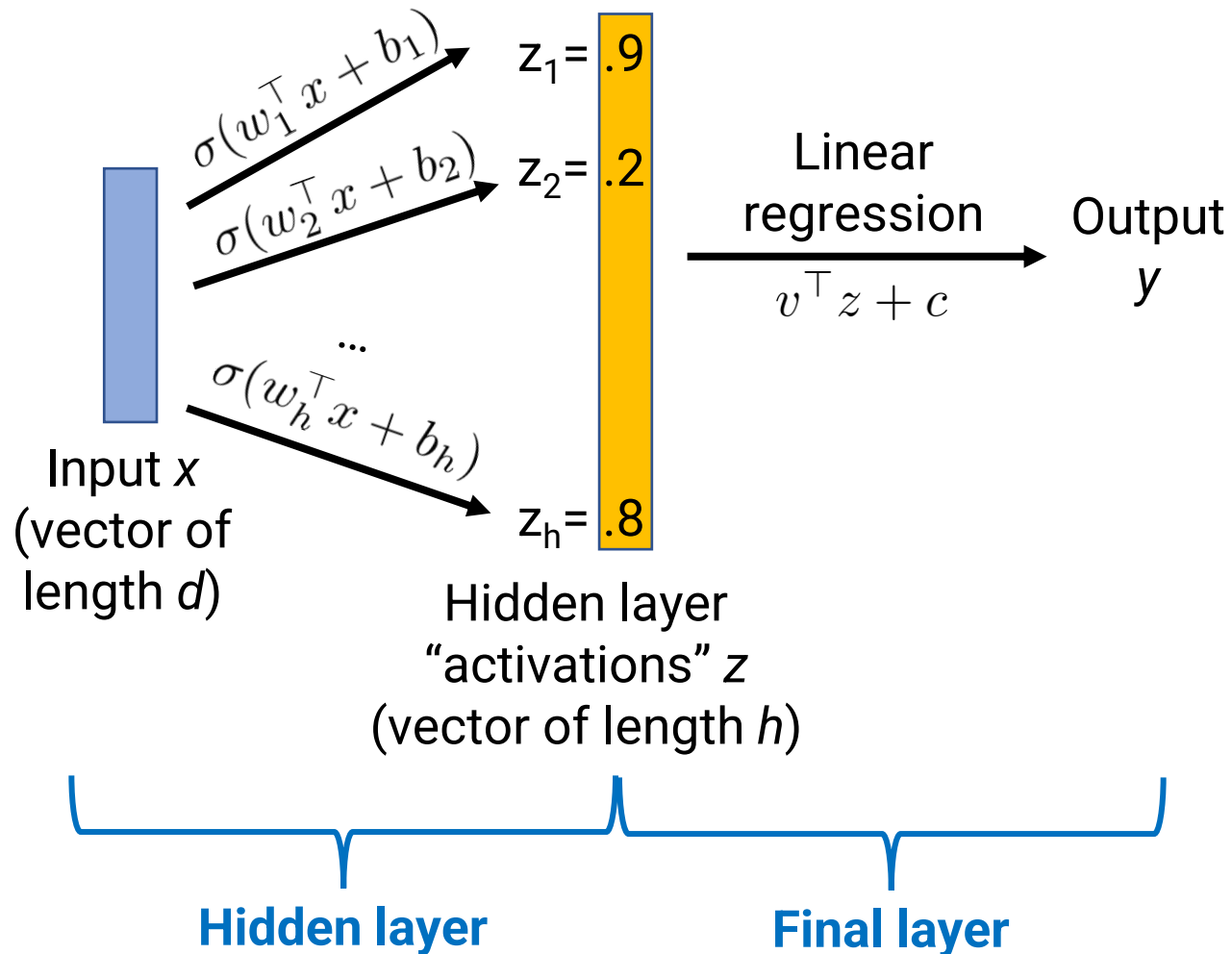


Neural Networks II: Backpropagation

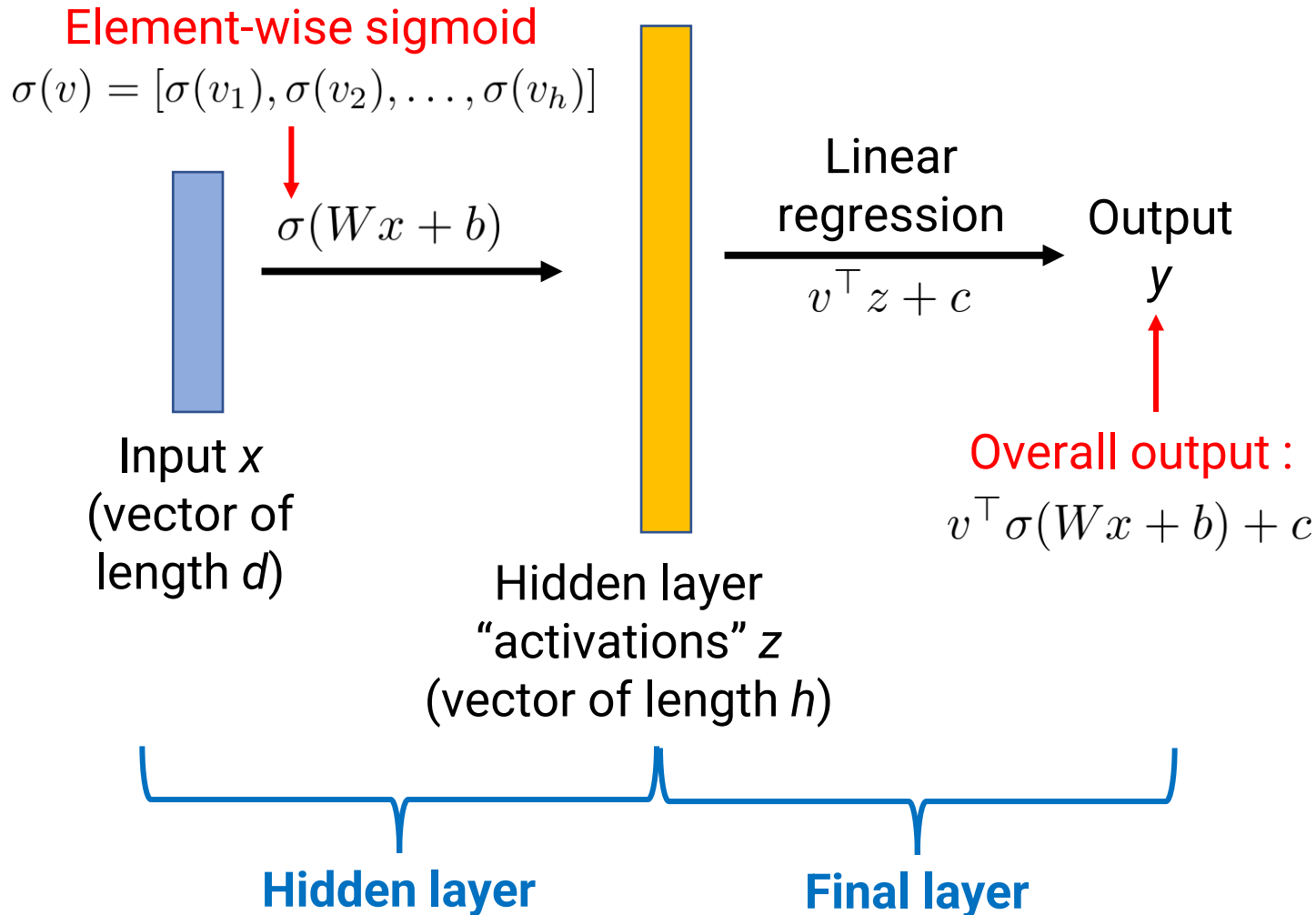
Robin Jia
USC CSCI 467, Fall 2026
September 23, 2026

Review: Neural Networks (2-layer MLP)



- Hidden layer = A bunch of logistic regression classifiers
 - Parameters: w_j and b_j for each classifier, for each $j=1, \dots, h$
 - h = number of neurons in hidden layer ("hidden nodes")
 - Produces "activations" = learned feature vector
- Final layer = linear model
 - For regression: linear model with weight vector v and bias c

Review: Neural Networks (2-layer MLP)

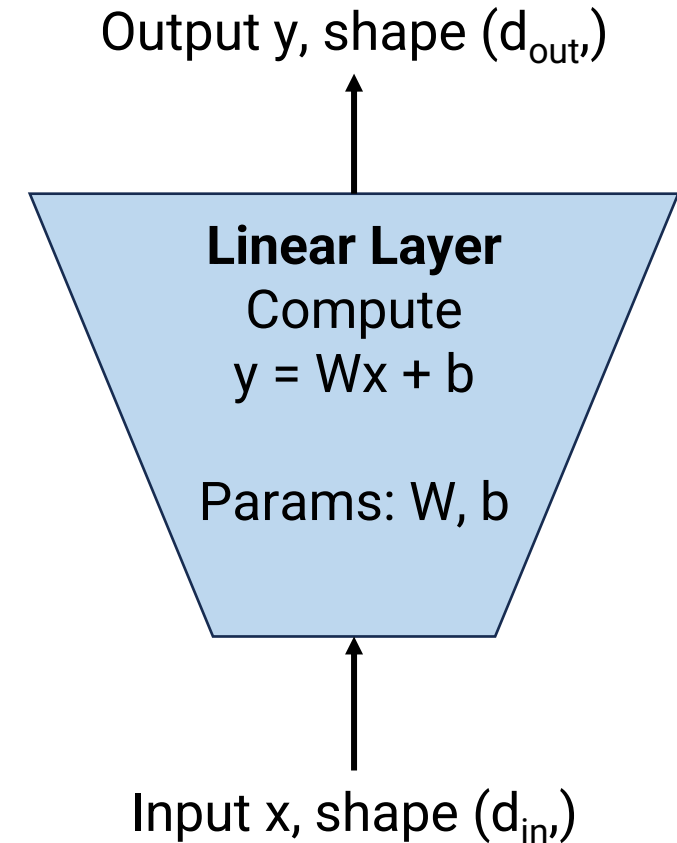


- Hidden layer = A bunch of logistic regression classifiers
 - Parameters: w_j and b_j for each classifier, for each $j=1, \dots, h$
 - **Equivalently: matrix W ($h \times d$) and vector b (length h)**
 - h = number of neurons in hidden layer (“hidden nodes”)
 - Produces “activations”= learned feature vector
- Final layer = linear model
 - For regression: linear model with weight vector v and bias c
- **Parameters of model are $\theta = (W, b, v, c)$**

Review: Neural Network Building Blocks

(1) Linear Layer

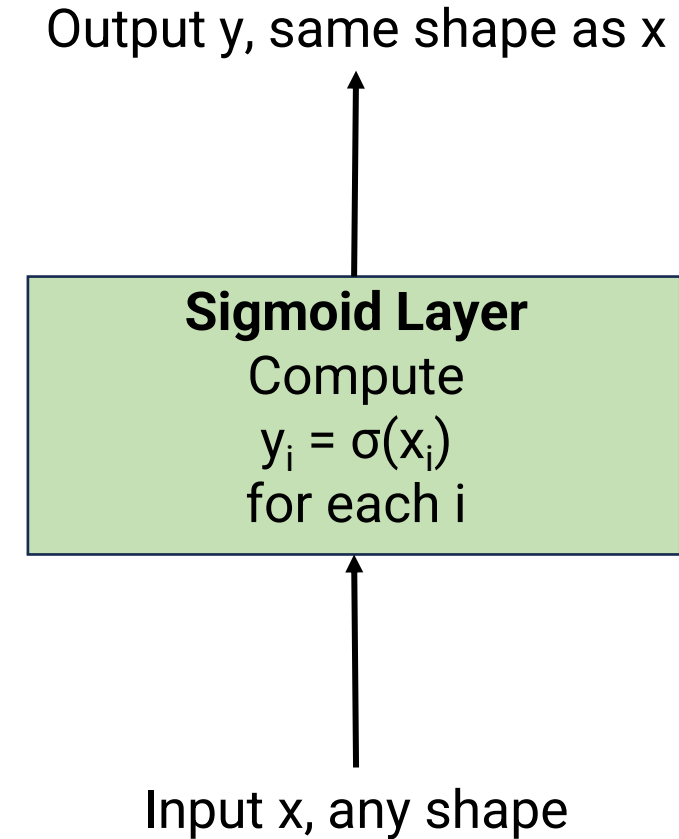
- Input x : Vector of dimension d_{in}
- Output y : Vector of dimension d_{out}
- Formula: $y = Wx + b$
- Parameters
 - W : $d_{out} \times d_{in}$ matrix
 - b : d_{out} vector
- In pytorch: `nn.Linear()`



Review: Neural Network Building Blocks

(2) Non-linearity Layer

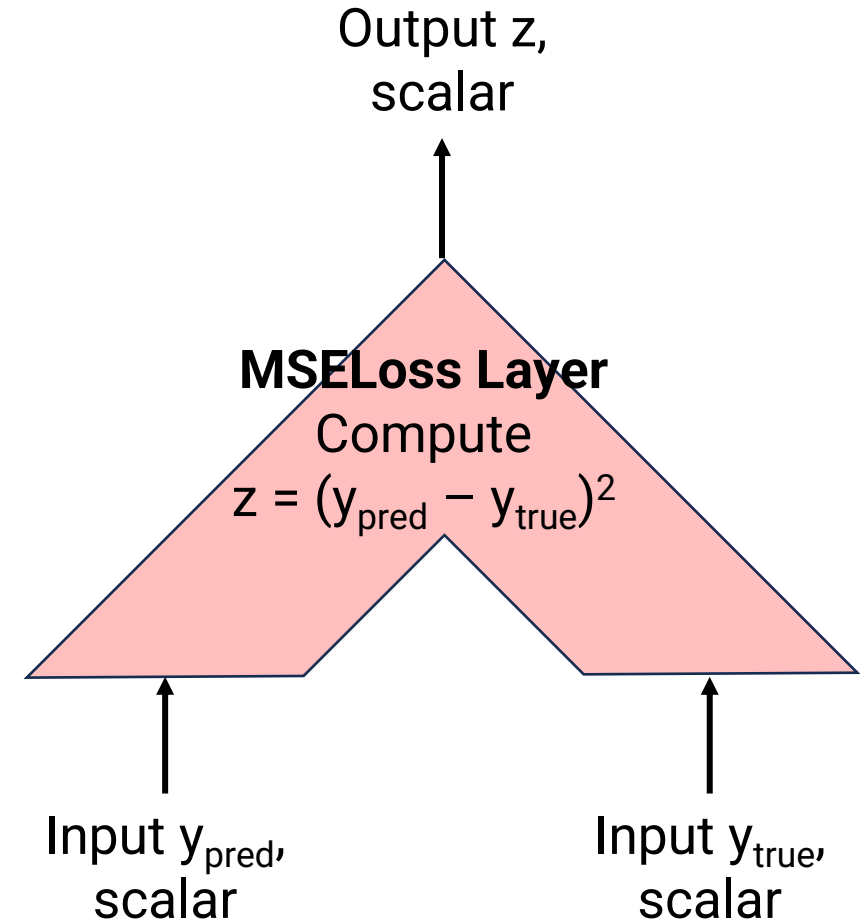
- Input x : Any number/vector/matrix
- Output y : Number/vector/matrix of same shape
- Possible formulas:
 - Sigmoid: $y = \sigma(x)$, elementwise
 - Tanh: $y = \tanh(x)$, elementwise
 - Relu: $y = \max(x, 0)$, elementwise
- Parameters: None
- In pytorch: `torch.sigmoid()`, `nn.functional.relu()`, etc.



Review: Neural Network Building Blocks

(3) Loss Layer

- Inputs:
 - y_{pred} : shape depends on task
 - y_{true} : scalar (e.g., correct regression value or class index)
- Output z : scalar
- Possible formulas:
 - Squared loss: y_{pred} is scalar, $z = (y_{\text{pred}} - y_{\text{true}})^2$
 - Softmax regression loss: y_{pred} is vector of length C ,
$$z = - \left(y_{\text{pred}}[y_{\text{true}}] - \log \sum_{i=1}^C \exp(y_{\text{pred}}[i]) \right)$$
- Parameters: None
- In pytorch: `nn.MSELoss()`, `nn.CrossEntropyLoss()`, etc.



Review: Training Neural Networks

Linear Regression

- Model's output is

$$g(x) = w^\top x + b$$

- (Unregularized) loss function is

$$\frac{1}{n} \sum_{i=1}^n (g(x^{(i)}) - y^{(i)})^2$$

Regression w/ Neural Networks

- Model's output is

$$g(x) = v^\top \sigma(Wx + b) + c$$

- **Use same loss function**, in terms of g !

$$\frac{1}{n} \sum_{i=1}^n (g(x^{(i)}) - y^{(i)})^2$$

Training objective for both types of models:

$$\frac{1}{n} \sum_{i=1}^n \ell(y^{(i)}, g(x^{(i)})), \text{ where } \ell(y, u) = (y - u)^2$$

Also applies for
logistic regression,
softmax regression, etc.

Review: Training Neural Networks

General loss function: $\frac{1}{n} \sum_{i=1}^n \ell \left(y^{(i)}, g(x^{(i)}) \right)$

Model's output, depends on all model parameters θ (includes all layers)

- How to minimize? Gradient Descent!

$$\theta \leftarrow \theta - \eta \cdot \underbrace{\frac{1}{n} \sum_{i=1}^n \nabla_{\theta} \ell \left(y^{(i)}, g(x^{(i)}) \right)}_{\text{Average of per-example gradients}}$$

Average of per-example gradients

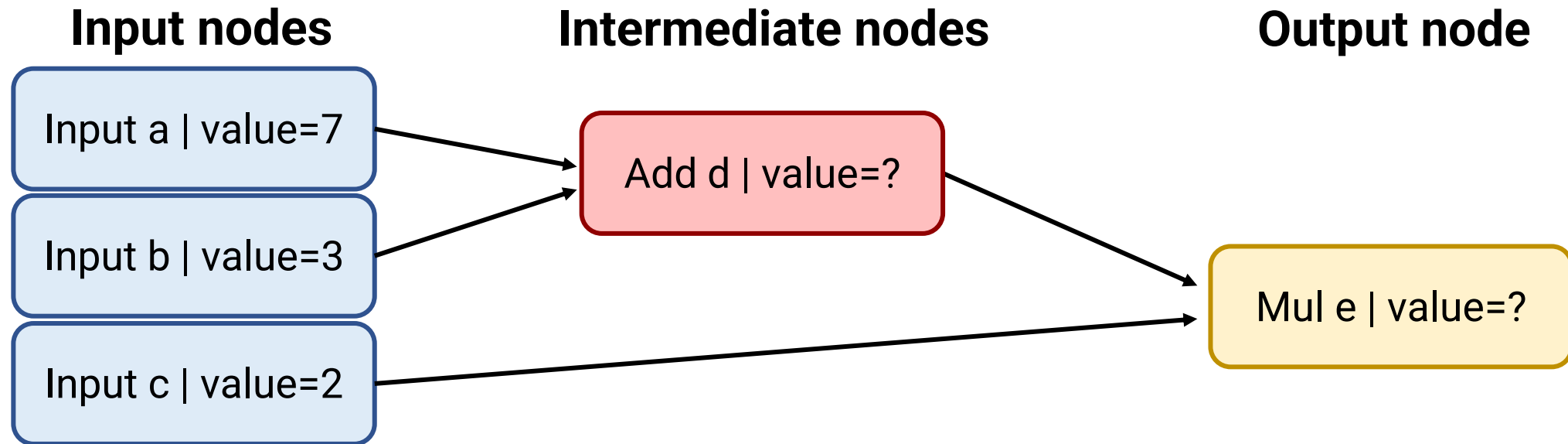
- **Today: How to compute gradient of loss w.r.t. parameters for any neural network**
 - So many different ways to assemble building blocks
 - Don't want to re-do gradient calculations by hand each time
 - **Can we write an algorithm to do it?**

Today's Plan

- The computation graph
- Backpropagation on trees
- Backpropagation on DAGs

Computation Graph

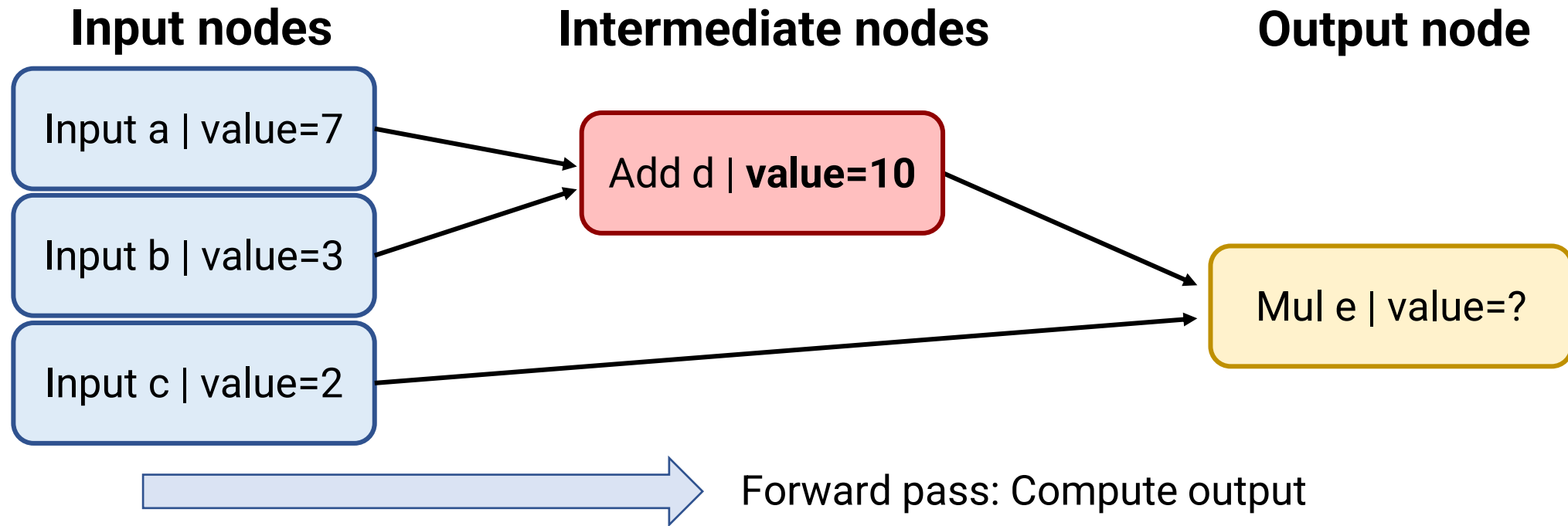
Computation graph for $(a + b) * c$ when $a=7, b=3, c=2$



Different way of drawing the “building blocks” of neural networks

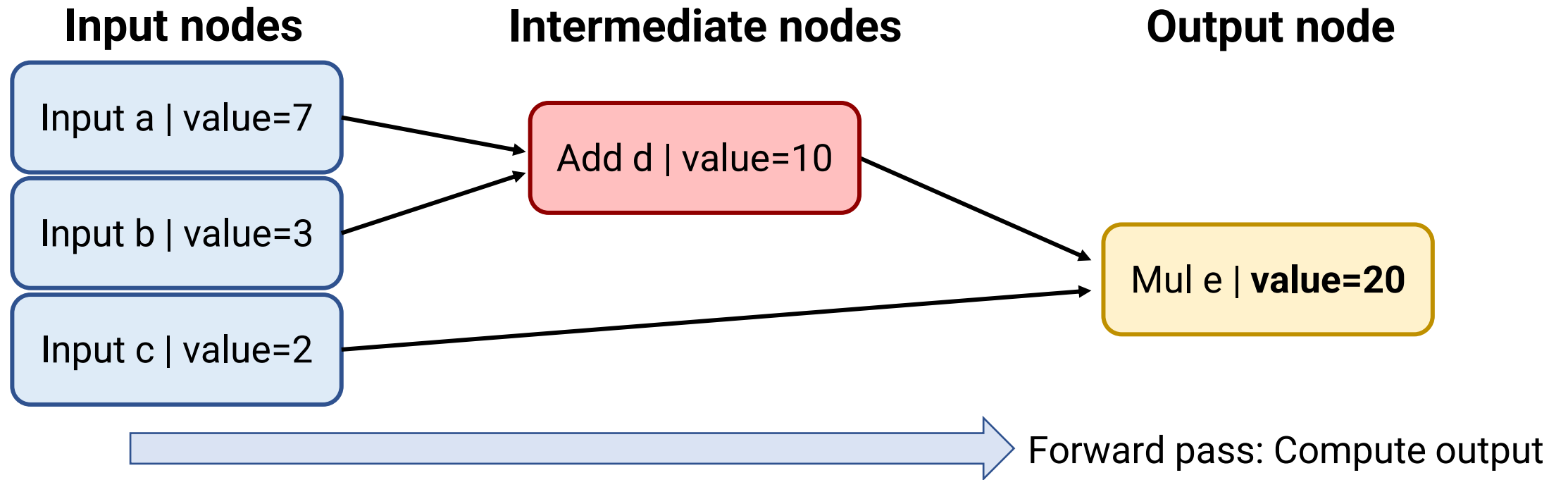
Computation Graph

Computation graph for $(a + b) * c$ when $a=7, b=3, c=2$



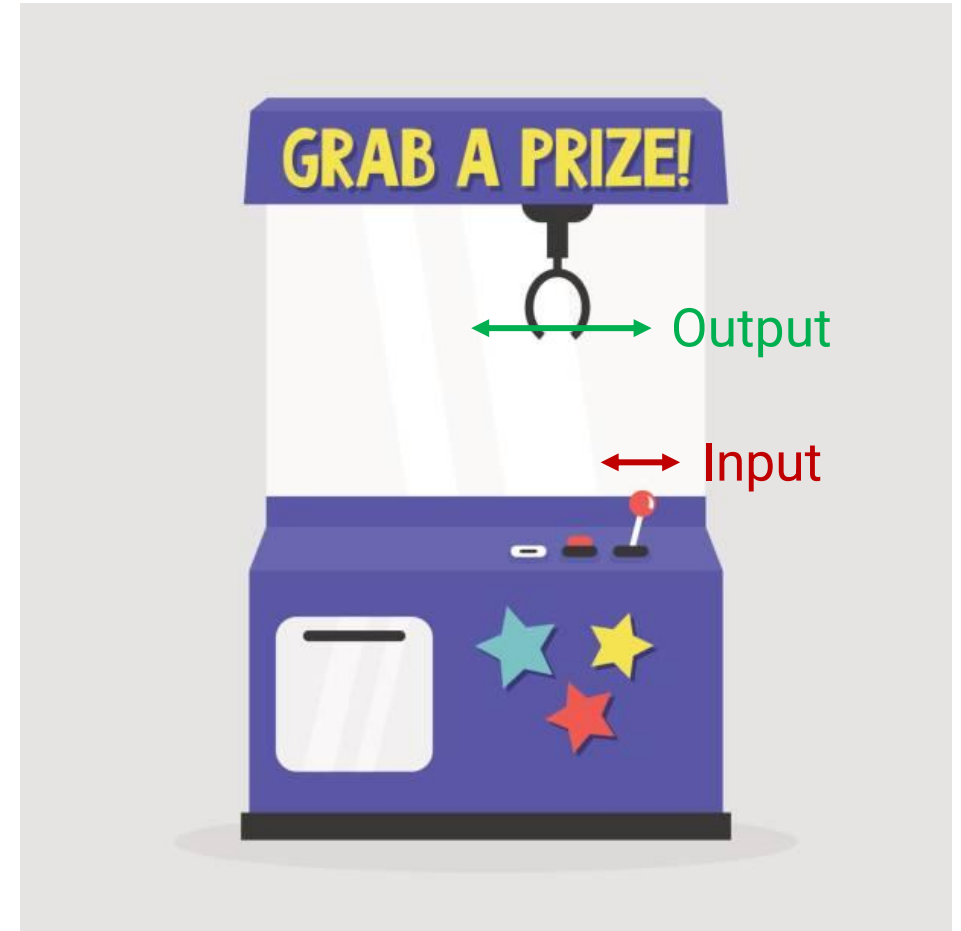
Computation Graph

Computation graph for $(a + b) * c$ when $a=7, b=3, c=2$



Gradient checking

- **Numerical gradients:** A simpler but less efficient way to compute gradients
- What does $\partial y / \partial x$ mean?
 - If I change x by epsilon, by what proportion of epsilon does y change?
- We can just compute this for every input node!
- Pro: Easy to implement, useful to check correctness
- Con: Slow—requires $O(\text{\#inputs})$ function evaluations



Let's implement!

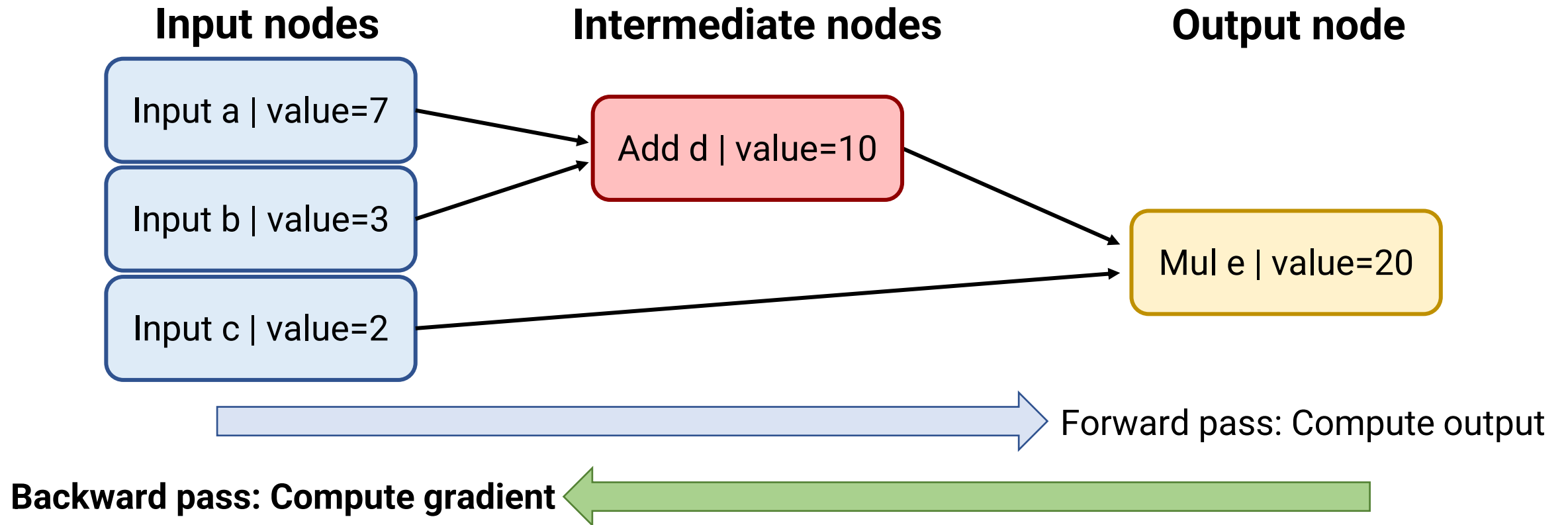


Today's Plan

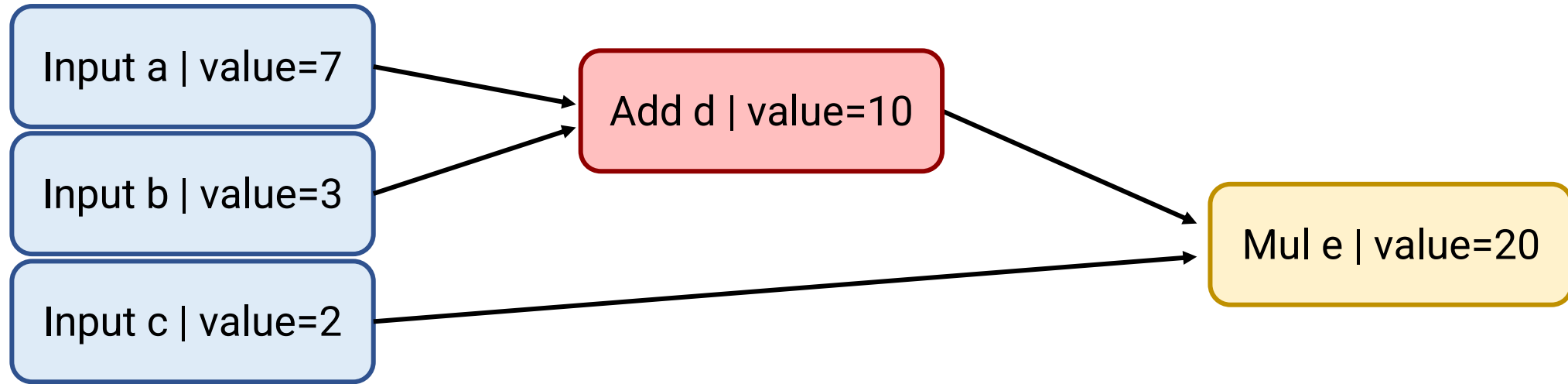
- The computation graph
- **Backpropagation on trees**
- Backpropagation on DAGs

Computation Graph

Computation graph for $(a + b) * c$ when $a=7, b=3, c=2$

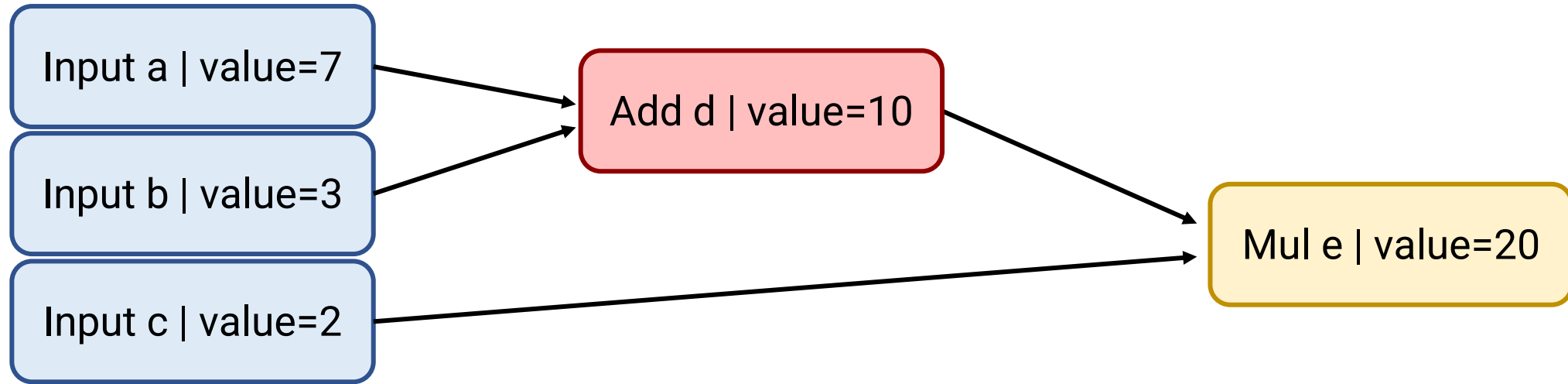


Backpropagation on trees



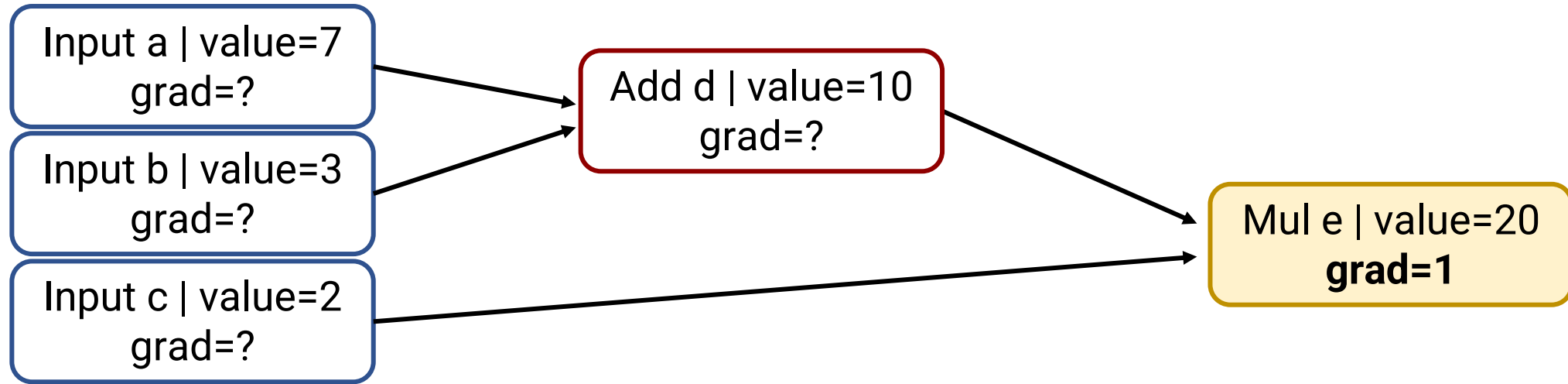
- For now: assume that the computation graph is a tree
 - Each node is only used in a single computation
 - Root of tree is output
 - Leaves of tree are inputs
- Idea: Recursively compute $\partial(\text{output})/\partial(\text{node})$ for each node, starting at output

Backpropagation on trees



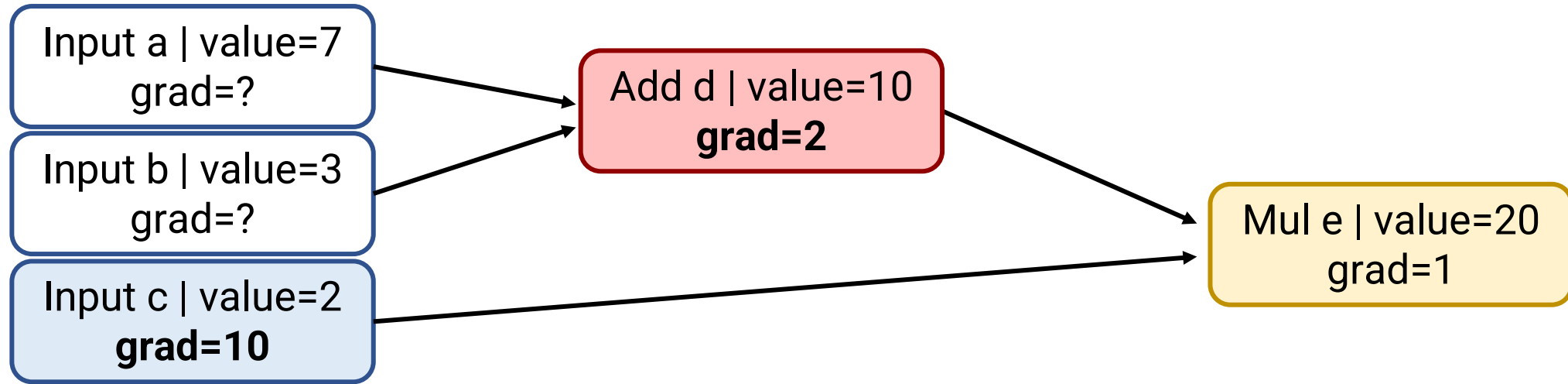
- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b, \partial e / \partial c]$

Backpropagation on trees



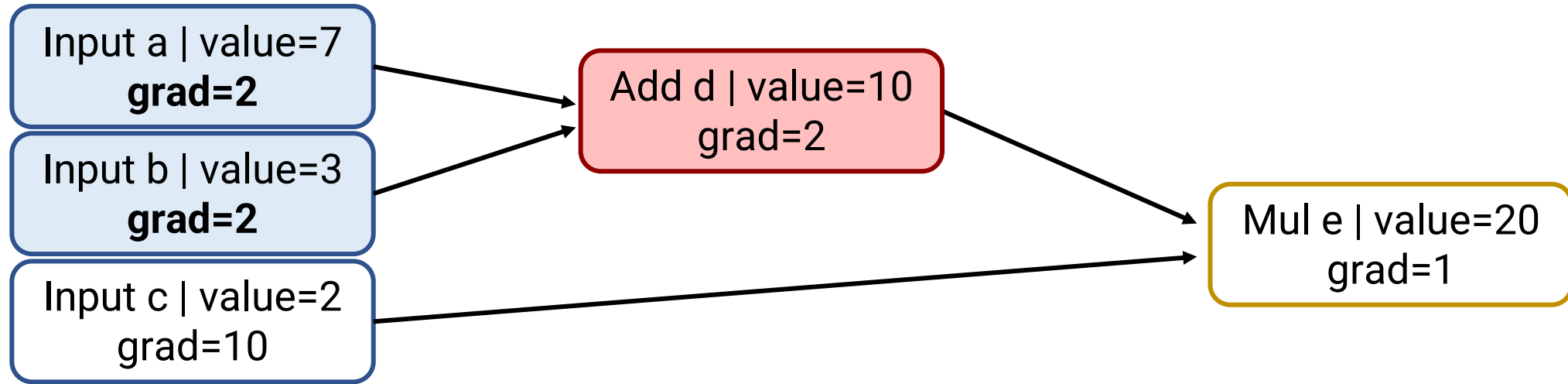
- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b, \partial e / \partial c]$
- Step 1: Base case: $\partial e / \partial e = 1$

Backpropagation on trees



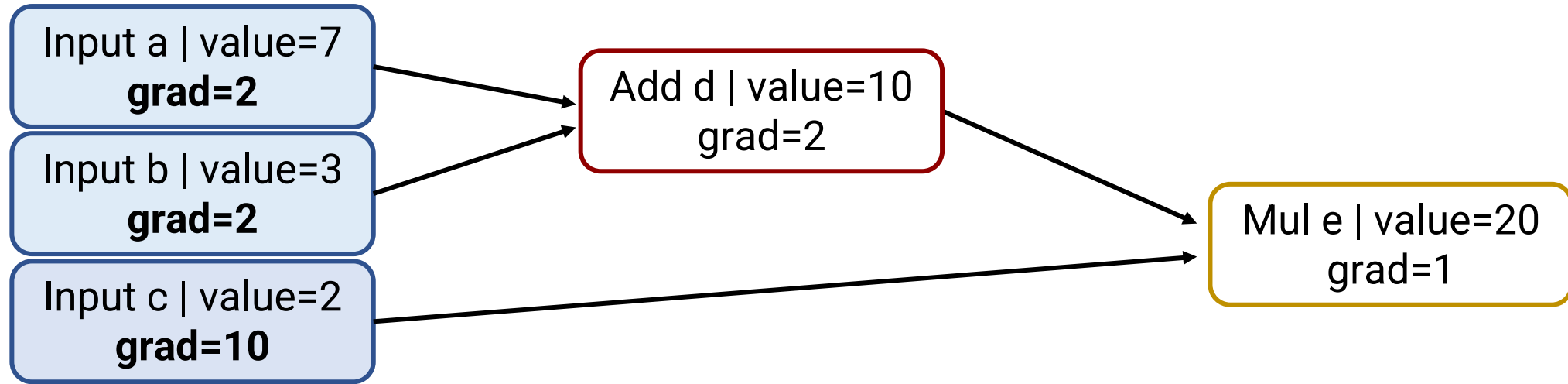
- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b, \partial e / \partial c]$
- Step 2: How does **Mul** (node e) “distribute” gradient to its children?
 - $\partial(x*y) / \partial x = y$
 - Chain Rule: $\partial(out) / \partial x = \partial(out) / \partial(x*y) * \partial(x*y) / \partial x = \partial(out) / \partial(x*y) * y$
 - General rule: Child gets **parent's gradient** * **value of other child**

Backpropagation on trees



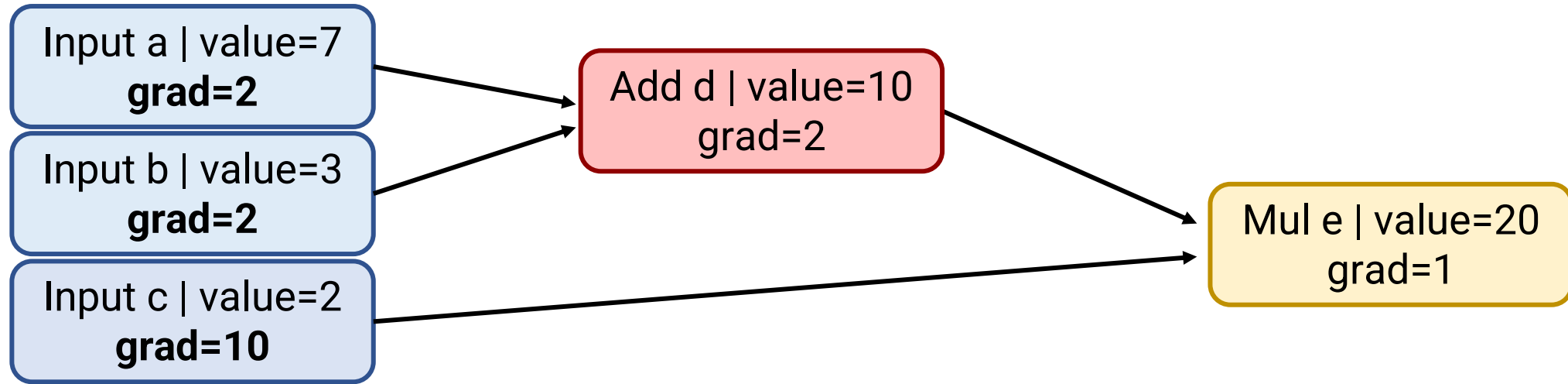
- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b, \partial e / \partial c]$
- Step 3: How does **Add** (node d) “distribute” gradient to its children?
 - $\partial(x+y) / \partial x = 1$
 - Chain Rule: $\partial(out) / \partial x = \partial(out) / \partial(x+y) * \partial(x+y) / \partial x = \partial(out) / \partial(x+y) * 1$
 - General rule: Child gets **parent’s gradient** * 1

Backpropagation on trees



- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b, \partial e / \partial c]$
- Step 4: Leaf nodes
 - Don't need to do anything

Backpropagation on trees

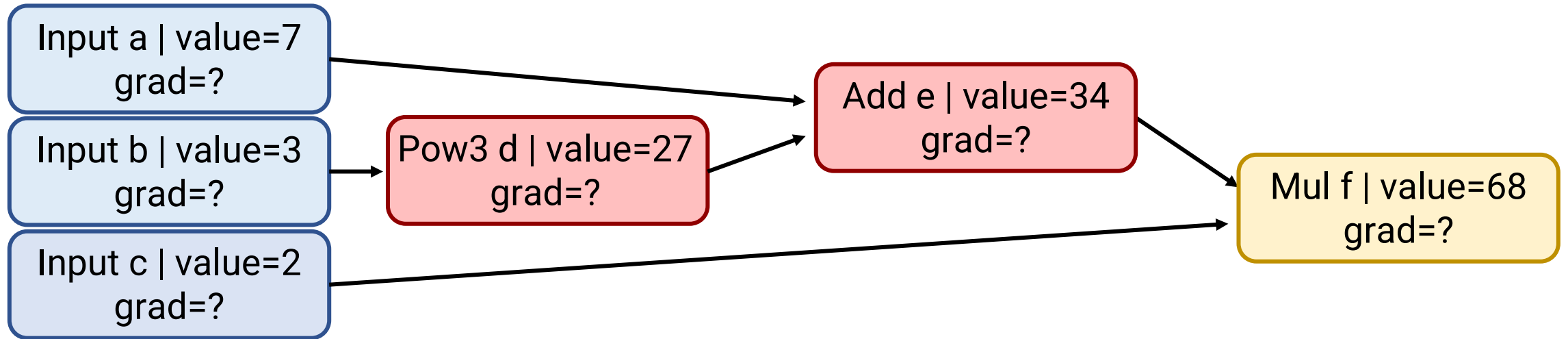


- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b, \partial e / \partial c]$
- Overall Recipe
 - Do forward pass
 - Start at root and recurse over children
 - Each node knows how to take **gradient of itself with respect to each child**
 - By Chain Rule, $\text{child.grad} = \text{parent.grad} * \partial(\text{parent}) / \partial(\text{child})$

Let's implement!

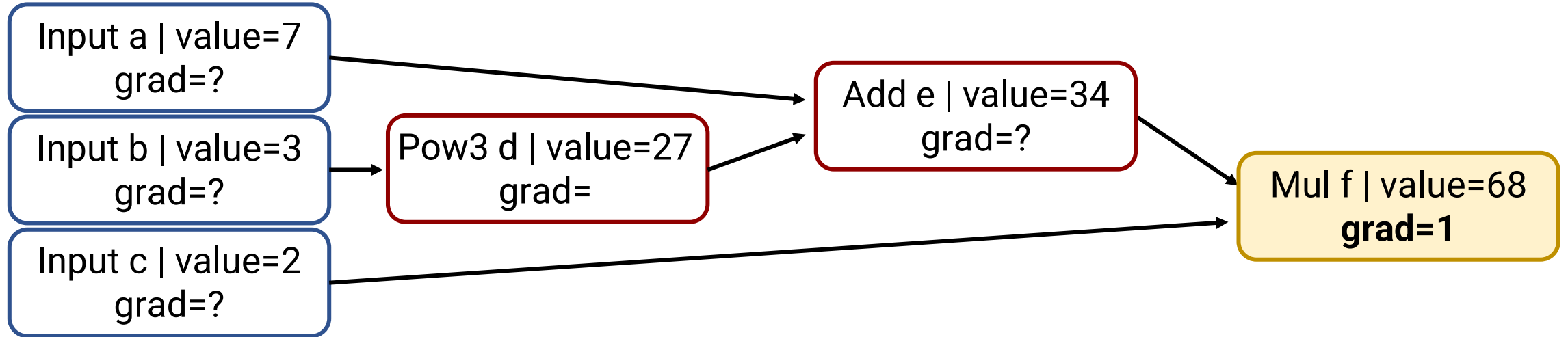


PowerNode



New function: $(a + b^3) * c$ where $a=7, b=3, c=2$

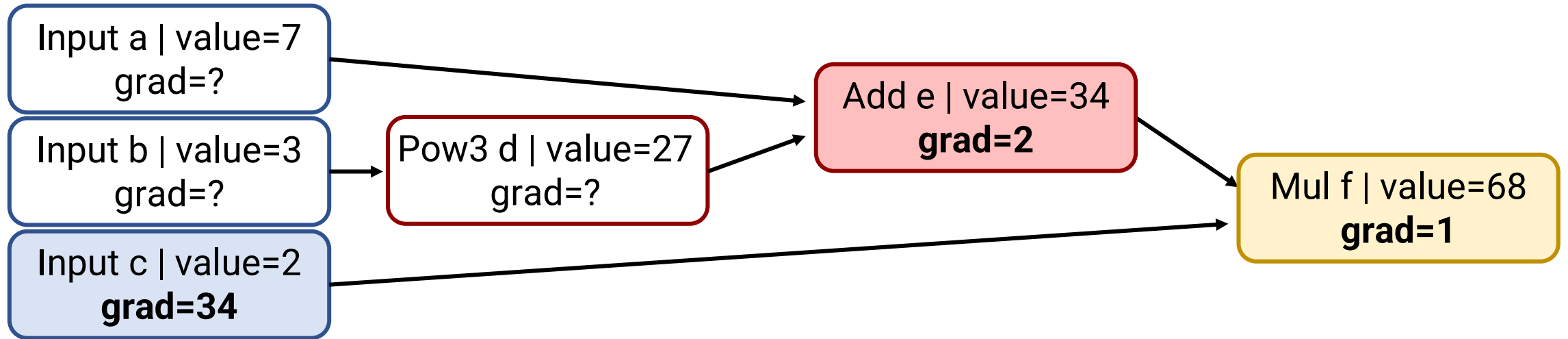
PowerNode



New function: $(a + b^3) * c$ where $a=7, b=3, c=2$

- Goal: Compute gradient $[\partial f / \partial a, \partial f / \partial b, \partial f / \partial c]$
- Step 1: Base case: $\partial f / \partial f = 1$

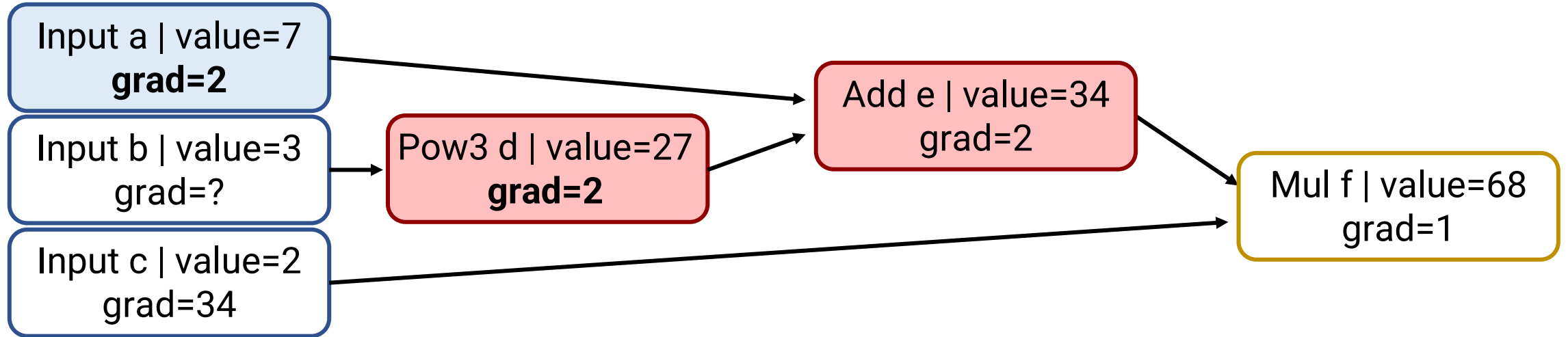
PowerNode



New function: $(a + b^3) * c$ where $a=7, b=3, c=2$

- Goal: Compute gradient $[\partial f / \partial a, \partial f / \partial b, \partial f / \partial c]$
- Step 2: Distribute **Mul** (node f) gradient to children

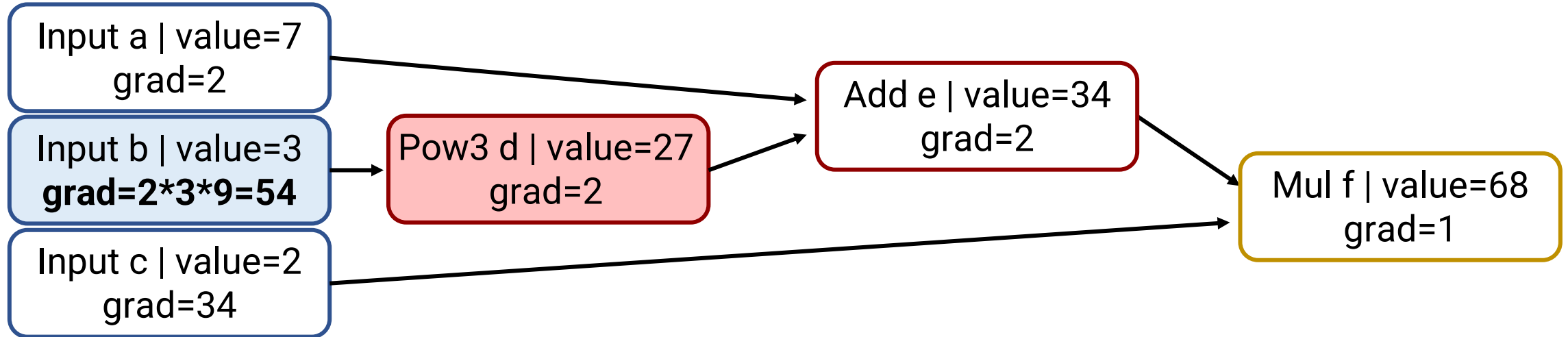
PowerNode



New function: $(a + b^3) * c$ where $a=7, b=3, c=2$

- Goal: Compute gradient $[\partial f / \partial a, \partial f / \partial b, \partial f / \partial c]$
- Step 3: Distribute **Add** (node e) gradient to children

PowerNode



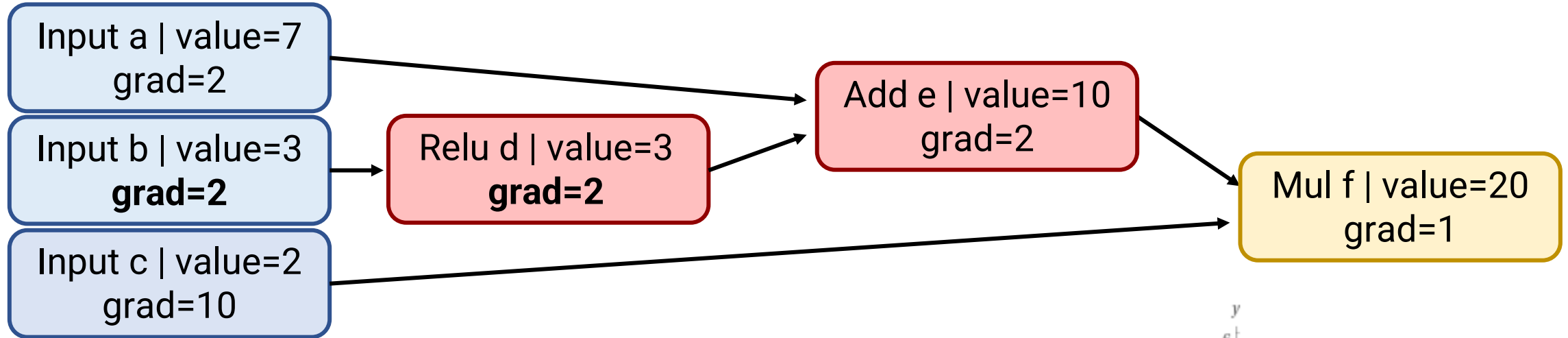
New function: $(a + b^3) * c$ where $a=7, b=3, c=2$

- Goal: Compute gradient $[\partial f / \partial a, \partial f / \partial b, \partial f / \partial c]$
- Step 4: Distribute **Pow3** (node d) gradient to children
 - $\partial(x^p) / \partial x = p * x^{p-1}$
 - By Chain Rule: Child gets **parent's gradient** * $p * child^{p-1}$

Let's implement!

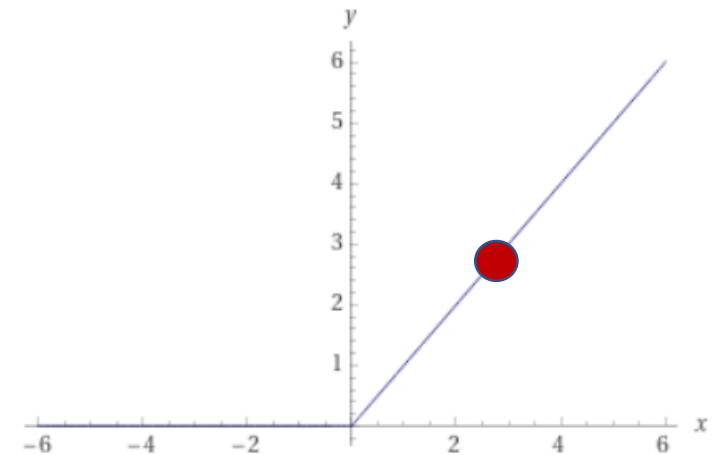


ReluNode



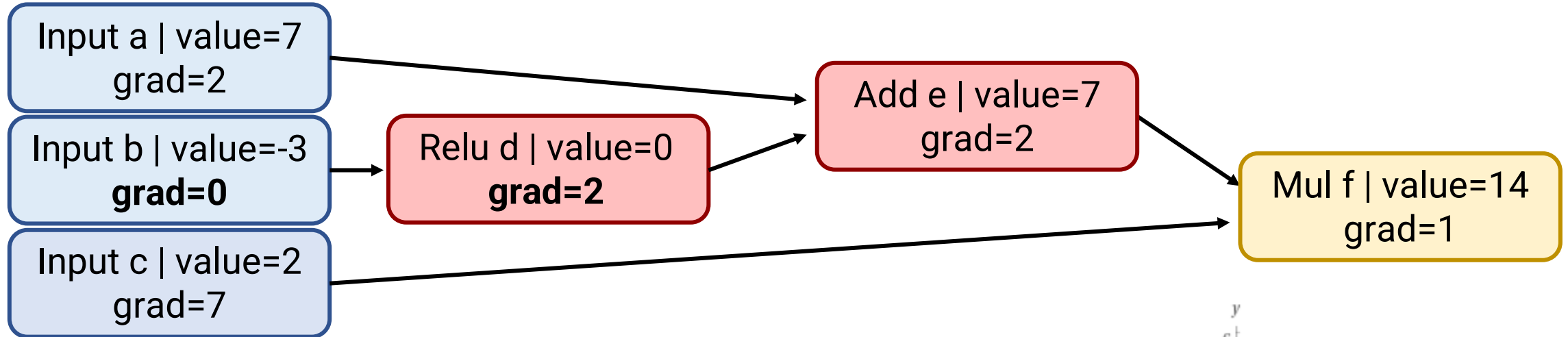
New function: $(a + \text{Relu}(b)) * c$ where $a=7, b=3, c=2$

- Steps 1-3 are the same
- Step 4: Relu
 - $\partial(\text{Relu}(x))/\partial x = 1$ if $x > 0$, 0 if $x \leq 0$
 - If $\text{child} > 0$, $\text{child.grad} = \text{parent.grad} * 1$
 - If $\text{child} \leq 0$, $\text{child.grad} = 0$



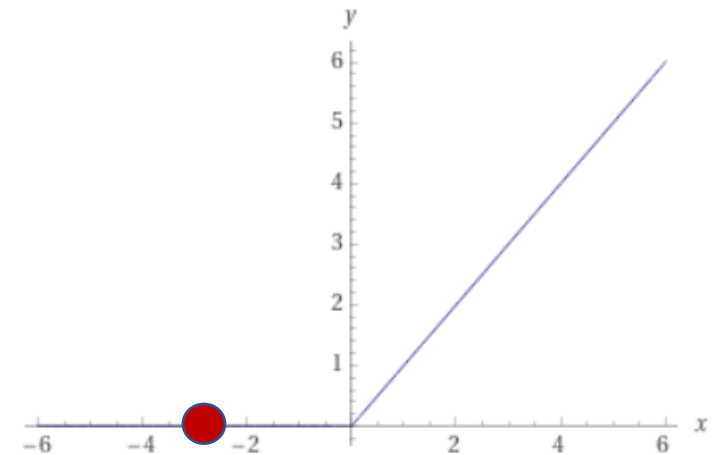
$$\text{ReLU}(z) = \max(z, 0)$$

ReluNode



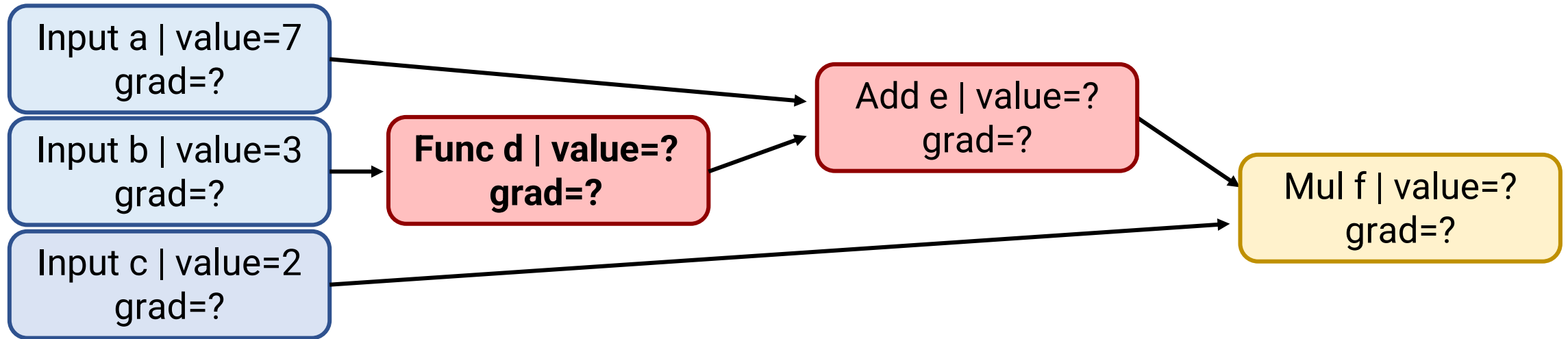
New function: $(a + \text{Relu}(b)) * c$ where $a=7$, $b=-3$, $c=2$

- Steps 1-3 are the same
- Step 4: Relu
 - $\partial(\text{Relu}(x))/\partial x = 1$ if $x > 0$, 0 if $x \leq 0$
 - If child > 0 , child.grad = parent.grad * 1
 - If child ≤ 0 , child.grad = 0



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

Generic Unary Function



New function: $(a + \text{Func}(b)) * c$ where $a=7, b=3, c=2$

- Steps 1-3 are the same
- Step 4: Func (generic function)
 - $\text{child.grad} = \text{parent.grad} * \partial(\text{Func}(\text{child}))/\partial \text{child}$

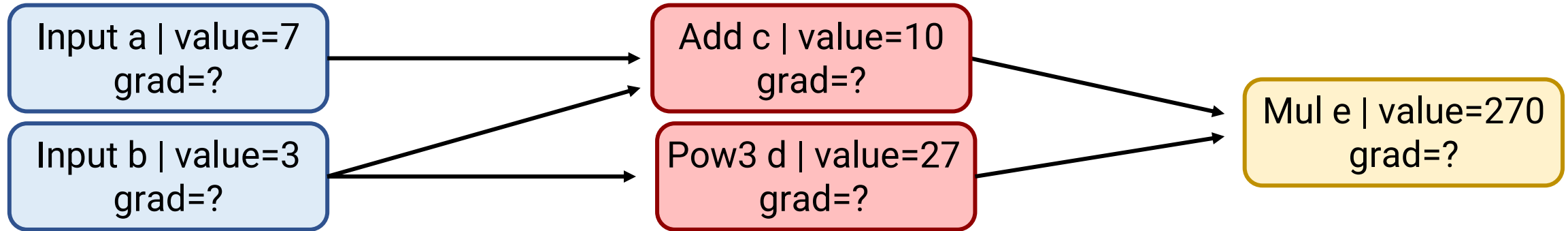
Announcements

- HW1 due today @ 11:59pm
- Oral exam sign-up (almost everyone has done it)
- Project proposals due Monday October 5
- Section Friday: Sci-kit Learn tutorial
- Notes on my schedule:
 - Next week I am skipping OH to make time for all the oral exams
 - If you want to discuss a project idea, you can email me to set up a custom time (or talk to TA/CP's)
 - Week of October 5, I will be traveling for a conference
 - Will teach class live on Zoom (may set up livestream in the classroom)
 - No office hours but again you can email to set up an appointment

Today's Plan

- The computation graph
- Backpropagation on trees
- Backpropagation on DAGs

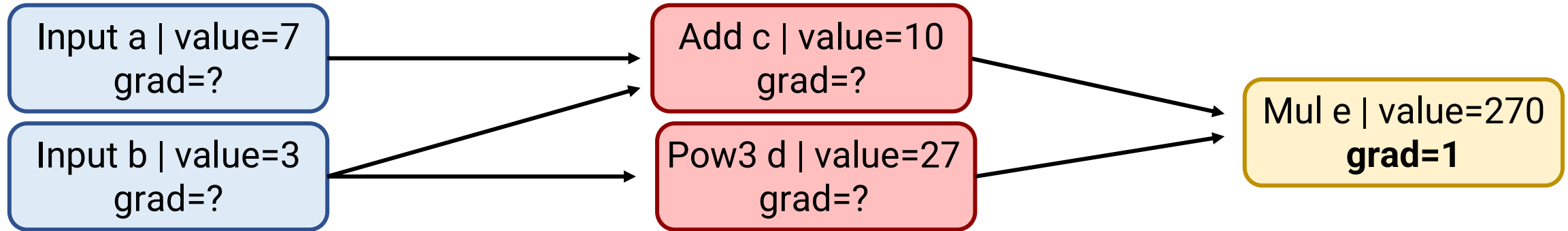
DAG Computation Graphs



New function: $(a + b) * b^3$ where $a=7, b=3$

- This is no longer a tree!
- Still a directed acyclic graph
- Let's see why our previous algorithm fails

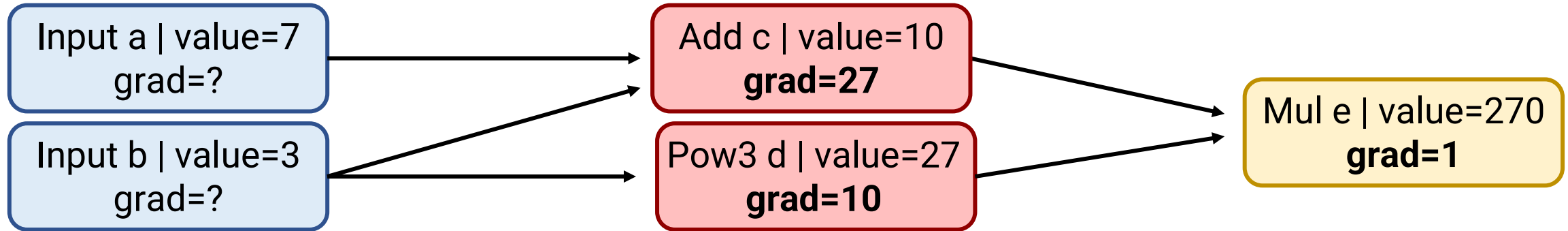
DAG Computation Graphs



New function: $(a + b) * b^3$ where $a=7, b=3$

- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b]$
- Step 1: Base case: $\partial e / \partial e = 1$

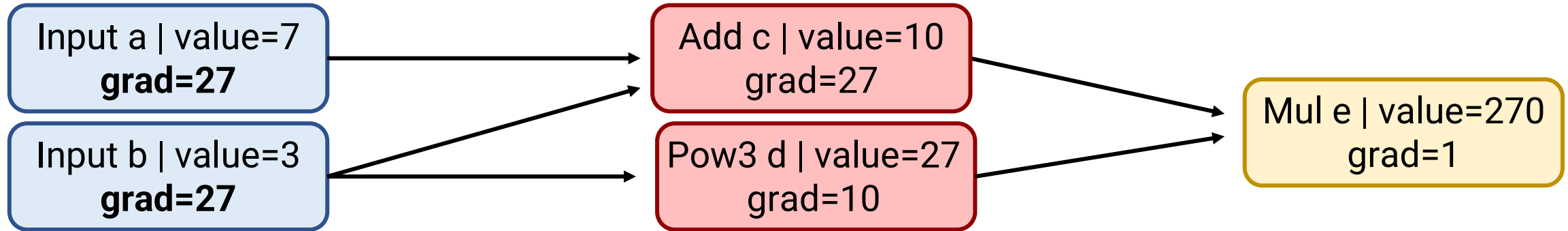
DAG Computation Graphs



New function: $(a + b) * b^3$ where $a=7, b=3$

- Goal: Compute gradient $[\partial f / \partial a, \partial f / \partial b]$
- Step 2: Distribute **Mul** (node e) gradient to children

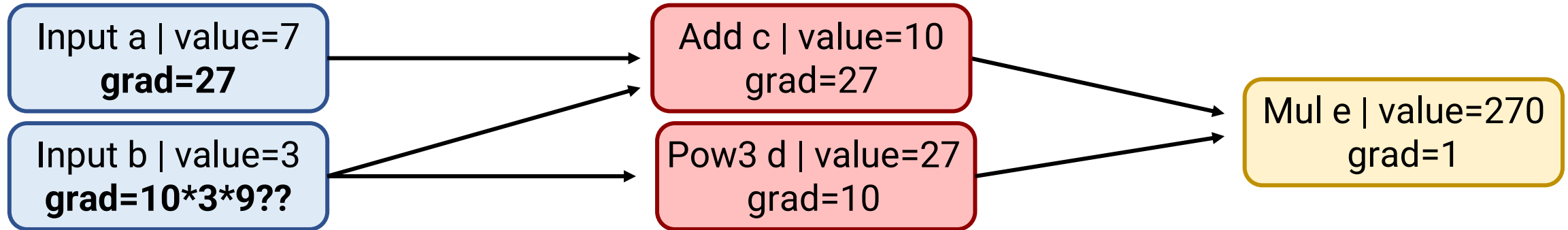
DAG Computation Graphs



New function: $(a + b) * b^3$ where $a=7, b=3$

- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b]$
- Step 3: Distribute **Add** (node c) gradient to children

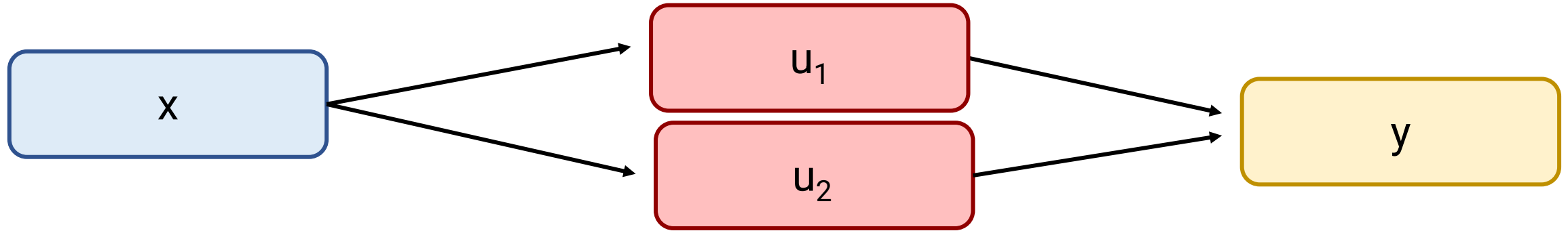
DAG Computation Graphs



New function: $(a + b) * b^3$ where $a=7, b=3$

- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b]$
- Step 4: Distribute **Pow3** (node d) gradient to child
 - By Chain Rule: Child gets **parent's gradient** * p * $child^{p-1}$
- Problem: We have overwritten the gradient from b to Add (node c)!

Multivariate chain rule



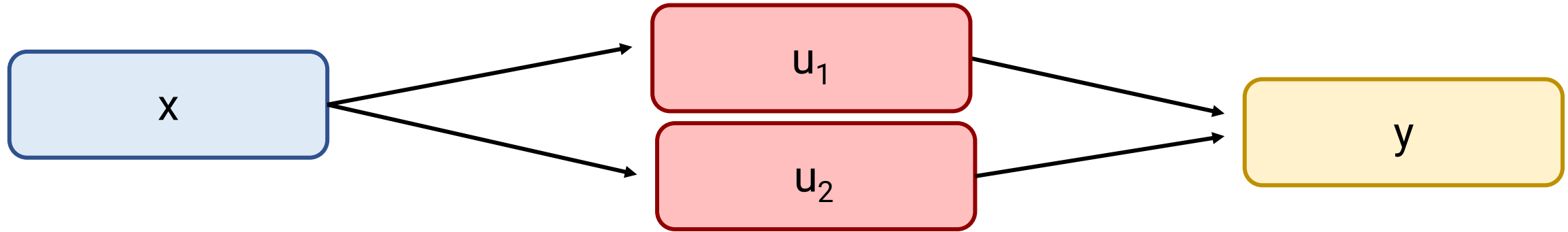
- Minimal example

- u_1 and u_2 depend on variable x (i.e., x has two parents)
- y depends on both u_1 and u_2

*Dot product of
 $[\partial y / \partial u_1, \partial y / \partial u_2]$ &
 $[\partial u_1 / \partial x, \partial u_2 / \partial x]$*

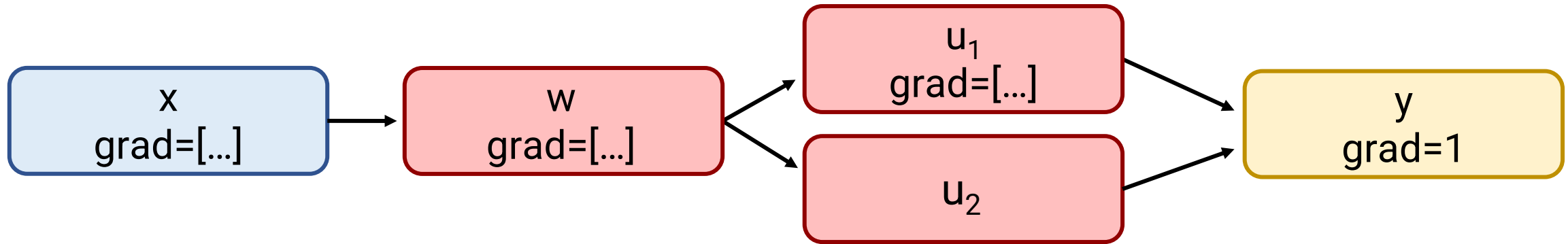
- By Multivariate Chain Rule: $\partial y / \partial x = \partial y / \partial u_1 * \partial u_1 / \partial x + \partial y / \partial u_2 * \partial u_2 / \partial x$
 - Changing x by epsilon changes each parent a bit; Effects add for small epsilon
 - Generalization of multiplying derivatives is matrix multiplication of Jacobians

Multivariate chain rule



- Minimal example
 - u_1 and u_2 depend on variable x (i.e., x has two parents)
 - y depends on both u_1 and u_2
- By Multivariate Chain Rule: $\partial y / \partial x = \partial y / \partial u_1 * \partial u_1 / \partial x + \partial y / \partial u_2 * \partial u_2 / \partial x$
- `node.grad` = **sum over parents** of `parent.grad` * $\partial(\text{parent}) / \partial(\text{child})$
- At each parent node, run `child.grad += parent.grad` * $\partial(\text{parent}) / \partial(\text{child})$

What order of traversal?

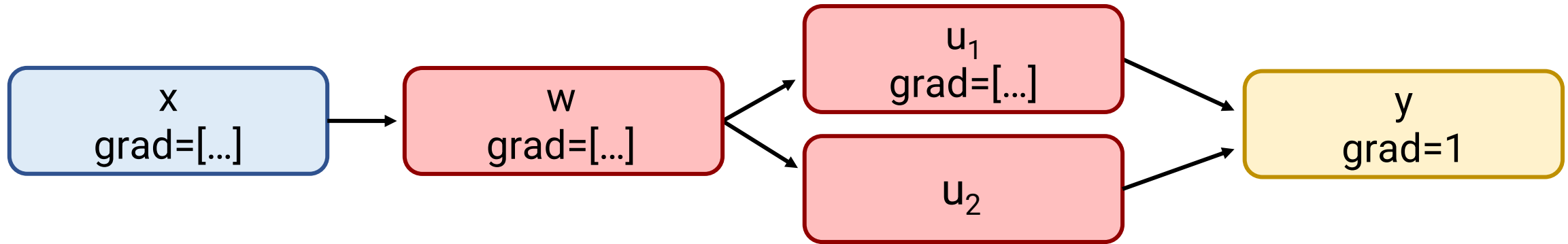


Current code:

- `y.backward()`
 - `u1.backward()`
 - `w.backward()`
 - `x.backward()`
 - `u2.backward()`
 - `w.backward()`
 - `x.backward()`

- Going recursively double-counts
 - First call to `w.backward()` makes final `x.grad` too large
- Solution: **Topological sort the nodes**
 - Iterate in reverse order, starting from output
 - Ensures that we process each node after all of its parents

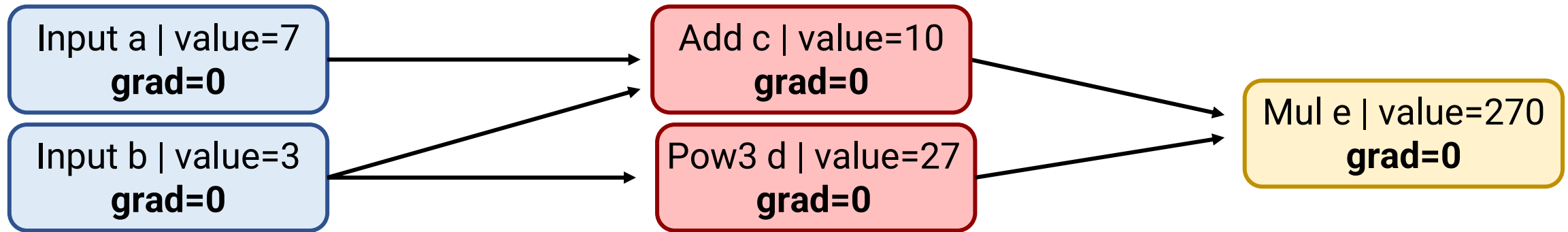
What order of traversal?



Better code:

- `topo_order = [x, w, u1, u2, y]`
- Iterate in reverse order:
 - `y.backward()`
 - `u2.backward()`
 - `u1.backward()`
 - `w.backward()`
 - `x.backward()`
- Going recursively double-counts
 - First call to `w.backward()` makes final `x.grad` too large
- Solution: **Topological sort the nodes**
 - Iterate in reverse order, starting from output
 - Ensures that we process each node after all of its parents

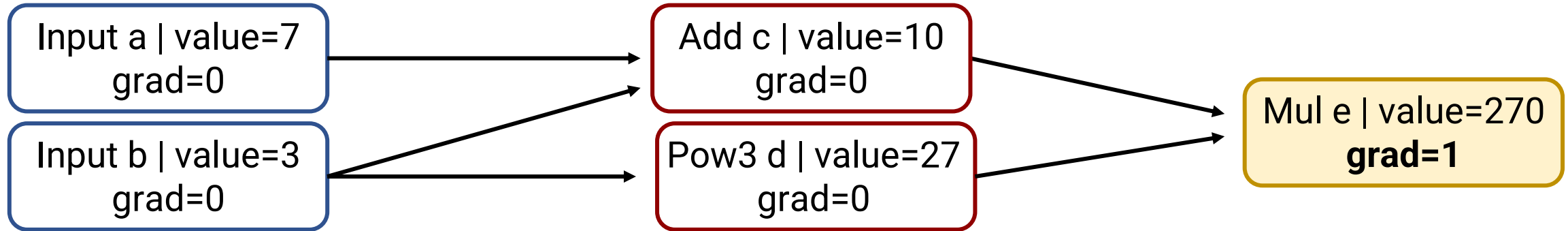
Backpropagation on DAGs



New function: $(a + b) * b^3$ where $a=7, b=3$

- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b]$
- Topological sort: $[a, b, c, d, e]$
- Step 0: Initialize all gradients to 0

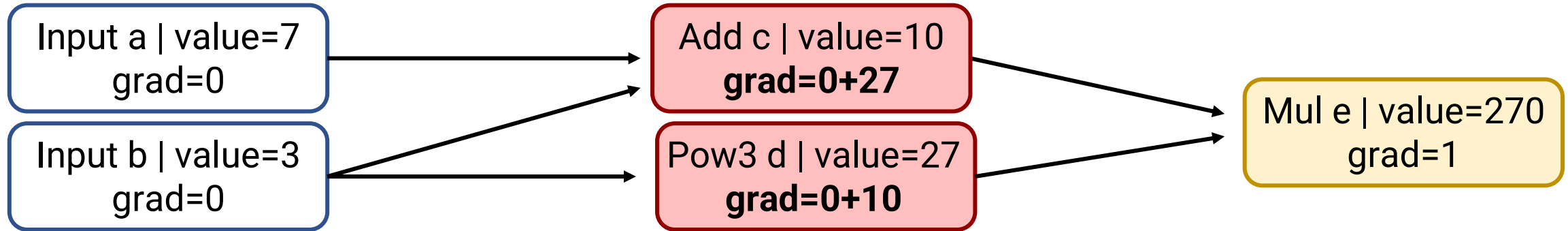
Backpropagation on DAGs



New function: $(a + b) * b^3$ where $a=7, b=3$

- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b]$
- Topological sort: $[a, b, c, d, e]$
- Step 1: Base case: $\partial e / \partial e = 1$

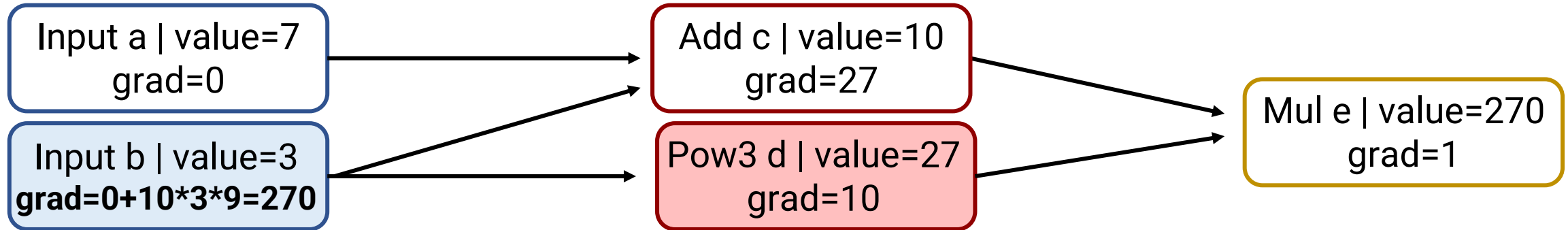
Backpropagation on DAGs



New function: $(a + b) * b^3$ where $a=7, b=3$

- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b]$
- Topological sort, reversed: $[e, d, c, b, a]$
- Step 2: Propagate **Mul** node e to children

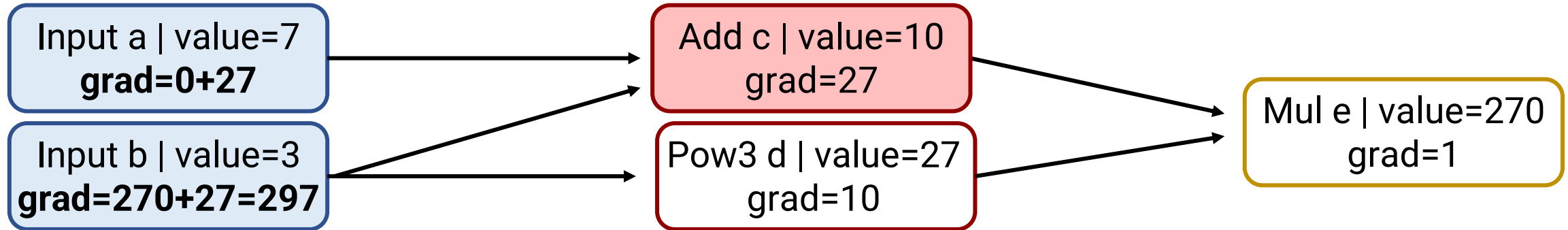
Backpropagation on DAGs



New function: $(a + b) * b^3$ where $a=7, b=3$

- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b]$
- Topological sort, reversed: $[e, d, c, b, a]$
- Step 3: Propagate **Pow3** node d to child

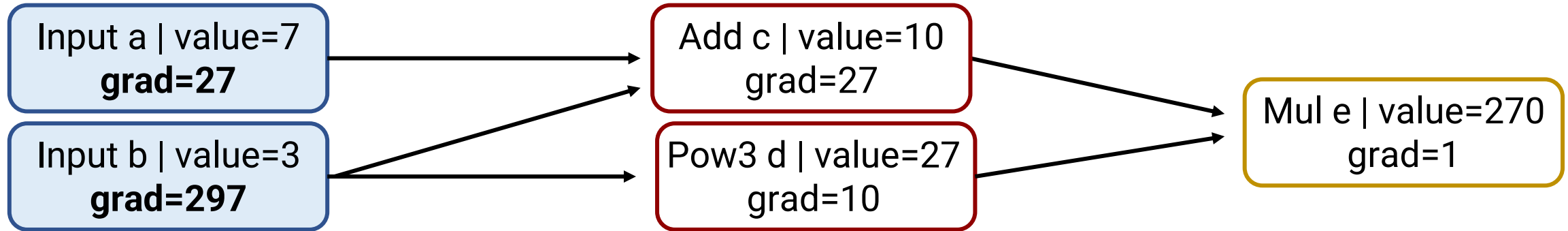
Backpropagation on DAGs



New function: $(a + b) * b^3$ where $a=7, b=3$

- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b]$
- Topological sort, reversed: $[e, d, c, b, a]$
- Step 4: Propagate **Add** node c to children

Backpropagation on DAGs



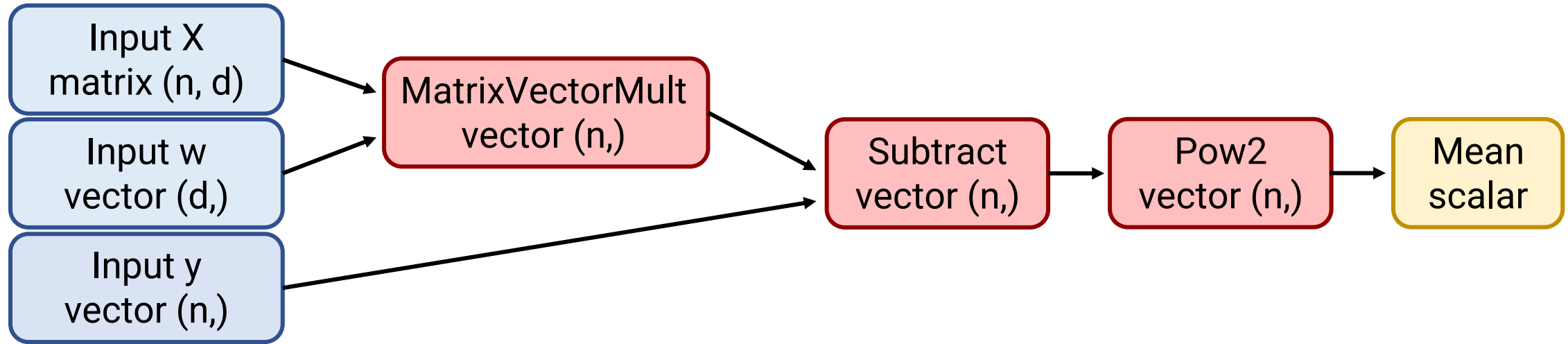
New function: $(a + b) * b^3$ where $a=7, b=3$

- Goal: Compute gradient $[\partial e / \partial a, \partial e / \partial b]$
- Topological sort, reversed: $[e, d, c, b, a]$
- Step 5, 6: `a.backward()`, `b.backward()` do nothing

Let's implement!



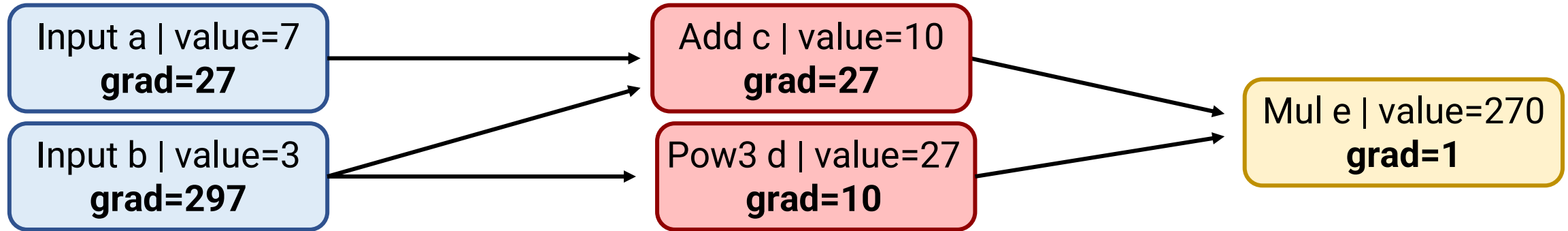
Backpropagation for vectors and matrices



Computation graph for $\text{mean}((Xw - y)^2)$, i.e. Linear Regression

- Basically the same, but each node can be a vector or matrix!
 - Each node.grad stores $\nabla_{\text{node}} \text{output}$
 - Parents know how to update $\nabla_{\text{child}} \text{output}$ based on $\nabla_{\text{parent}} \text{output}$

Conclusion



- Backpropagation computes gradient of output with respect to all nodes in computation graph
 - Forward pass: Compute values of all nodes
 - Backward pass: Iterate through nodes in reverse order,
At each parent node, run $\text{child.grad} += \text{parent.grad} * \frac{\partial(\text{parent})}{\partial(\text{child})}$
- Big picture: Makes it easy to run gradient descent on arbitrary computation graphs
 - Easy to try new architectures for neural networks