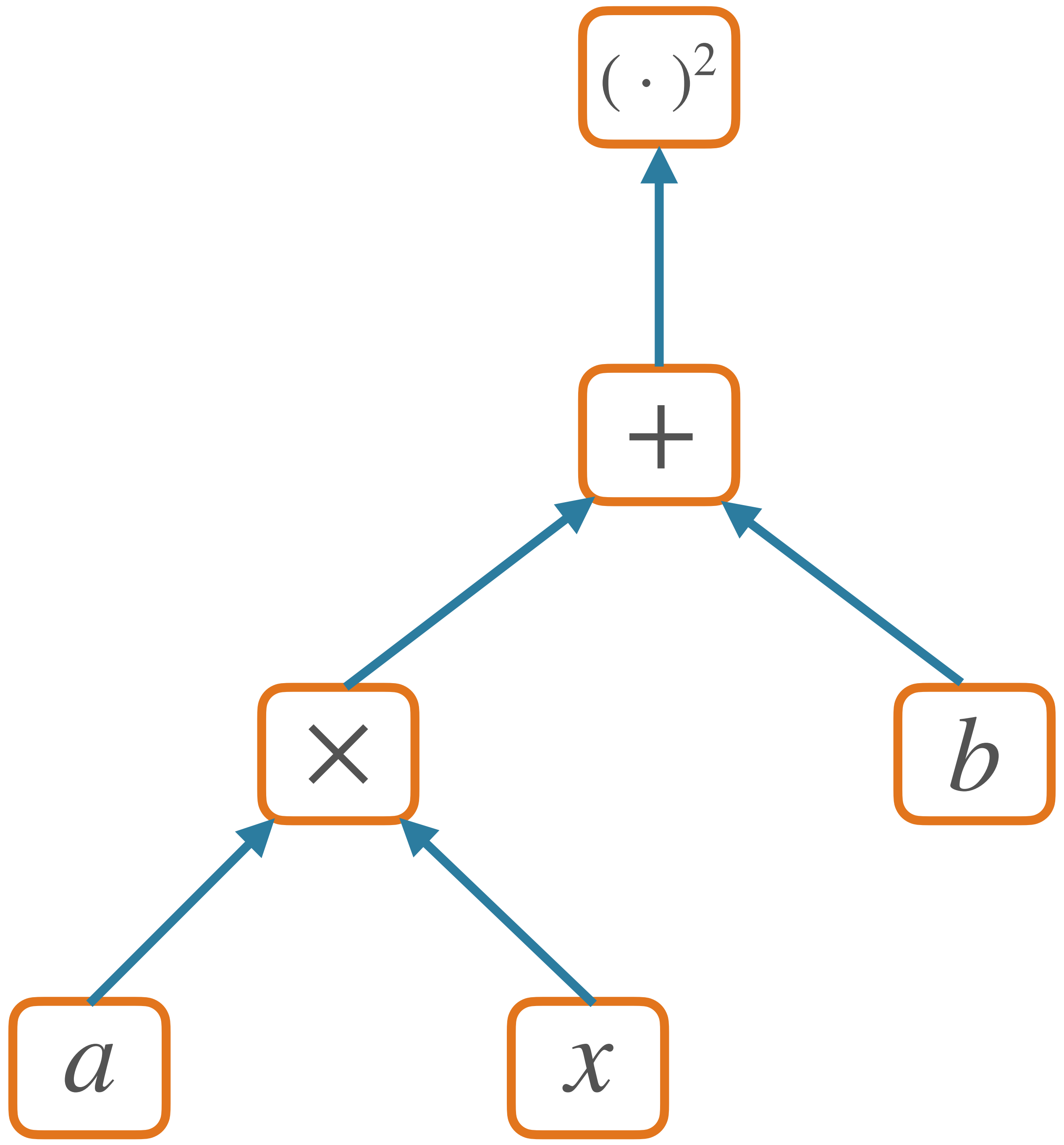
# Backpropagation

# So what?

- So far, this is just fancy re-writing of basic mathematical computation
- The real victory of the graph abstraction comes in computing *derivatives*
- Backpropagation:
  - A dynamic programming algorithm on computation graphs that allows the gradient of an output to be computed with respect to *every node* in the graph

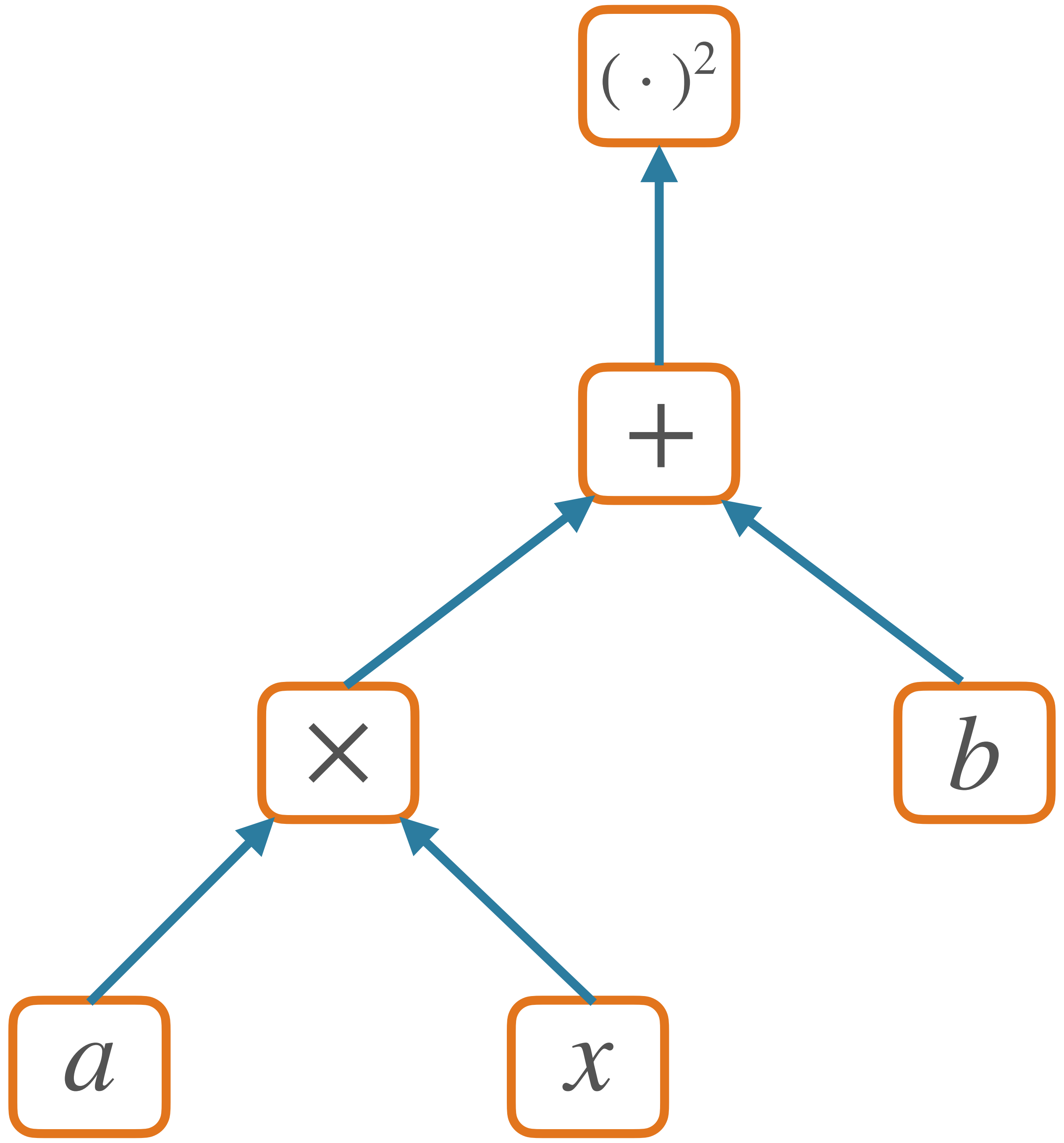
# Computing Derivatives

$$f(x; a, b) = (ax + b)^2$$



# Computing Derivatives

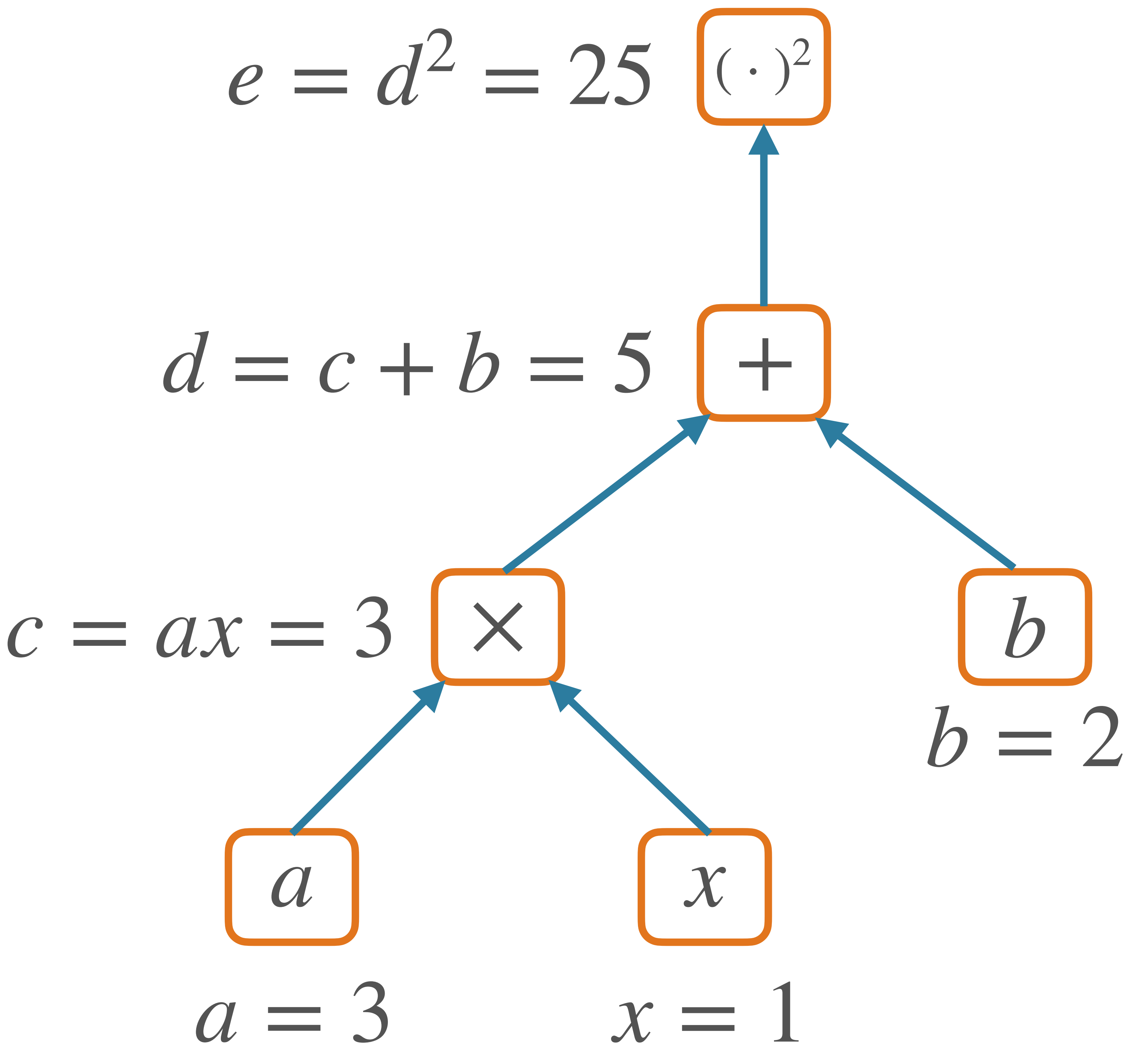
$$f(x; a, b) = (ax + b)^2$$




$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial(ax+b)}\frac{\partial(ax+b)}{\partial x}$$
$$= 2(ax+b)a$$
$$\frac{\partial f}{\partial a} = 2(ax+b)x$$
$$\frac{\partial f}{\partial b} = 2(ax+b)$$

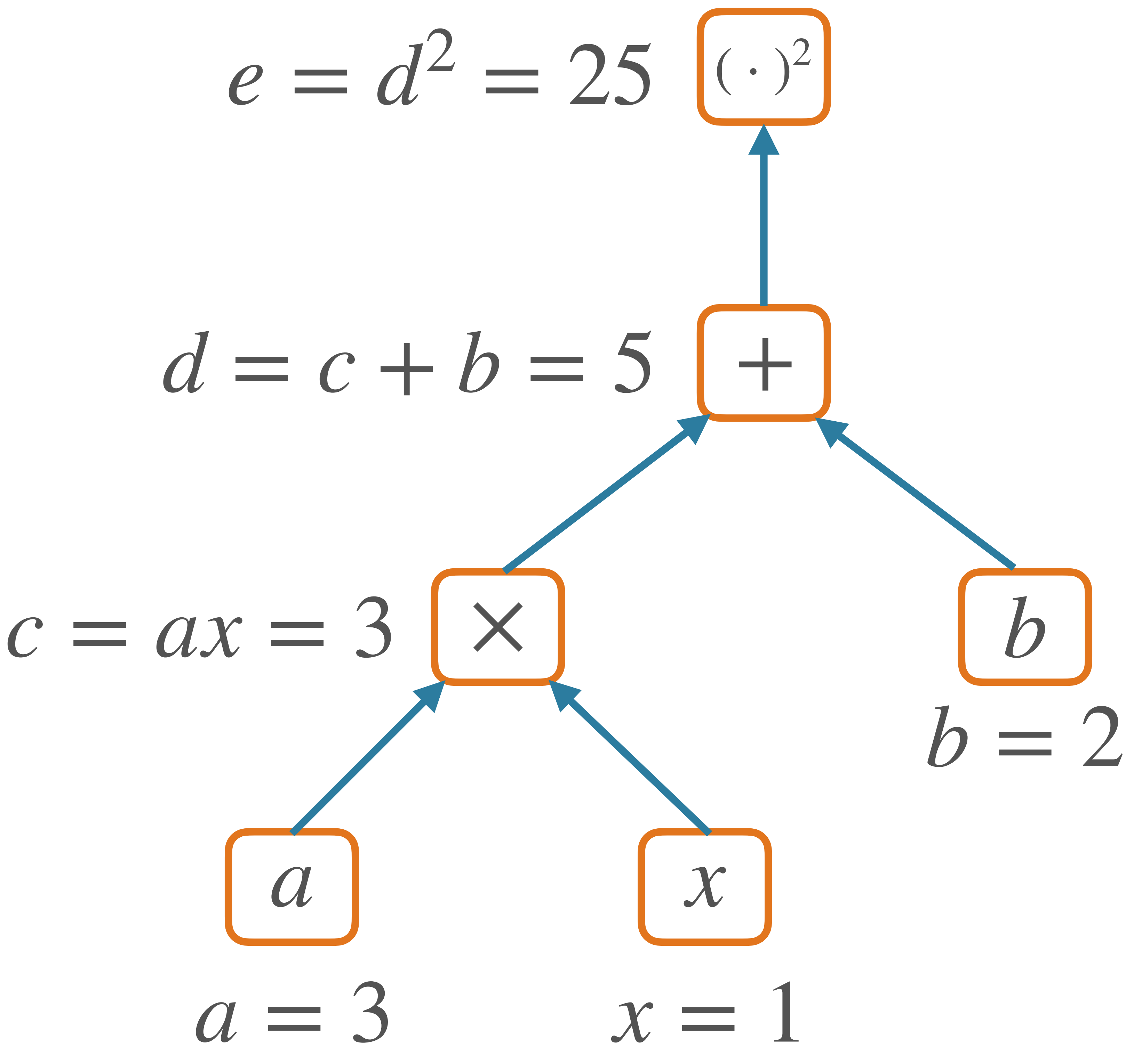
# Backpropagation Example

$$f(x; a, b) = (ax + b)^2$$



# Backpropagation Example

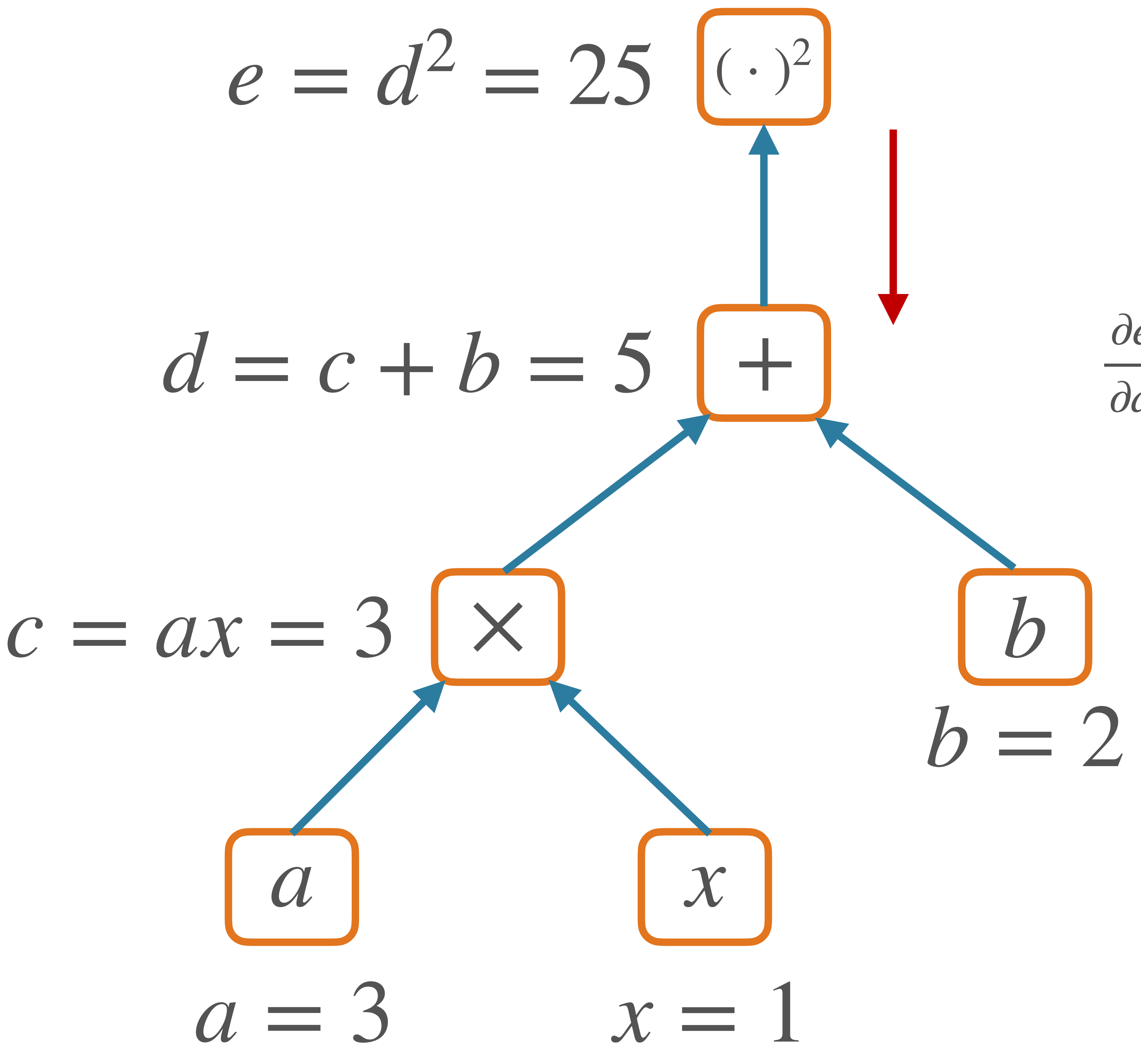
$$f(x; a, b) = (ax + b)^2$$




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

# Backpropagation Example

$$f(x; a, b) = (ax + b)^2$$

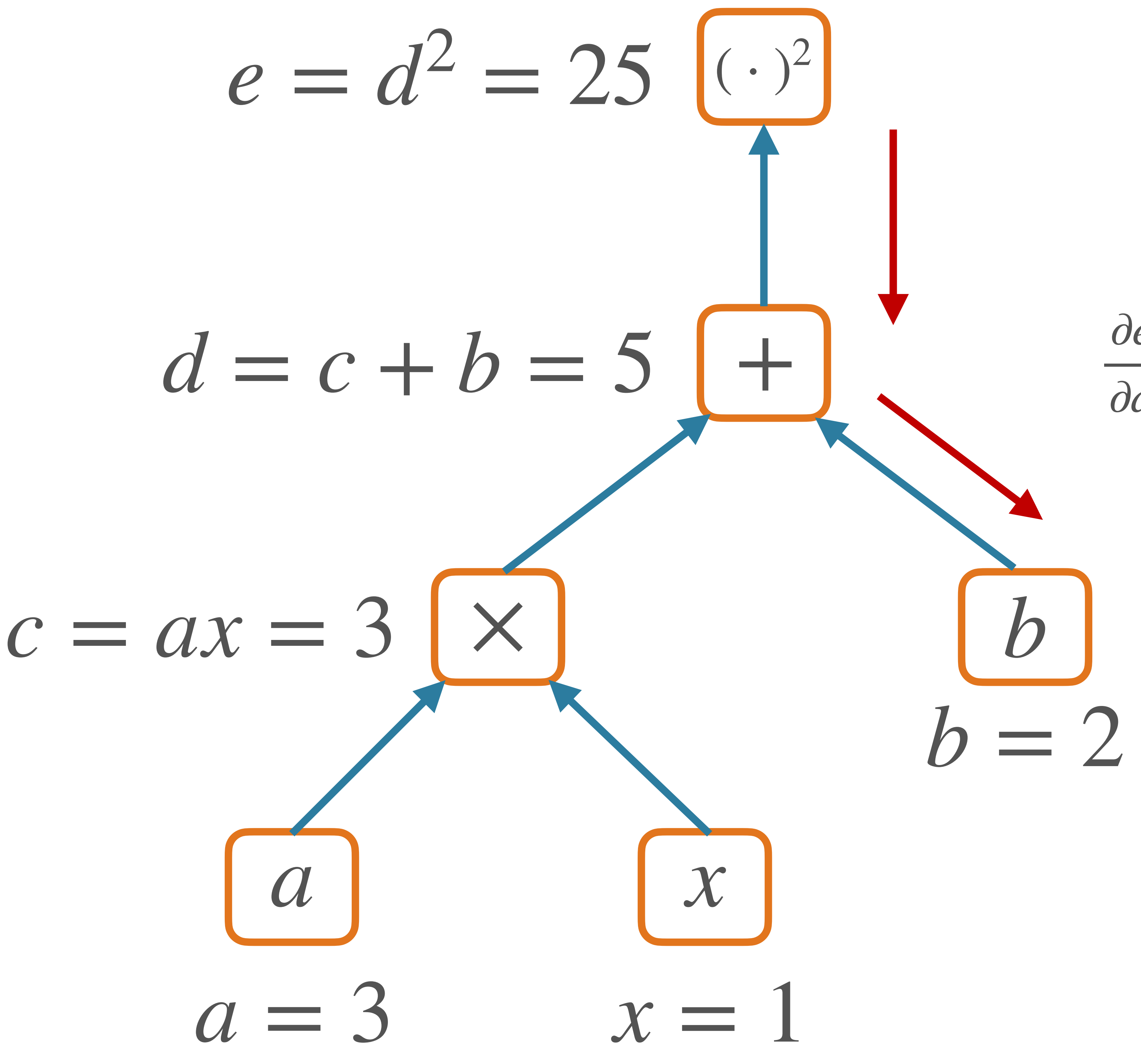



$$\frac{\partial e}{\partial e} = 1$$

$$\frac{\partial e}{\partial d} = 2d\frac{\partial e}{\partial e} = 10$$

# Backpropagation Example

$$f(x; a, b) = (ax + b)^2$$



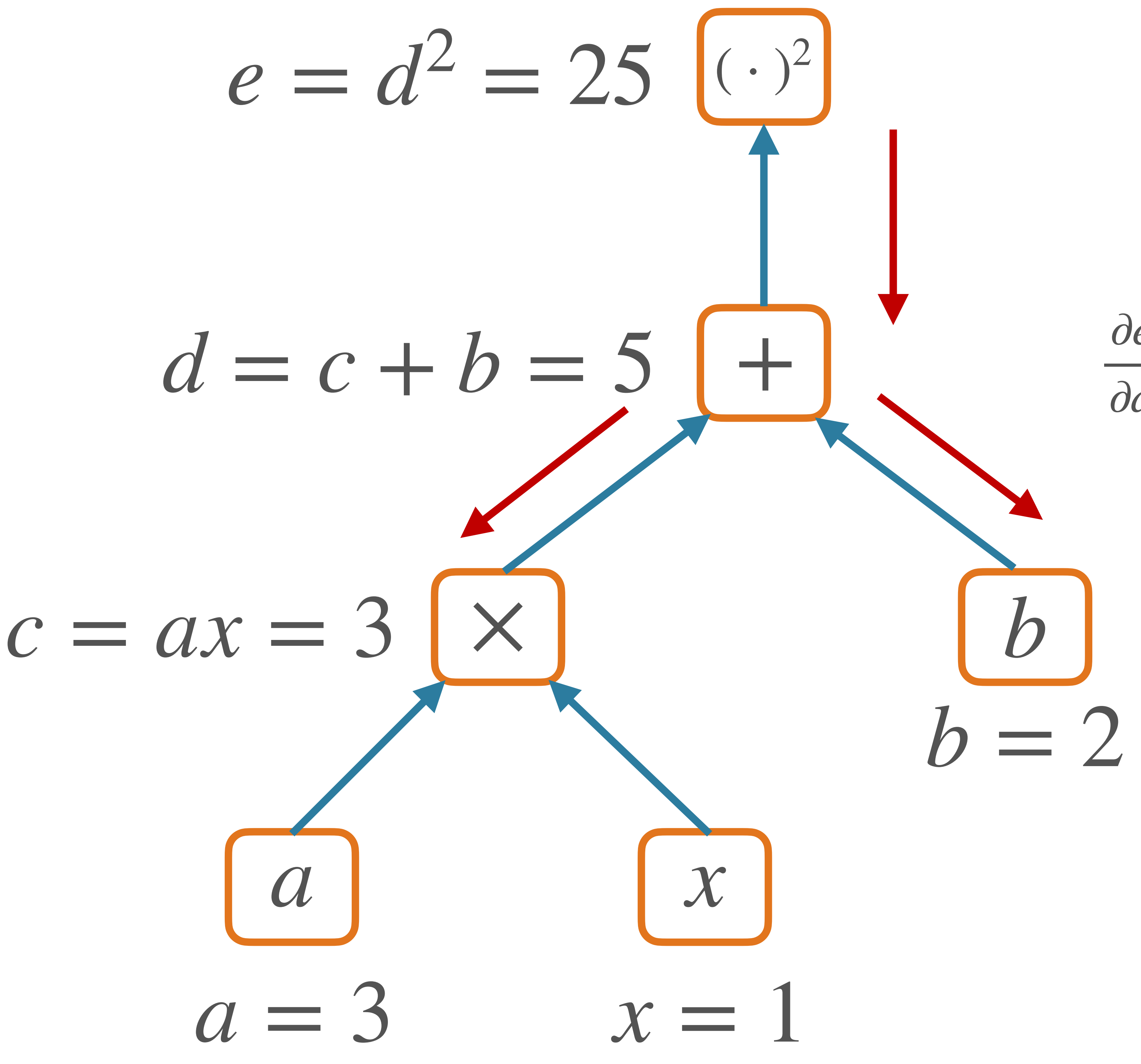

$$\frac{\partial e}{\partial e} = 1$$

$$\frac{\partial e}{\partial d} = 2d\frac{\partial e}{\partial e} = 10$$

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial b} = 10\frac{\partial c + b}{\partial b} = 10$$

# Backpropagation Example

$$f(x; a, b) = (ax + b)^2$$




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

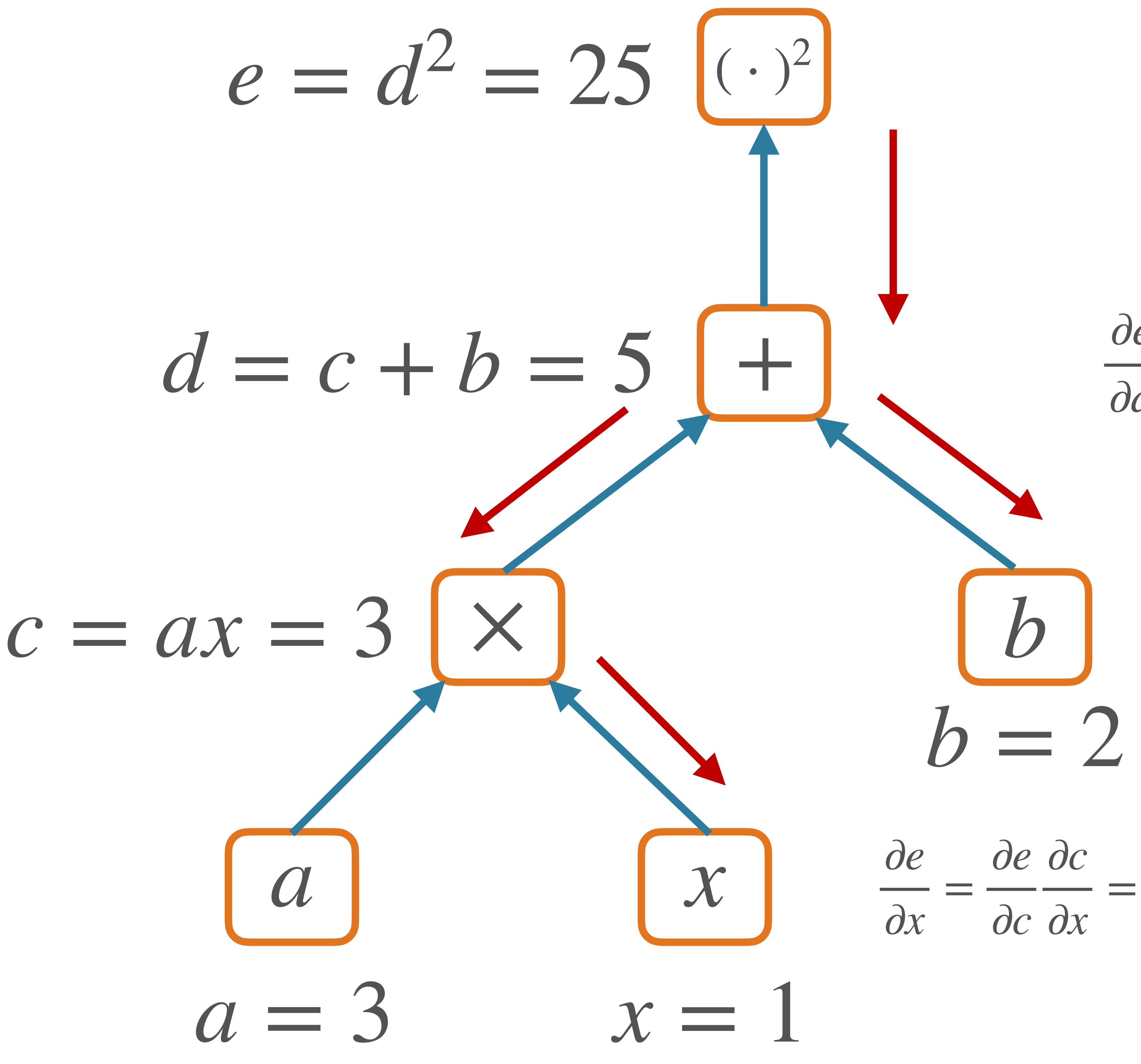
$$\frac{\partial e}{\partial d} = 2d\frac{\partial e}{\partial e} = 10$$

$$\frac{\partial e}{\partial c} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial c} = 10\frac{\partial c + b}{\partial c} = 10$$

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial b} = 10\frac{\partial c + b}{\partial b} = 10$$

# Backpropagation Example

$$f(x; a, b) = (ax + b)^2$$




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

$$\frac{\partial e}{\partial d} = 2d\frac{\partial e}{\partial e} = 10$$

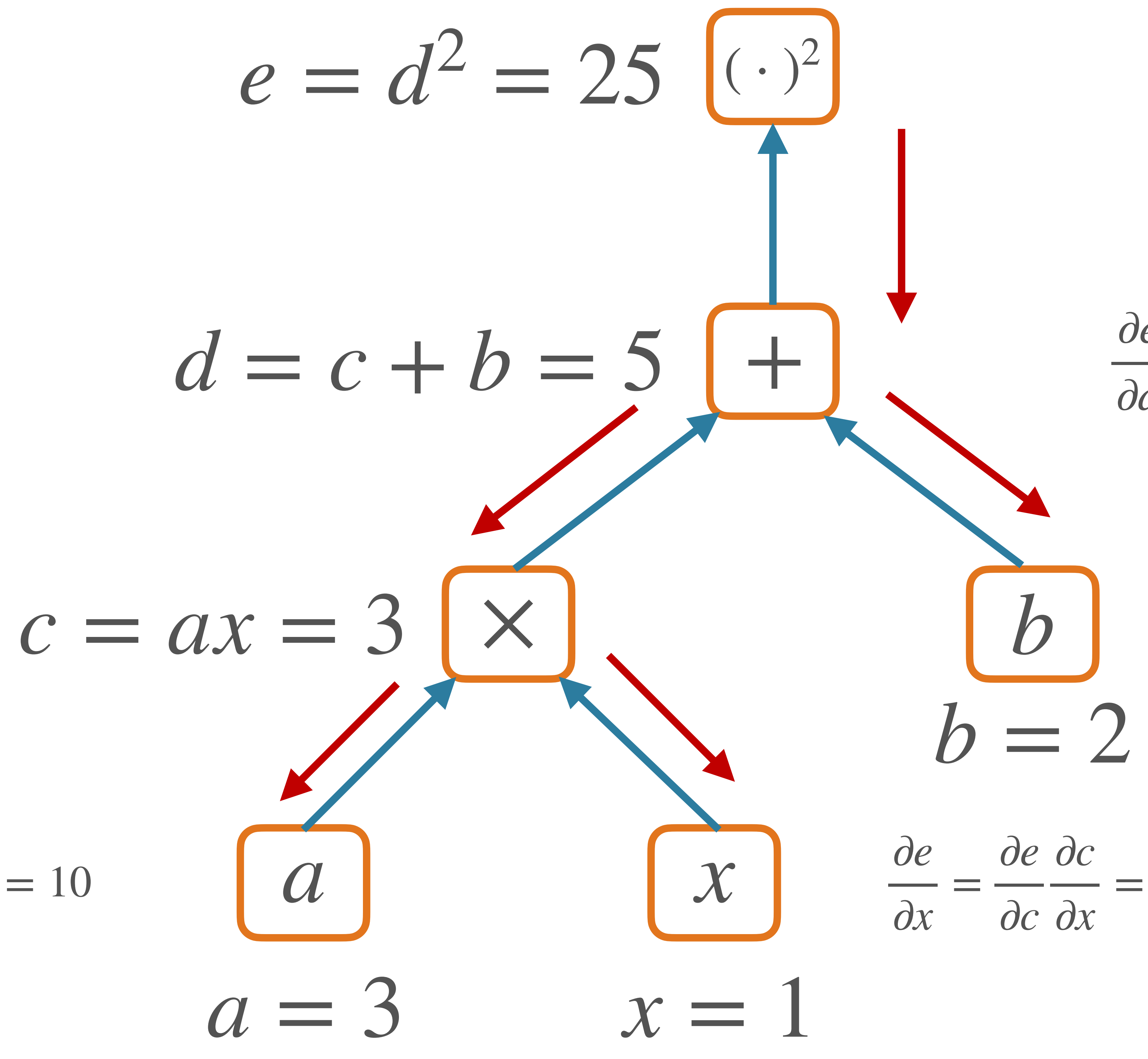
$$\frac{\partial e}{\partial c} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial c} = 10\frac{\partial c + b}{\partial c} = 10$$

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial b} = 10\frac{\partial c + b}{\partial b} = 10$$

$$\frac{\partial e}{\partial x} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial x} = 10a = 30$$

# Backpropagation Example

$$f(x; a, b) = (ax + b)^2$$




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

$$\frac{\partial e}{\partial d} = 2d\frac{\partial e}{\partial e} = 10$$

$$\frac{\partial e}{\partial c} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial c} = 10\frac{\partial c + b}{\partial c} = 10$$

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial b} = 10\frac{\partial c + b}{\partial b} = 10$$

$$\frac{\partial e}{\partial a} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial a} = 10x = 10$$

$$\frac{\partial e}{\partial x} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial x} = 10a = 30$$

# Backpropagation

- Initialize gradient to 1 for given output node $f$
  - [NB: assuming that this output node is a *scalar*]
- Loop over nodes in graph in *reversed topological order* [i.e. children come before parents]
  - Compute gradient of output node w/r/t this node, in terms of gradients w/r/t this node's children
    - [i.e. apply the chain rule!]

# Backpropagation Algorithm

```
def backward(self) -> None:
    """Run backward pass from a scalar tensor.

    All Tensors in the graph above this one will wind up having their
    gradients stored in `grad`.

    Raises:
        ValueError, if this is not a scalar.
    """
    if not np.isscalar(self.value):
        raise ValueError("Can only call backward() on scalar Tensors.")
    # dL / dL = 1
    self.grad = np.ones(self.value.shape)
    # NOTE: building a graph, then sorting, is not maximally efficient
    # but the graph can be used for visualization etc
    graph = self.get_graph_above()
    reverse_topological = reversed(list(nx.topological_sort(graph)))
    for tensor in reverse_topological:
        tensor._backward()
```

From Tensor class in [edugrad]()

# Backpropagation Algorithm

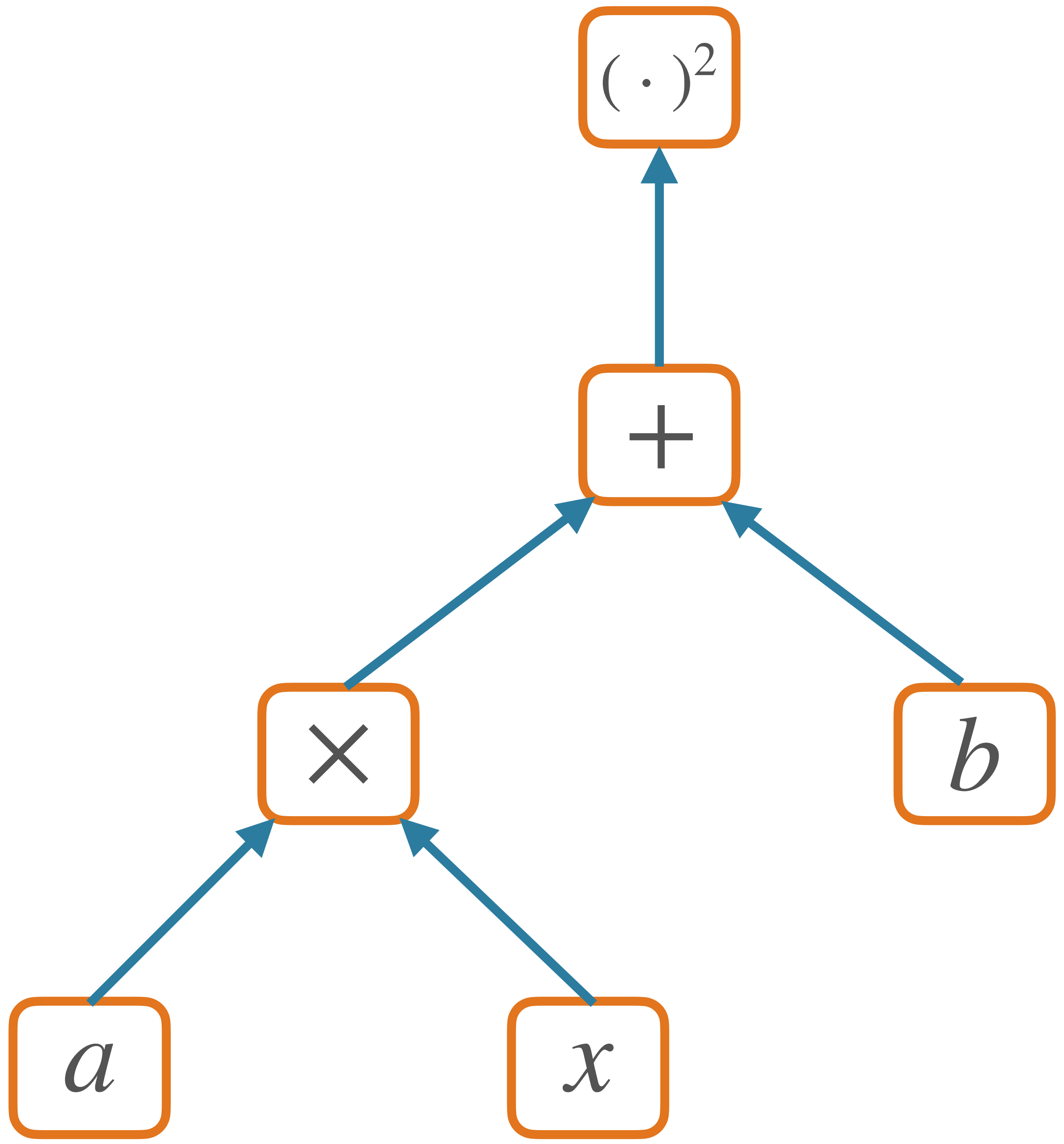
```
def backward(self) -> None:
    """Run backward pass from a scalar tensor.

    All Tensors in the graph above this one will wind up having their
    gradients stored in `grad`.

    Raises:
        ValueError, if this is not a scalar.
    """
    if not np.isscalar(self.value):
        raise ValueError("Can only call backward() on scalar Tensors.")
    # dL / dL = 1
    self.grad = np.ones(self.value.shape)
    # NOTE: building a graph, then sorting, is not maximally efficient
    # but the graph can be used for visualization etc
    graph = self.get_graph_above()
    reverse_topological = reversed(list(nx.topological_sort(graph)))
    for tensor in reverse_topological:
        tensor._backward()
```

From Tensor class in edugrad

Local gradient + chain rule application

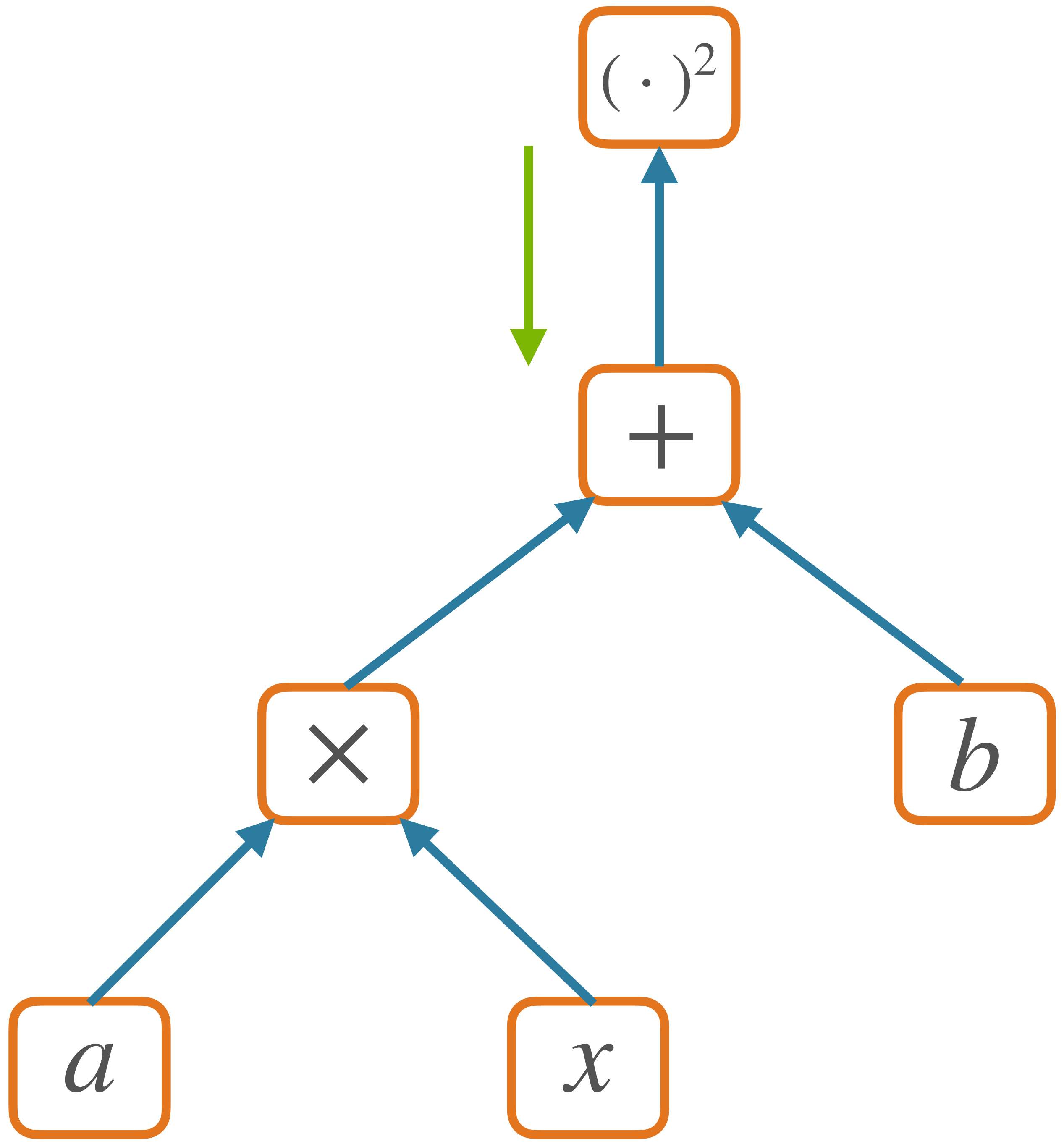
# Why back-propagation?

- Extremely efficient method for computing all gradients
  - Compute *once*
  - Store and re-use redundant computation
  - Whence a form of dynamic programming
- Traverse each edge once, instead of once per dependency path



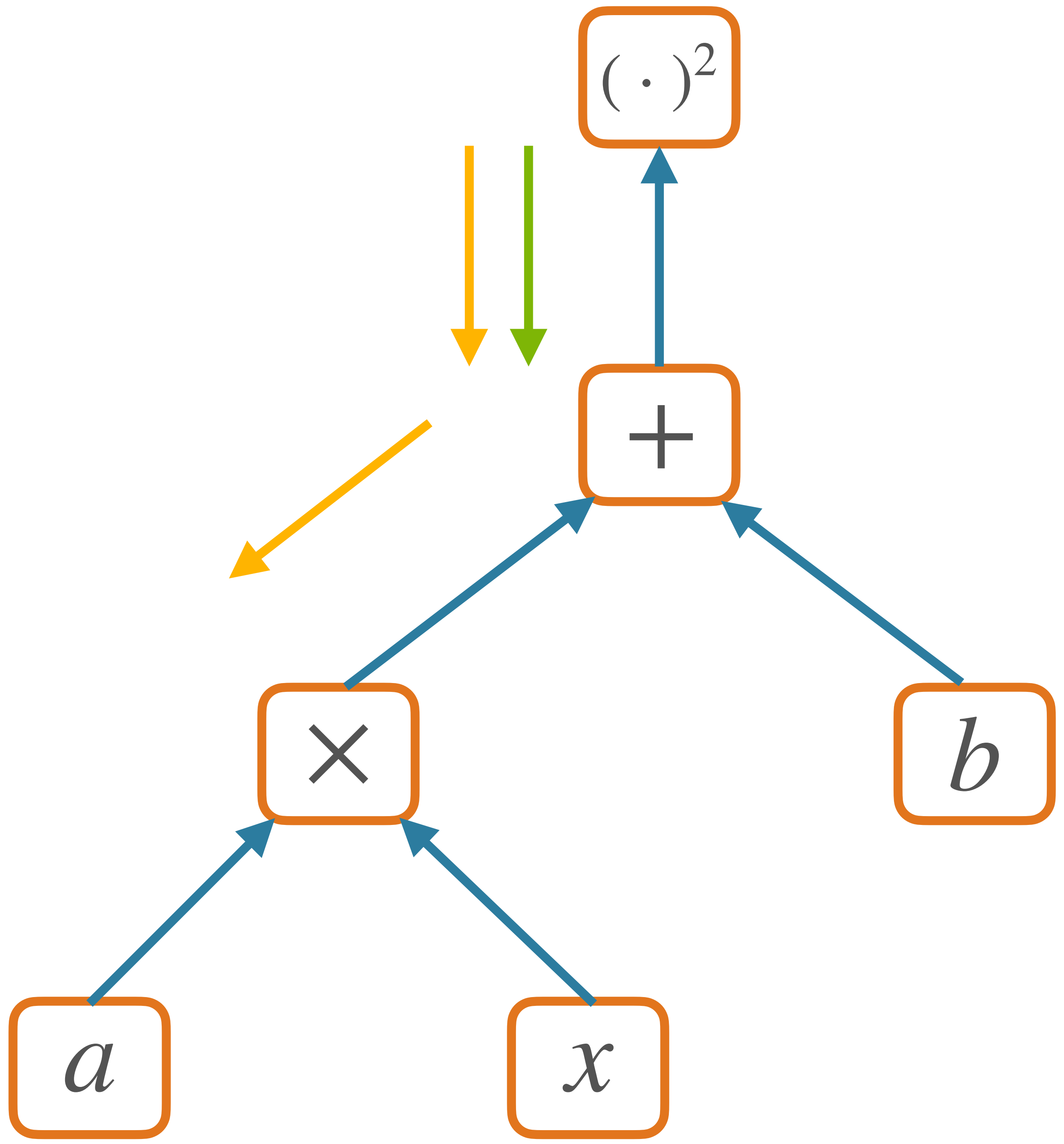
# Why back-propagation?

- Extremely efficient method for computing all gradients
  - Compute *once*
  - Store and re-use redundant computation
  - Whence a form of dynamic programming
- Traverse each edge once, instead of once per dependency path



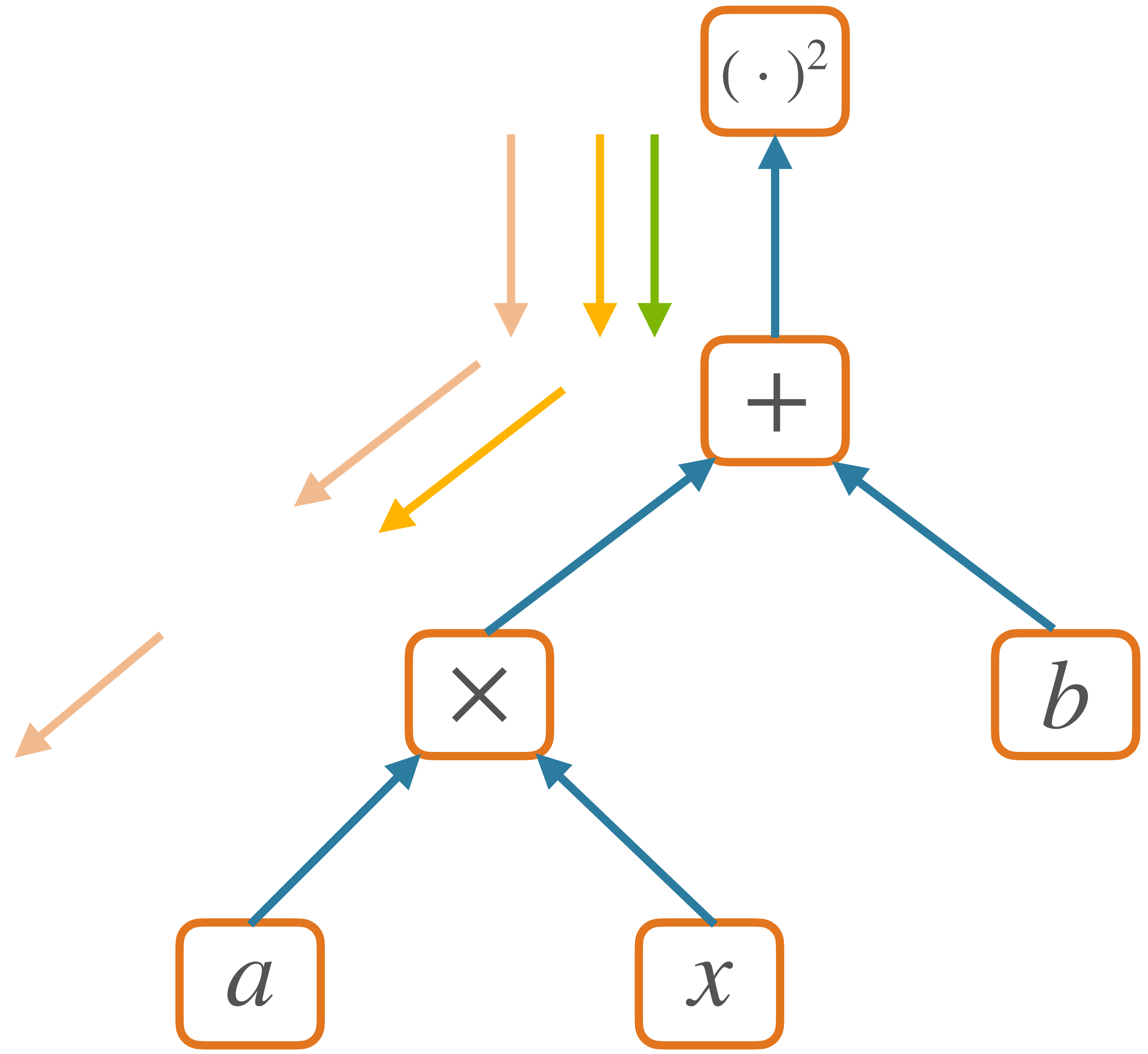
# Why back-propagation?

- Extremely efficient method for computing all gradients
  - Compute *once*
  - Store and re-use redundant computation
  - Whence a form of dynamic programming
- Traverse each edge once, instead of once per dependency path



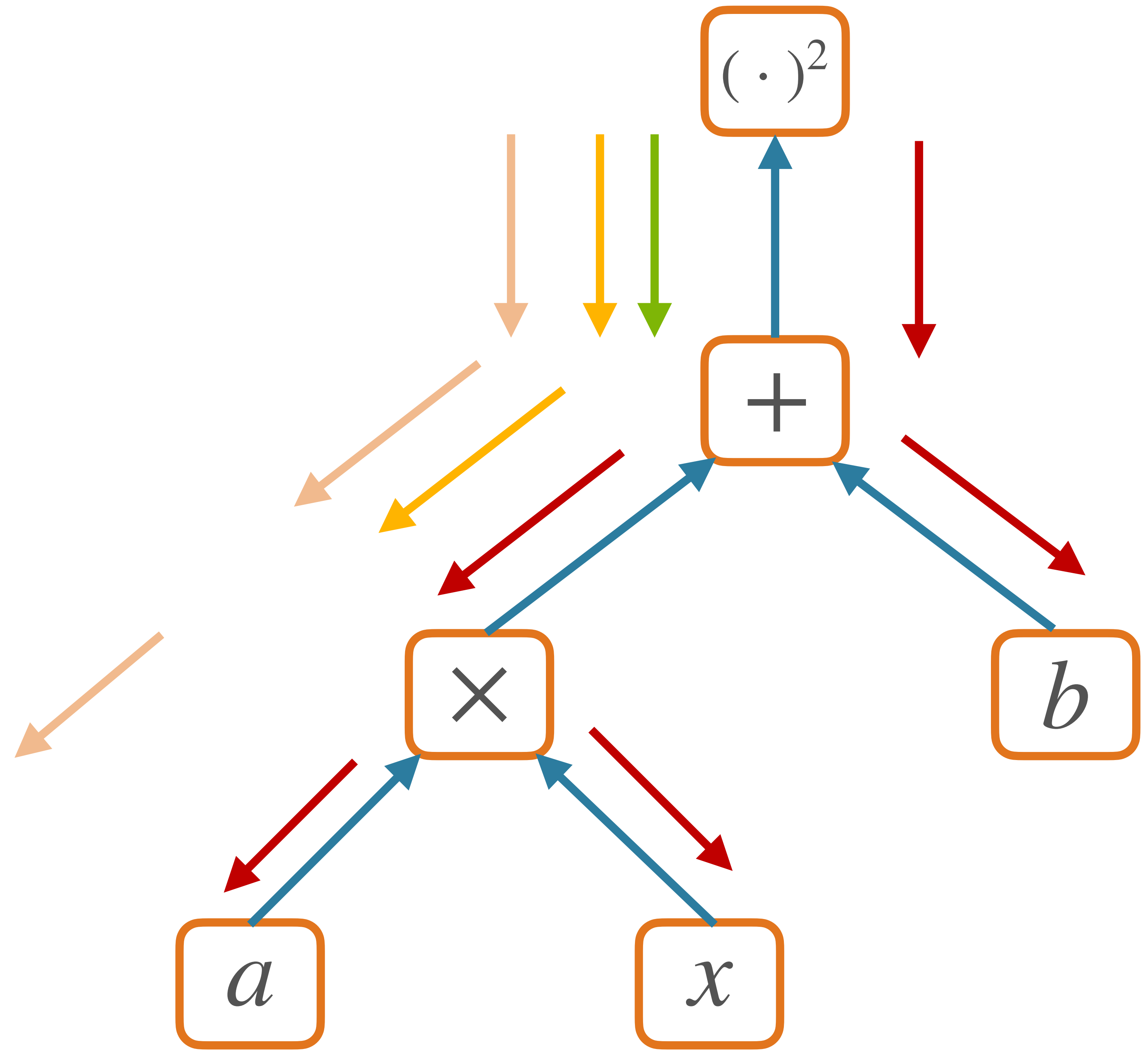
# Why back-propagation?

- Extremely efficient method for computing all gradients
  - Compute *once*
  - Store and re-use redundant computation
  - Whence a form of dynamic programming
- Traverse each edge once, instead of once per dependency path



# Why back-propagation?

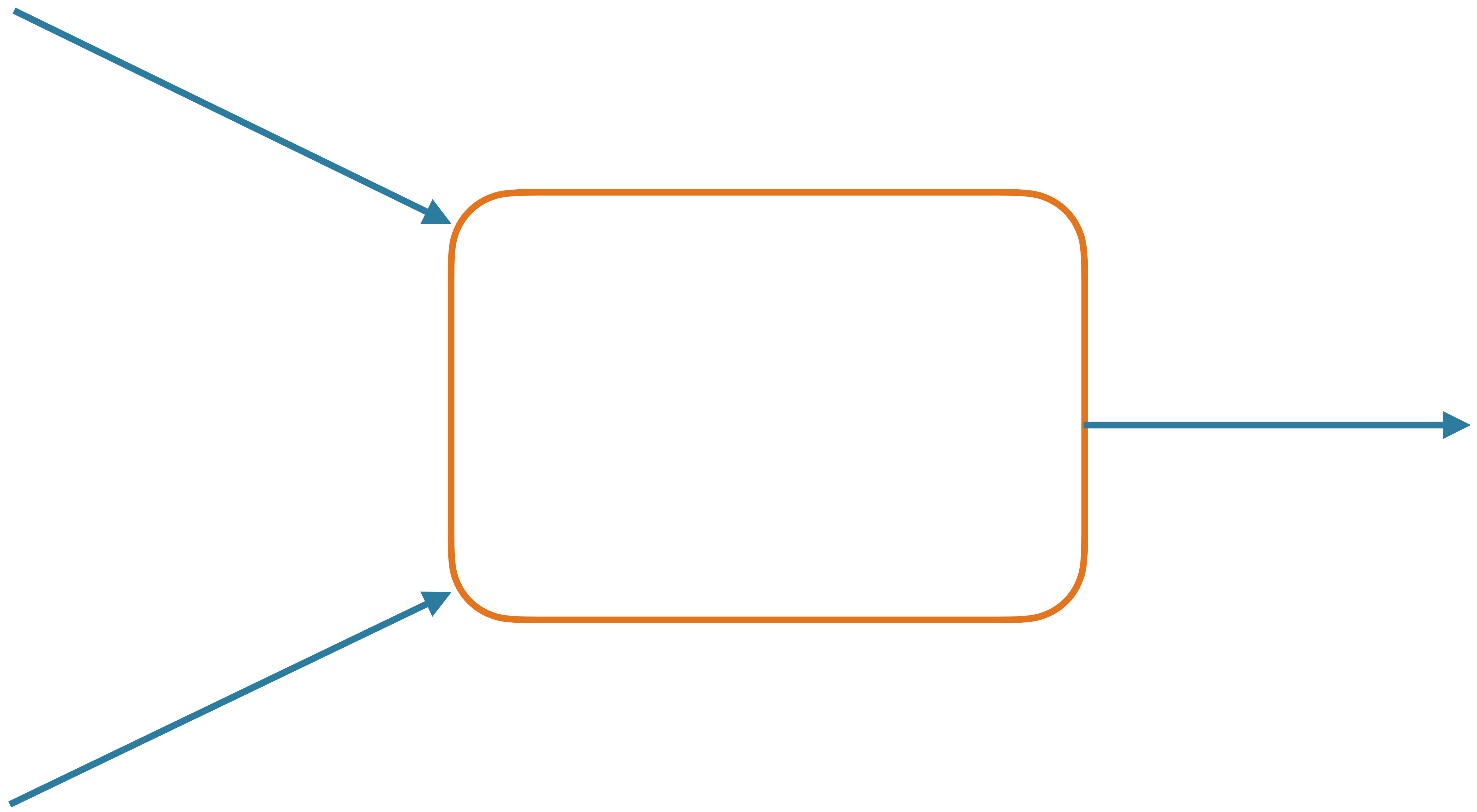
- Extremely efficient method for computing all gradients
  - Compute *once*
  - Store and re-use redundant computation
  - Whence a form of dynamic programming
- Traverse each edge once, instead of once per dependency path



# Forward/backward API

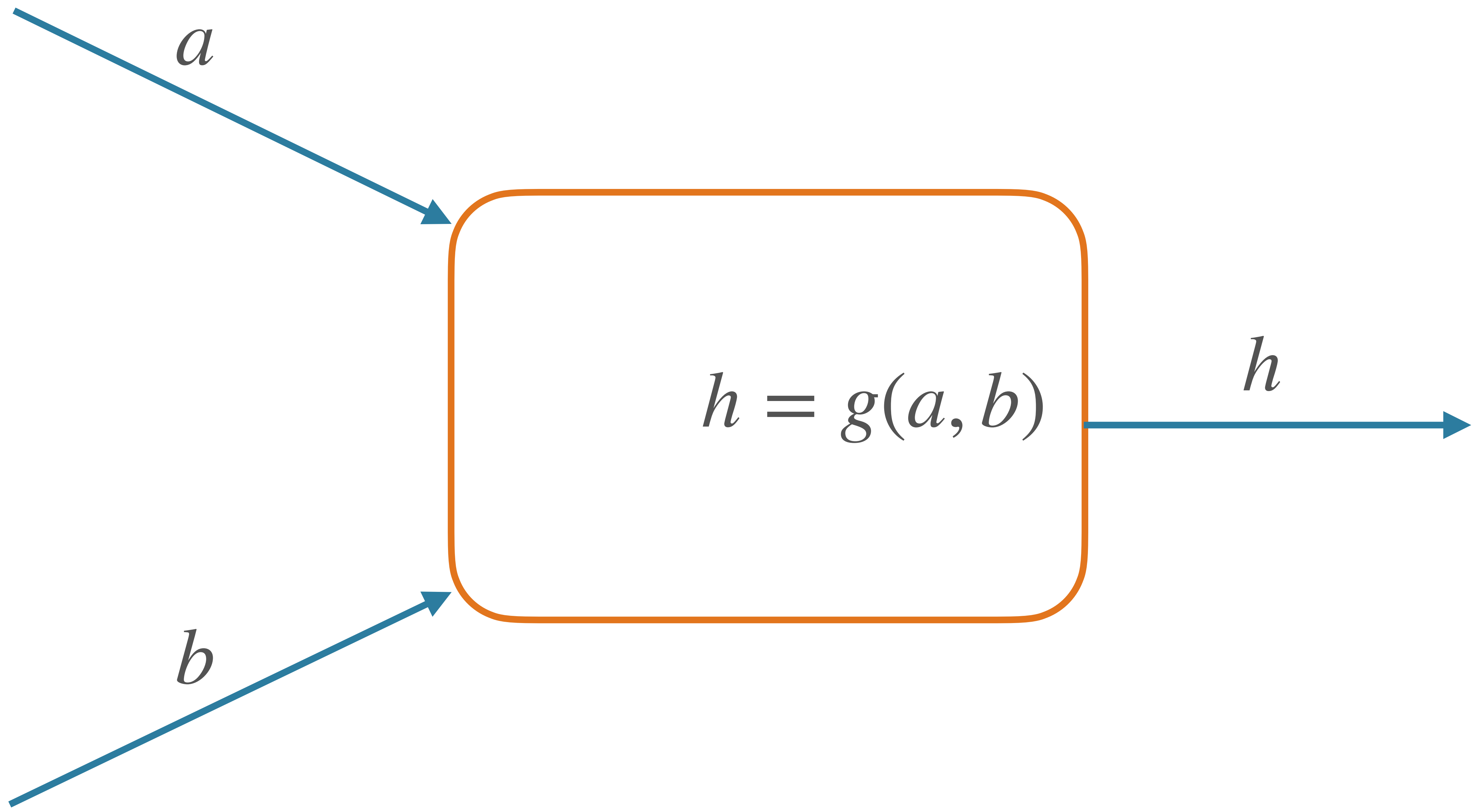
# Nodes in Computational Graph

- Forward pass:
  - Compute value given parents' values
- Backward pass:
  - Compute parents' gradients given children's



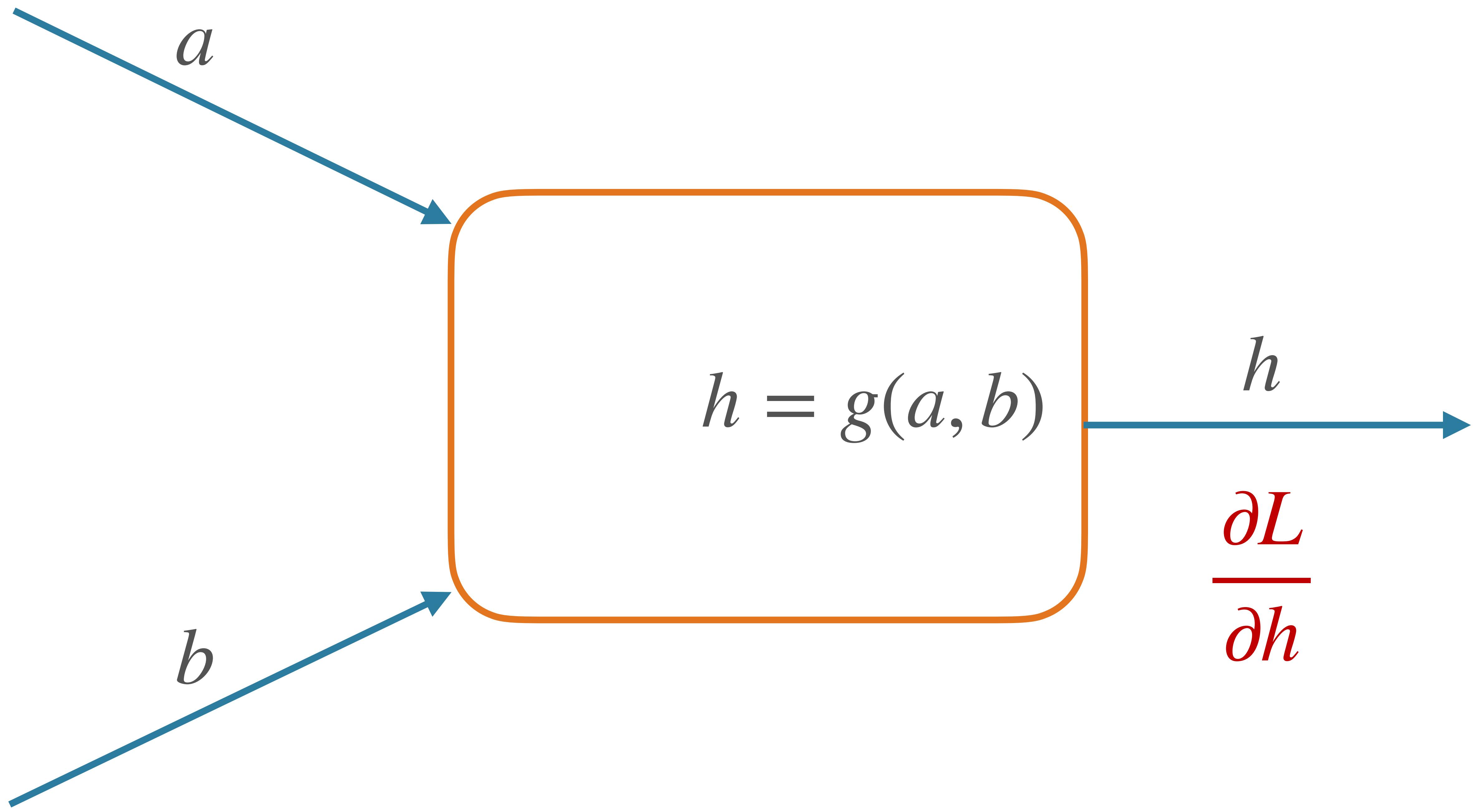
# Nodes in Computational Graph

- Forward pass:
  - Compute value given parents' values
- Backward pass:
  - Compute parents' gradients given children's



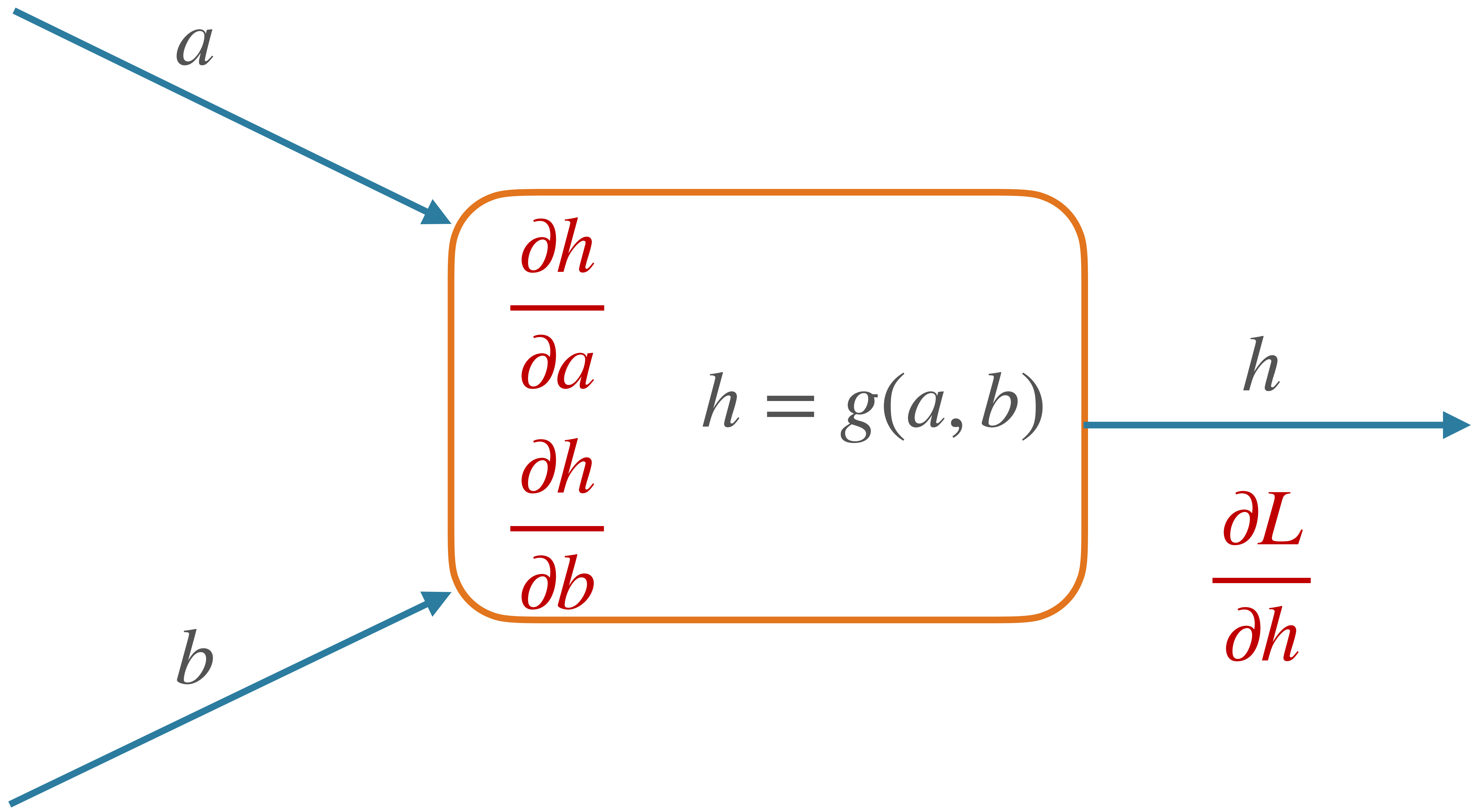
# Nodes in Computational Graph

- Forward pass:
  - Compute value given parents' values
- Backward pass:
  - Compute parents' gradients given children's



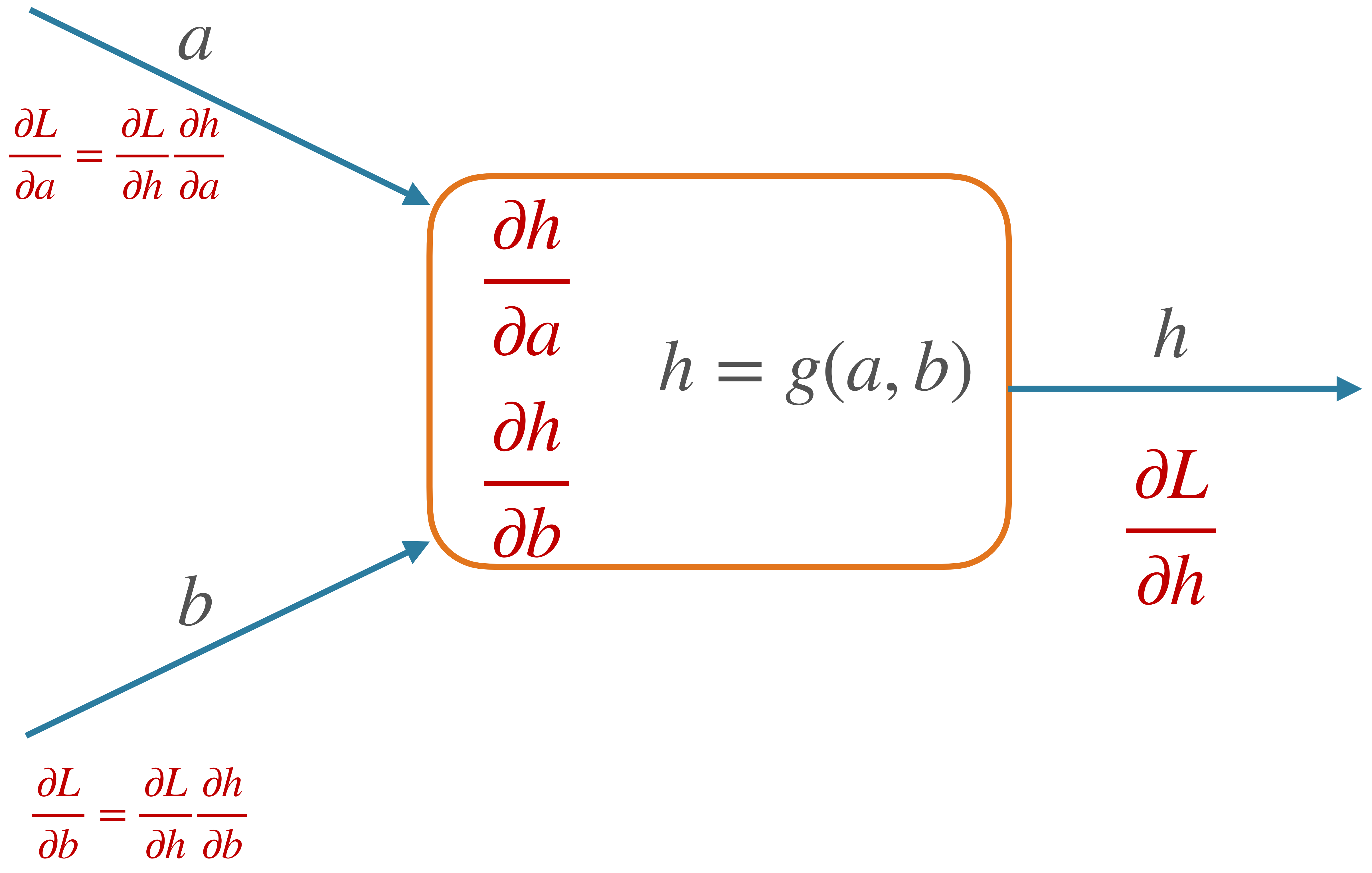
# Nodes in Computational Graph

- Forward pass:
  - Compute value given parents' values
- Backward pass:
  - Compute parents' gradients given children's



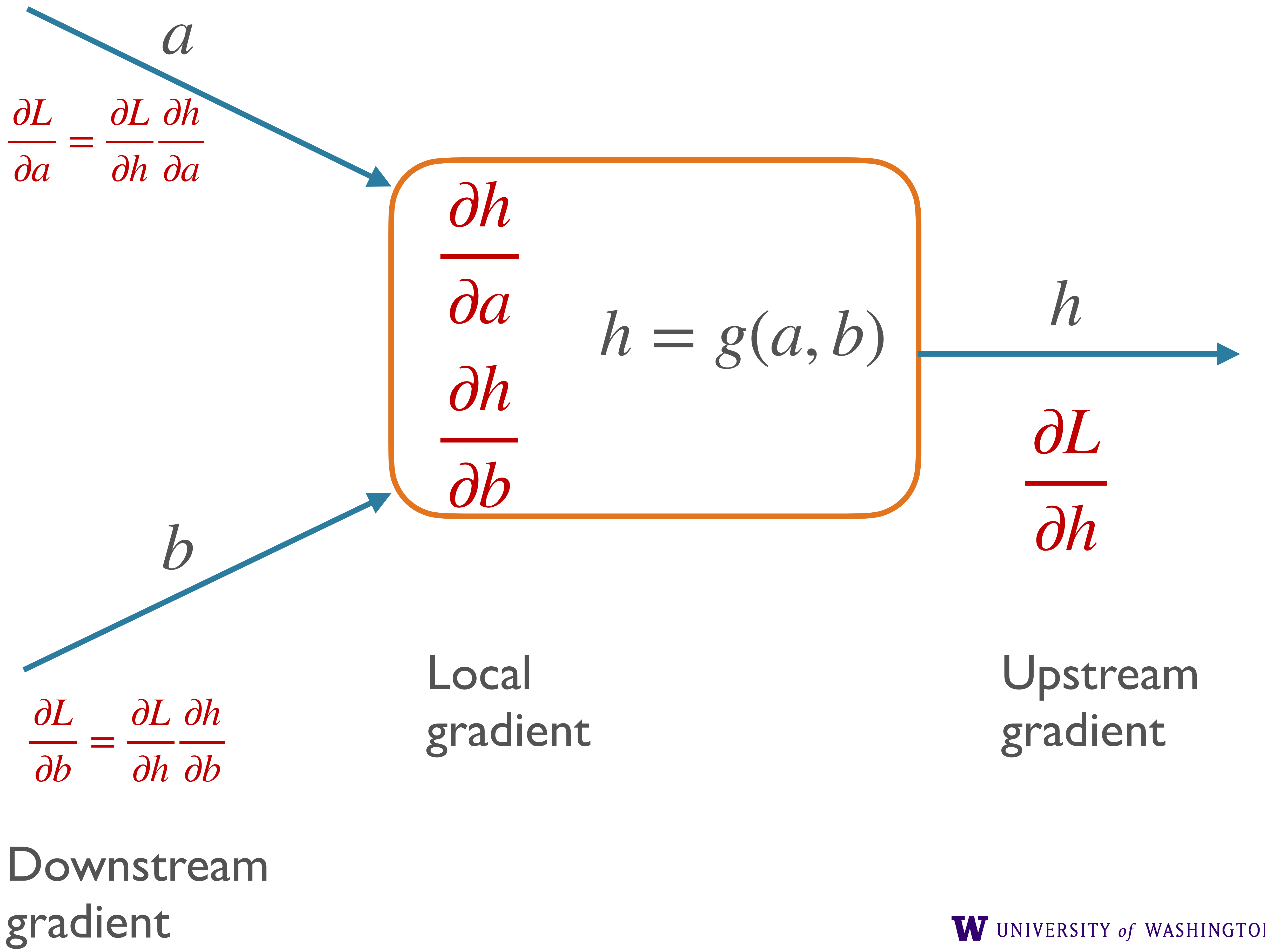
# Nodes in Computational Graph

- Forward pass:
  - Compute value given parents' values
- Backward pass:
  - Compute parents' gradients given children's



# Nodes in Computational Graph

- Forward pass:
  - Compute value given parents' values
- Backward pass:
  - Compute parents' gradients given children's



# Forward/Backward API

```python
class Operation:
    @staticmethod
    def forward(
        ctx: List[np.ndarray], *inputs: List[np.ndarray], **kwargs
    ) -> np.ndarray:
        """Forward pass of an operation.

        Args:
            ctx: empty list of arrays; can be used to store values for backward pass
            inputs: arguments to this operation

        Returns:
            output of the operation, assumed to be one numpy array
        """
        raise NotImplementedError

    @staticmethod
    def backward(ctx: List[np.ndarray], grad_output: np.ndarray) -> List[np.ndarray]:
        """Backward pass of an op, returns dL / dx for each x in parents of this op.

        Args:
            ctx: stored values from the forward pass
            grad_output: dL/dv, where v is output of this node

        Returns:
            a _list_ of arrays, dL/dx, for each x that was input to this op
        """
        raise NotImplementedError
```

From my edugrad mini-library, which you will use :)

# Example: Addition

```
@tensor_op
class add(Operation):
    @staticmethod
    def forward(ctx, a, b):
        return a + b

    @staticmethod
    def backward(ctx, grad_output):
        return grad_output, grad_output
```

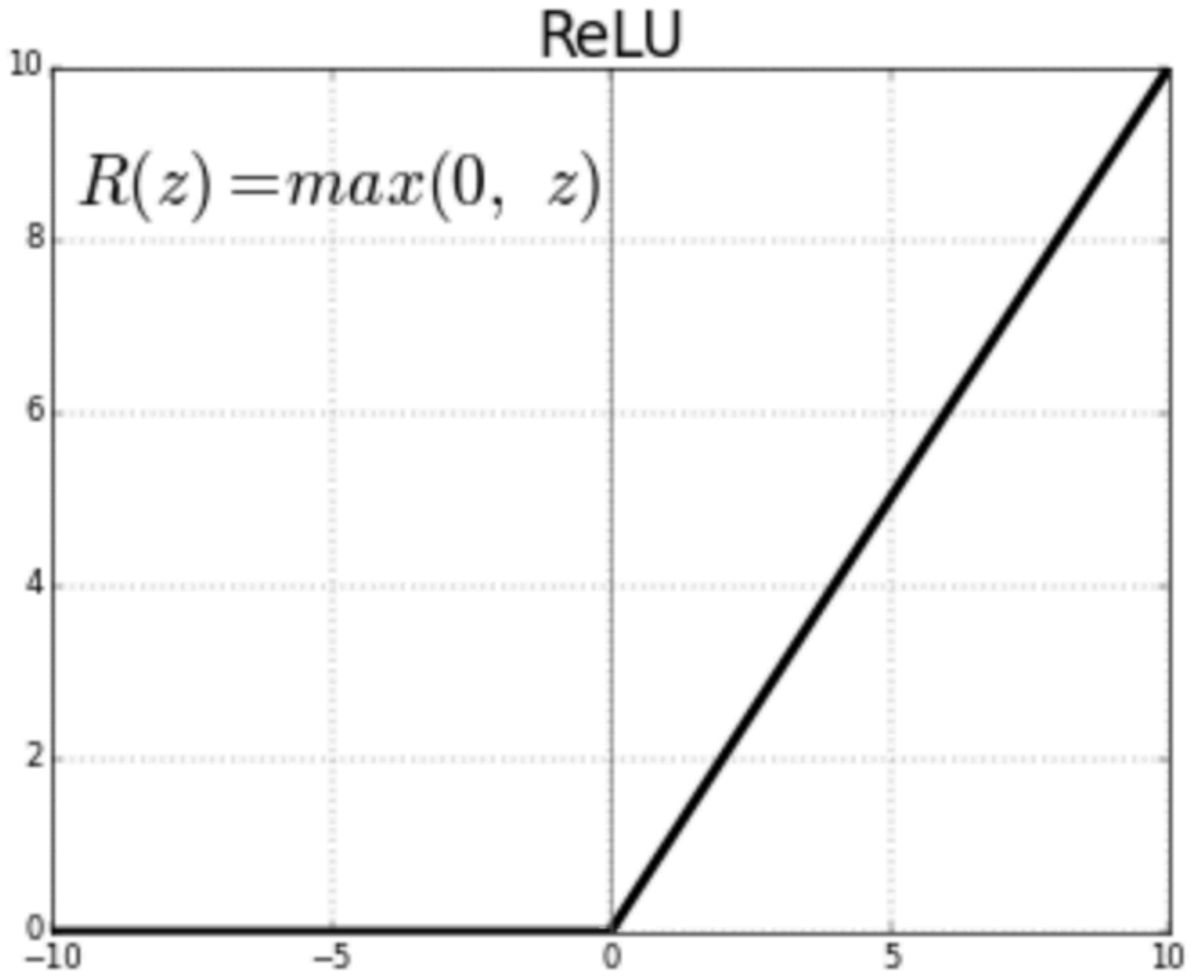
$$\frac{\partial L}{\partial a} \qquad \frac{\partial L}{\partial b}$$

# Example: ReLU

```
class relu(Operation):
 def forward(ctx, x):
  return np.maximum(0, x)

 def backward(ctx, grad_output):
```

$\text{ReLU}(x) = \max(0,x)$

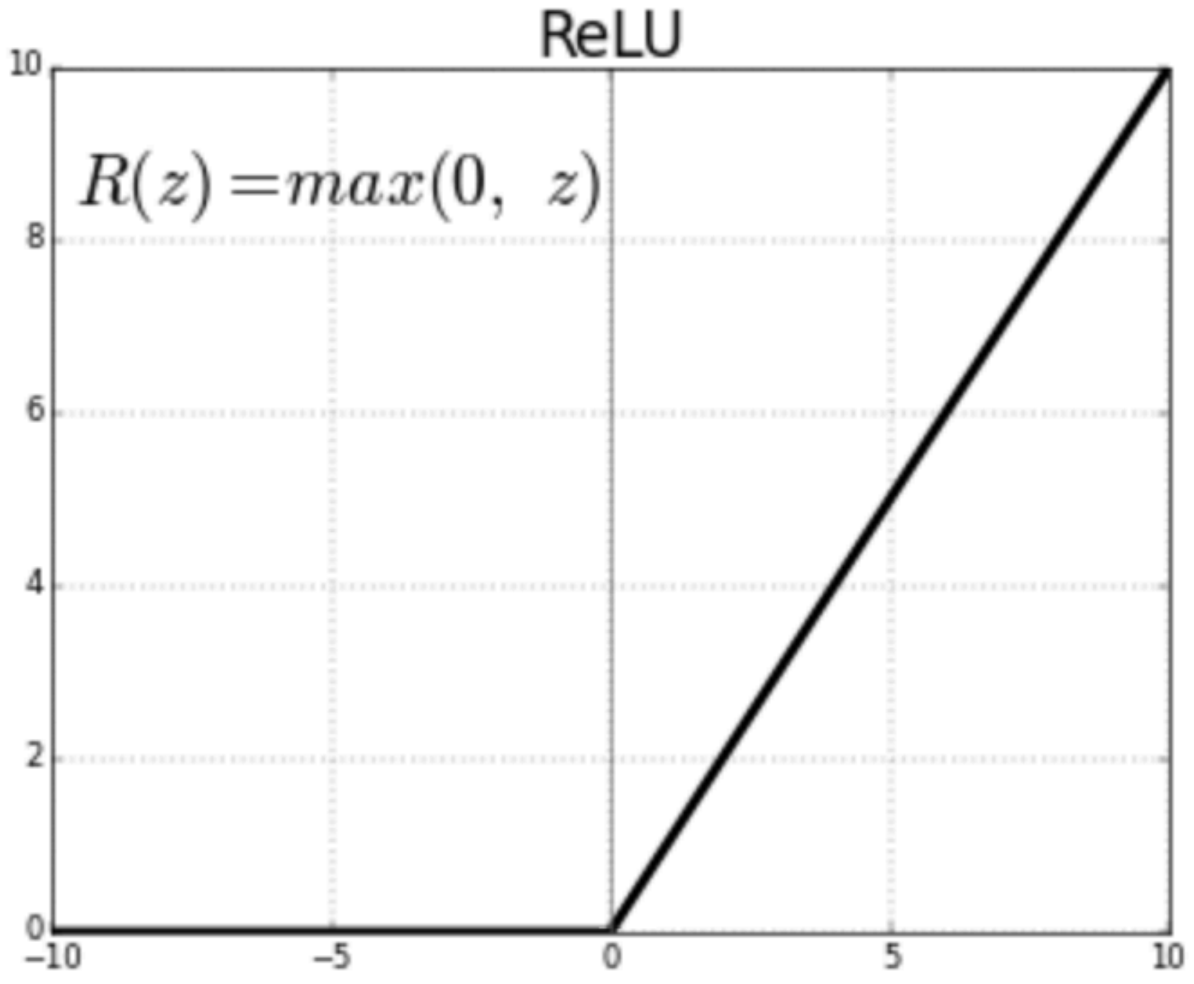
# Example: ReLU




$\text{ReLU}(x) = \max(0, x)$

```
class relu(Operation):
 def forward(ctx, x):
  return np.maximum(0, x)

 def backward(ctx, grad_output):
```

# Example: ReLU

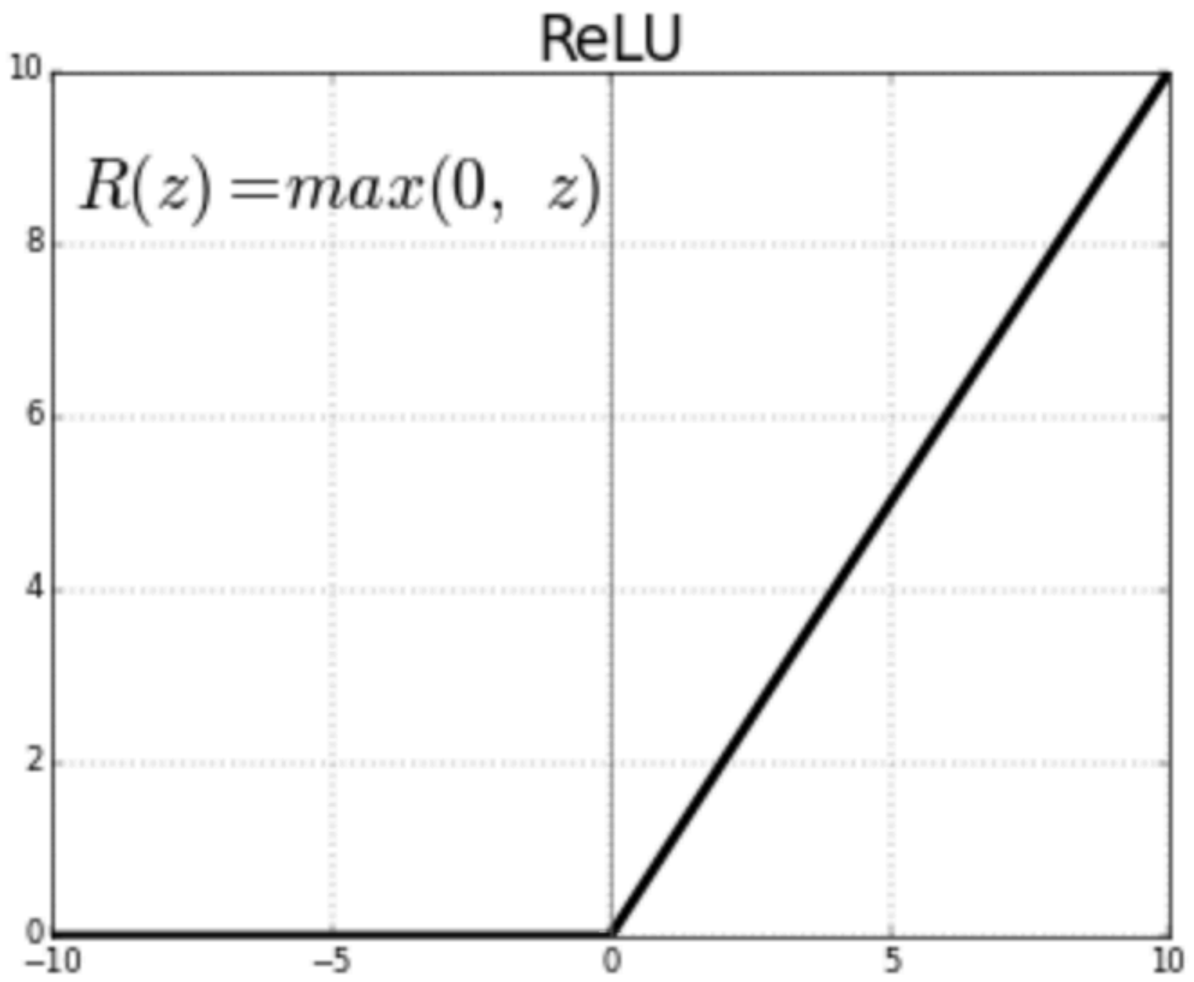



ReLU$(x) = \max(0,x)$

$$\frac{\partial R}{\partial x} = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

```
class relu(Operation):
  def forward(ctx, x):
    return np.maximum(0, x)

  def backward(ctx, grad_output):
```

# Example: ReLU




$\text{ReLU}(x) = \max(0, x)$

$$\frac{\partial R}{\partial x} = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

```
class relu(Operation):
  def forward(ctx, x):
    return np.maximum(0, x)

  def backward(ctx, grad_output):
```

🤔🤔🤔 where's x???

# Example: ReLU

```python
@tensor_op
class relu(Operation):
    @staticmethod
    def forward(ctx, value):
        new_val = np.maximum(0, value)
        ctx.append(new_val)
        return new_val

    @staticmethod
    def backward(ctx, grad_output):
        value = ctx[-1]
        return [(value > 0).astype(float) * grad_output]
```

# Example: ReLU

```
@tensor_op
class relu(Operation):
    @staticmethod
    def forward(ctx, value):
        new_val = np.maximum(0, value)
        ctx.append(new_val)
        return new_val

    @staticmethod
    def backward(ctx, grad_output):
        value = ctx[-1]
        return [(value > 0).astype(float) * grad_output]
```

Save and retrieve the input value!

# Example: ReLU

```
@tensor_op
class relu(Operation):
    @staticmethod
    def forward(ctx, value):
        new_val = np.maximum(0, value)
        ctx.append(new_val)
        return new_val

    @staticmethod
    def backward(ctx, grad_output):
        value = ctx[-1]
        return [(value > 0).astype(float) * grad_output]
```

Save and retrieve the input value!

local gradient times upstream gradient

# Example: ReLU

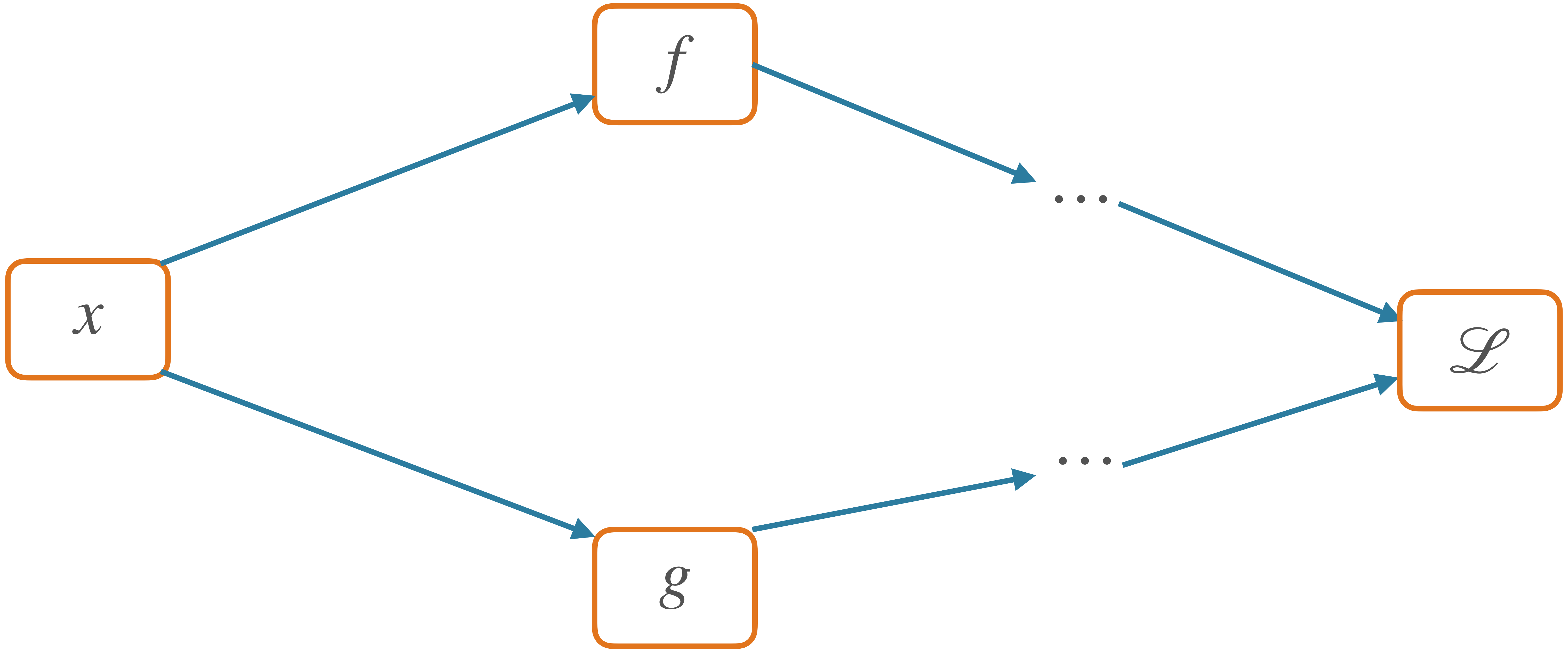
```
@tensor_op
class relu(Operation):
    @staticmethod
    def forward(ctx, value):
        new_val = np.maximum(0, value)
        ctx.append(new_val)
        return new_val

    @staticmethod
    def backward(ctx, grad_output):
        value = ctx[-1]
        return [(value > 0).astype(float) * grad_output]
```

Save and retrieve the input value!

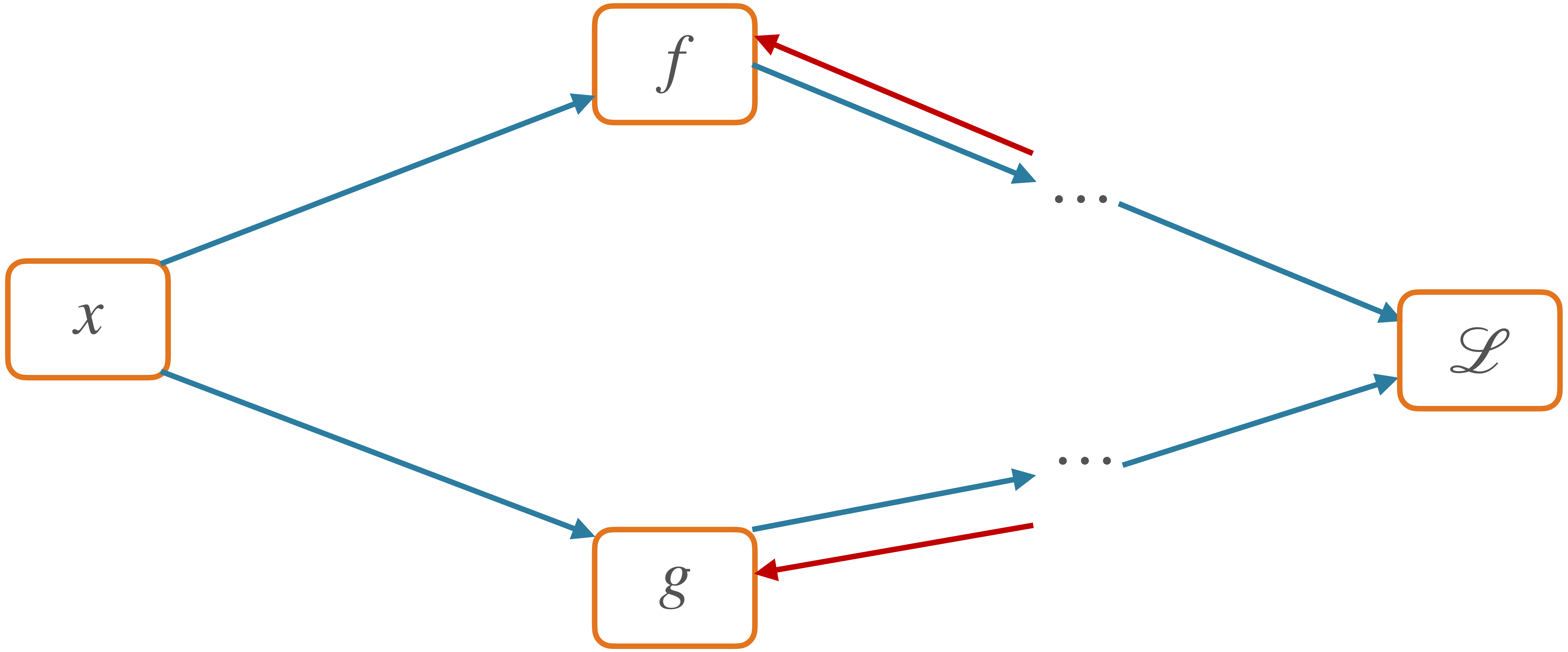
NB: list, one downstream gradient per input (in this case, one)

local gradient times upstream gradient

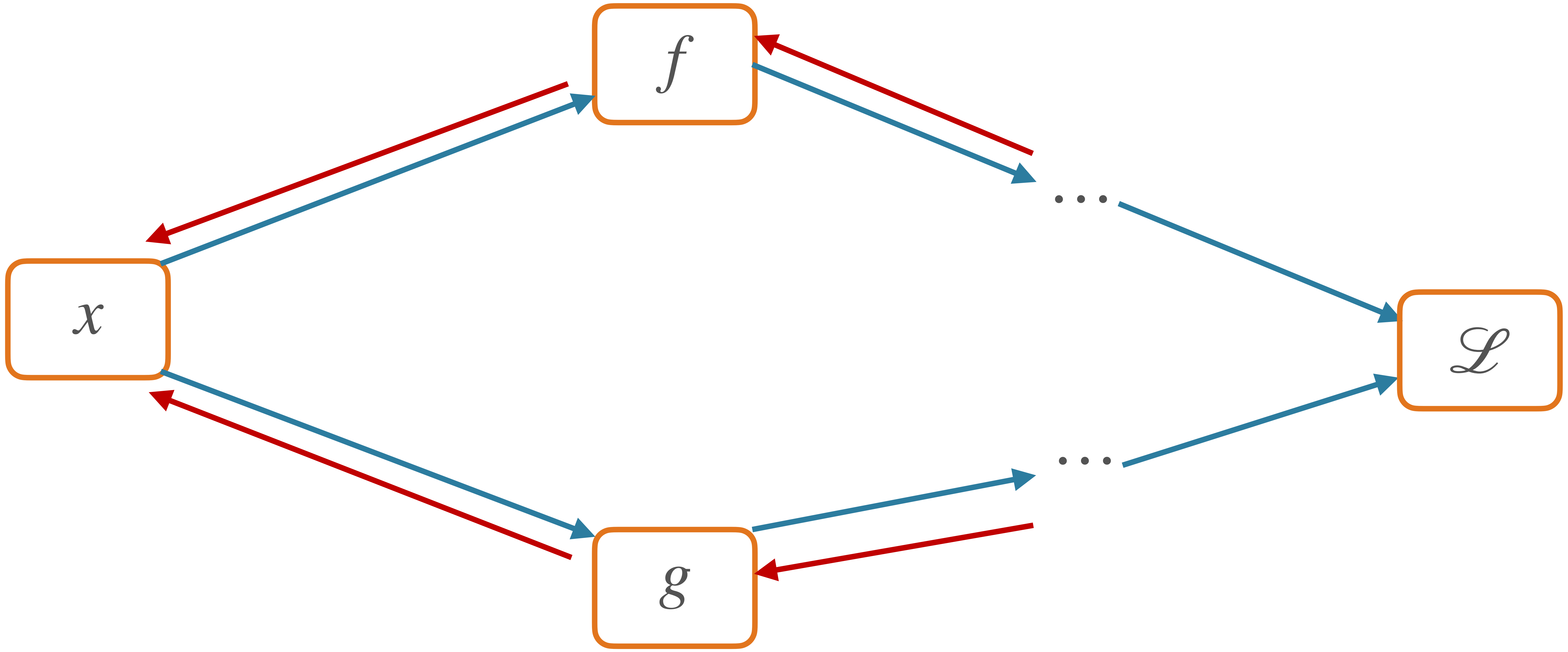
# Adding Gradients with Multiple Outputs



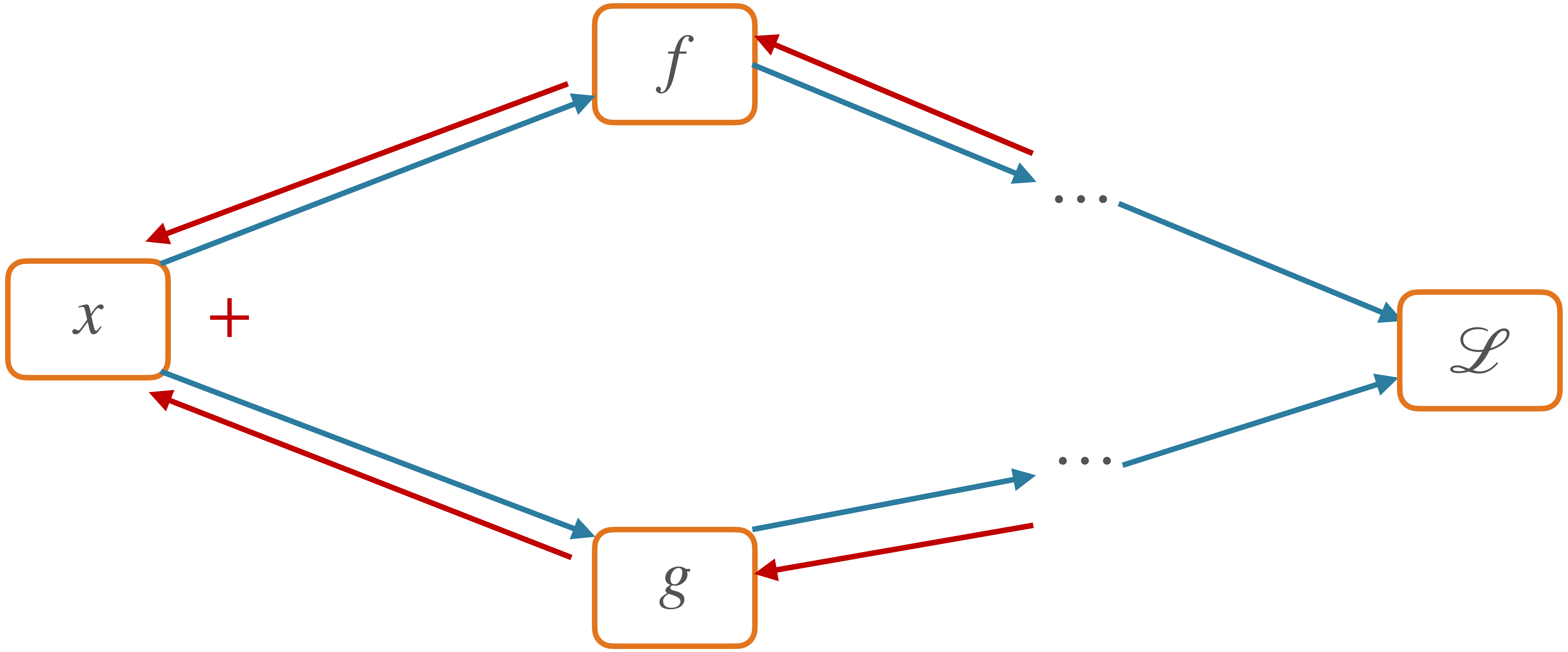
# Adding Gradients with Multiple Outputs



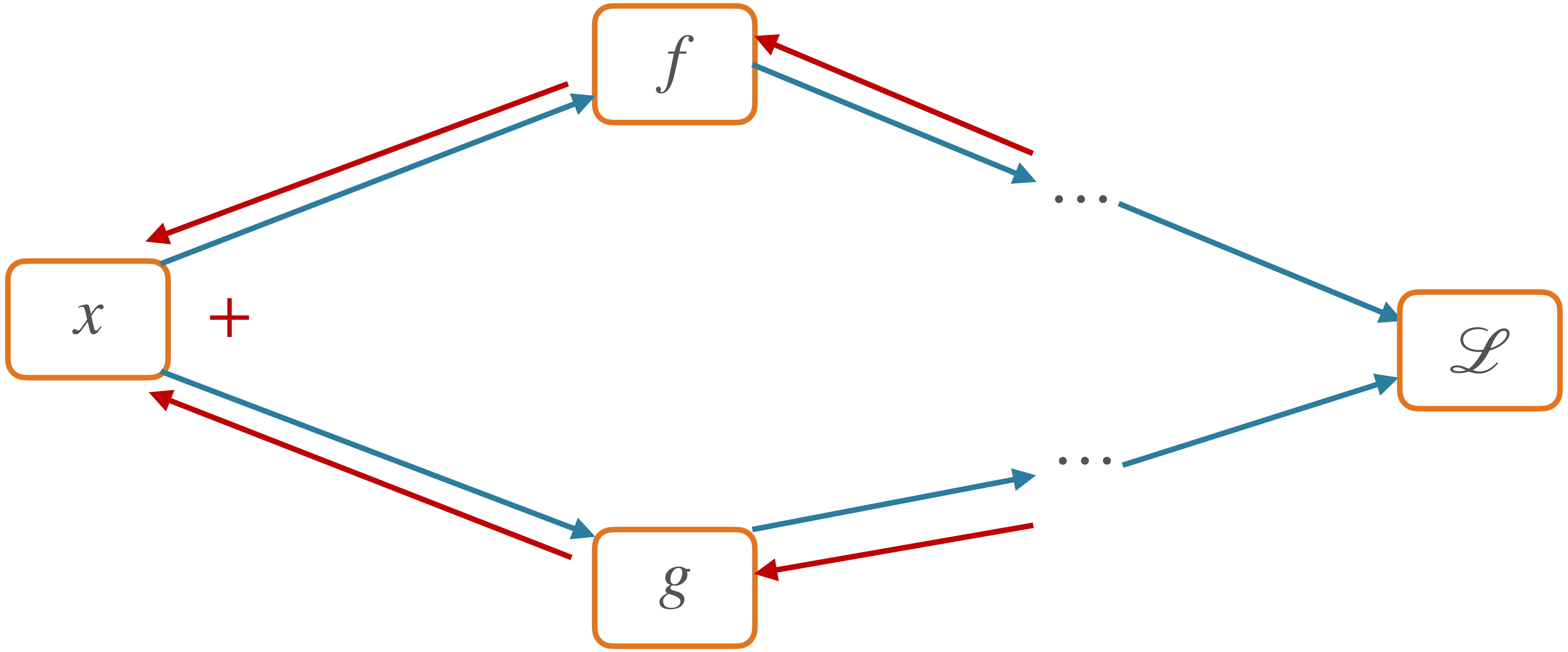
# Adding Gradients with Multiple Outputs



# Adding Gradients with Multiple Outputs



# Adding Gradients with Multiple Outputs




Multivariable chain rule: $$\frac{\partial \mathscr{L}}{\partial x} = \frac{\partial \mathscr{L}}{\partial f}\frac{\partial f}{\partial x} + \frac{\partial \mathscr{L}}{\partial g}\frac{\partial g}{\partial x}$$

# Adding Gradients with Multiple Outputs

$$f(x) = x^2 \times 3x$$

Live demo and/or exercise!

# Adding Gradients with Multiple Outputs

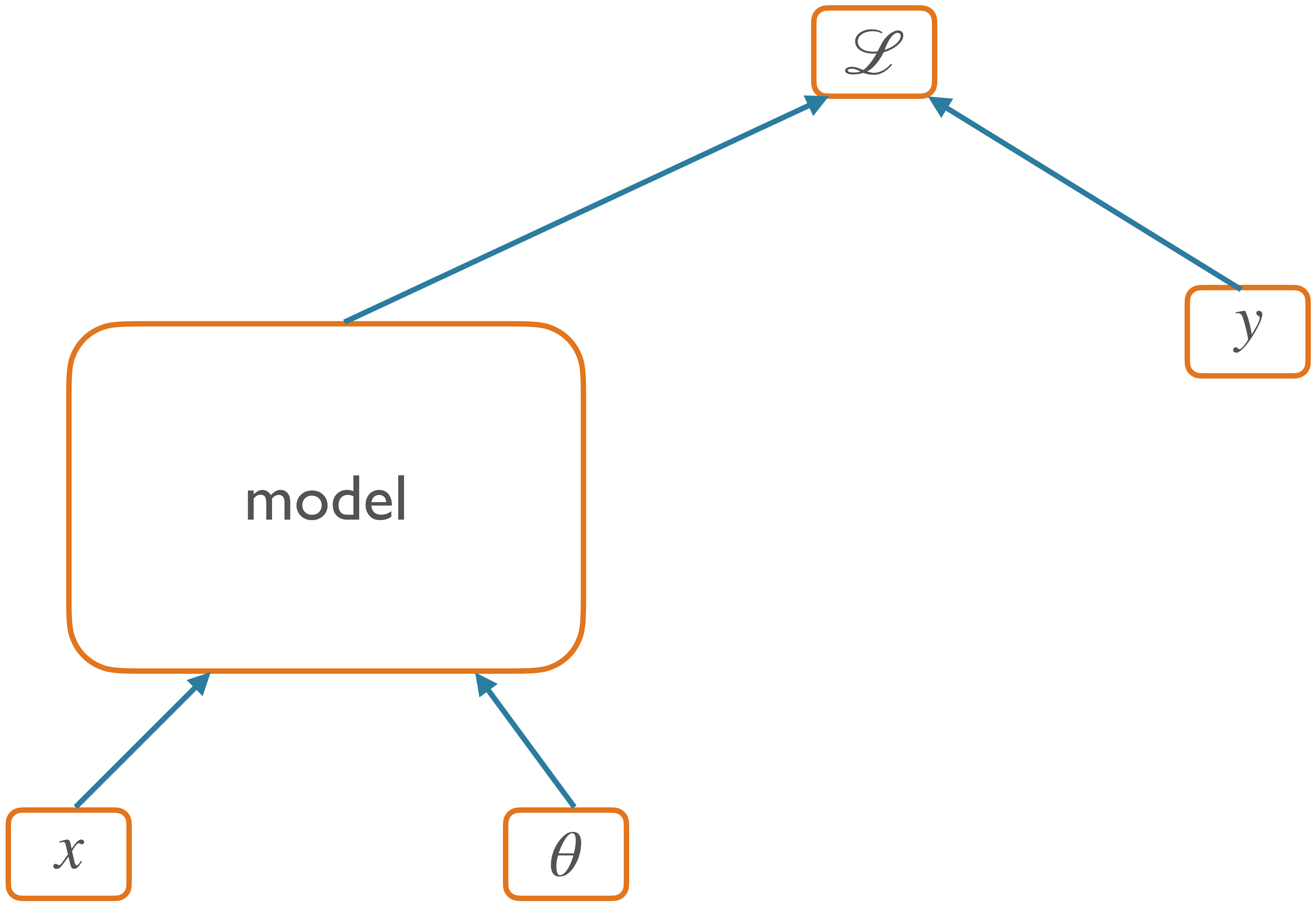
```
def _backward():
    grads = op.backward(ctx, new_tensor.grad)
    for idx in range(len(inputs)):
        inputs[idx].grad += grads[idx]
```

# Adding Gradients with Multiple Outputs

```
def _backward():
    grads = op.backward(ctx, new_tensor.grad)
    for idx in range(len(inputs)):
        inputs[idx].grad += grads[idx]
```

Adding over paths handled implicitly in auto-grad libraries; more power to the forward/backward API

# Schematic of Graph for Training



# Two Modes of Graph Construction

- Static (e.g. TensorFlow <2.x)
  - First: define entire graph structure
  - Then: pass in inputs, execute nodes
  - [session.run, feed_dicts, oh my!]
- Dynamic (e.g. PyTorch, TensorFlow 2.x)
  - The graph is defined *dynamically* in the forward pass
  - E.g. operators on Tensors store the links to their input Tensors, thus building a graph

# Training Loop

- Define (now, dynamically) computation graph, get backprop “automatically”

# Training Loop

- Define (now, dynamically) computation graph, get backprop “automatically”

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

# Training Loop

- Define (now, dynamically) computation graph, get backprop “automatically”

Backprop the loss!

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

# Training Loop

- Define (now, dynamically) computation graph, get backprop "automatically"

Backprop the loss!

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

Update the parameters

# Training Loop

- Define (now, dynamically) computation graph, get backprop “automatically”

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

Backprop the loss!

Update the parameters

Yes, you should understand backdrop!

# More Resources

- Debugging:
  - Symbolic gradient computation; $f(x+h) - f(x-h)/2h$
  - Shapes! Gradients should be same shape as values [b/c scalar outputs]
- Computing vector/matrix derivatives
  - Work with small toy examples, compute for a single element, generalize
  - http://cs231n.stanford.edu/vecDerivs.pdf
  - http://web.stanford.edu/class/cs224n/readings/gradient-notes.pdf

# Next Time

- Feed-forward models for:
  - Classification: Deep Averaging Network
  - Language Modeling
- Training tips and tricks