Implementation of Forward Mode AD I

- In the slides, we introduce how automatic differentiation can be implemented
- A corresponding technical report showing details is at `https://www.csie.ntu.edu.tw/~cjlin/papers/autodiff/`
- A sample implementation is also available at `https://github.com/ntumlgroup/simpleautodiff`
- For simplicity, we consider the forward mode. The reverse mode can be designed in a similar way

Implementation of Forward Mode AD II

- Consider a function $f : R^n \to R$ with

$$y = f(\boldsymbol{x}) = f(x_1, x_2, \ldots, x_n)$$

- For any given $\boldsymbol{x}$, we show the computation of

$$\frac{\partial y}{\partial x_1}$$

as an example

Calculating Function Values I

- We are calculating the derivative, so at the first glance, function values are not needed
- However, we show that it is necessary to calculate the function value
- The main reason is due to the function structure and the use of the chain rule

Calculating Function Values II

- To explain this, we begin with knowing that the function of a network is usually a nested composite function

$$f(\boldsymbol{x}) = h_k(h_{k-1}(\ldots h_1(\boldsymbol{x})))$$

  due to the layered structure
- To facilitate our discussion, let's assume that $f(\boldsymbol{x})$ is the following general composite function

$$f(\boldsymbol{x}) = g(h_1(\boldsymbol{x}), h_2(\boldsymbol{x}), \ldots, h_k(\boldsymbol{x}))$$

Calculating Function Values III

- For example, we see that the function considered earlier

$$f(x_1, x_2) = \ln x_1 + x_1 x_2 - \sin x_2 \tag{1}$$

can be written in the following composite function

$$g(h_1(x_1, x_2), h_2(x_1, x_2))$$

with

$$\begin{aligned} g(h_1, h_2) &= h_1 - h_2 \\ h_1(x_1, x_2) &= \ln x_1 + x_1 x_2 \\ h_2(x_1, x_2) &= \sin(x_2) \end{aligned}$$

Calculating Function Values IV

- To calculate the derivative at $x = x_0$ using the chain rule, we have

$$\frac{\partial f}{\partial x_1}\bigg|_{x=x_0} = \sum_{i=1}^{k} \left(\frac{\partial g}{\partial h_i}\bigg|_{h=h(x_0)} \times \frac{\partial h_i}{\partial x_1}\bigg|_{x=x_0} \right),$$

where the notation

$$\frac{\partial g}{\partial h_i}\bigg|_{h=h(x_0)}$$

means the derivative of g with respect to h_i evaluated at $h(x_0) = \begin{bmatrix} h_1(x_0) & \cdots & h_k(x_0) \end{bmatrix}^T$

Calculating Function Values V

- Clearly, we must calculate the inner function values $h_1(\boldsymbol{x}_0), \ldots, h_k(\boldsymbol{x}_0)$ first
- The process of computing all $h_i(\boldsymbol{x}_0)$ is part of (or almost the same as) the process of computing $f(\boldsymbol{x}_0)$
- This explanation tells why for calculating the partial derivatives, we need the function value first
- Next we discuss the implementation of getting the function value
- For the function (1), recall we have a table recording the order to get $f(x_1, x_2)$:

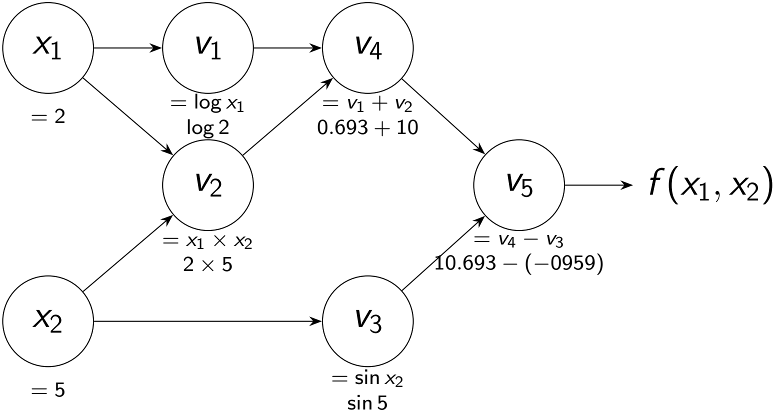
Calculating Function Values VI

$$
\begin{array}{lll}
x_1 & & = 2 \\
x_2 & & = 5 \\
\hline
v_1 & = \ln x_1 & = \ln 2 \\
v_2 & = x_1 \times x_2 & = 2 \times 5 \\
v_3 & = \sin x_2 & = \sin 5 \\
v_4 & = v_1 + v_2 & = 0.693 + 10 \\
v_5 & = v_4 - v_3 & = 10.693 + 0.959 \\
\hline
y & = v_5 & = 11.652
\end{array}
$$

Calculating Function Values VII

- Also, we have a computational graph to generate the computing order

Calculating Function Values VIII



- Therefore, we must check how to build the graph

Creating the Computational Graph I

- A graph consists of nodes and edges
- We must discuss what a node/edge is and how to store information
- From the graph shown above, we see that each node represents an intermediate expression:

$$
\begin{aligned}
v_1 &= \ln x_1 \\
v_2 &= x_1 \times x_2 \\
v_3 &= \sin x_2 \\
v_4 &= v_1 + v_2 \\
v_5 &= v_4 - v_3
\end{aligned}
$$

Creating the Computational Graph II

- The expression in each node is produced by applying an operation to expressions in other nodes
- Therefore, it's natural to construct an edge

  $$u \to v,$$

  if the expression of a node v is based on the expression of another node u
- We say node u is a parent node (of v) and node v is a child node (of u)
- To do the forward calculation, at node v we should store v's parents

Creating the Computational Graph III

- Additionally, we need to record the operator applied on the node's parents and the resulting value
- For example, the construction of the node

$$v_2 = x_1 \times x_2$$

  requires to store v_2's parent nodes $\{x_1, x_2\}$, the corresponding operator "$\times$" and the resulting value
- Up to now, we can implement each node as a class `Node` with the following members

Creating the Computational Graph IV

member	data type	example for `Node` v_2
numerical value	`float`	10
parent nodes	`List[Node]`	$[x_1, x_2]$
child nodes	`List[Node]`	$[v_4]$
operator	`string`	`"mul"` (for $\times$)

- At this moment, it is unclear why we should store child nodes in our `Node` class. Later we will explain why such information is needed
- Once the `Node` class is ready, starting from initial nodes (which represent x_i's), we use nested function calls to build the whole graph

Creating the Computational Graph V

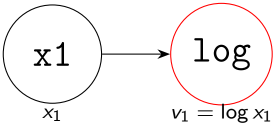
- In our case, the graph for $y = f(x_1, x_2)$ can be constructed via

```
y = sub(add(log(x1), mul(x1, x2)),
        sin(x2))
```

- Let's see this process step by step and check what each function must do

Creating the Computational Graph VI

- `log(x1)`:



- In our `log` function, a `Node` instance is created to store

$$\log(x_1).$$

  This node is the v_1 node in our computational graph

Creating the Computational Graph VII

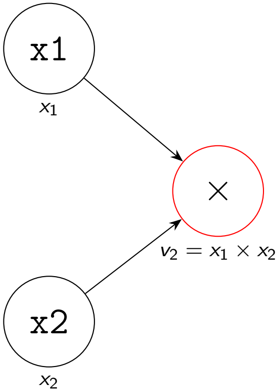
- To create this node, from the current log function and the input node x_1, we know contents of the following members
  - parent nodes: $[x_1]$
  - operator: `"log"`
  - numerical value: $\log 2$
- However, we have no information about children of this node
- The reason is obvious because we have not had a graph including its child nodes yet

Creating the Computational Graph VIII

- Instead, we leave this member "child nodes" empty and let child nodes to write back the information
- By this idea, our log function should add v_1 to the "child nodes" of x_1
- See more discussion later about "wrapping functions"

Creating the Computational Graph IX

- `mul(x1, x2)`

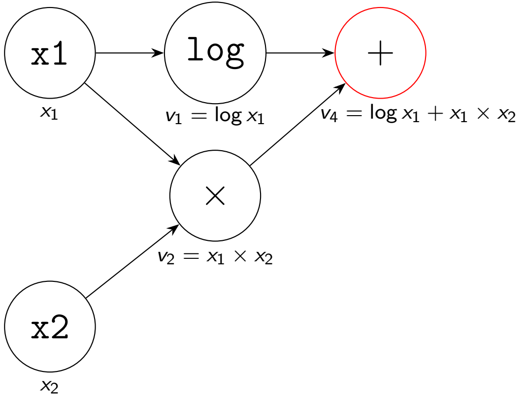


Creating the Computational Graph X

- Similarly, the `mul` function generates a `Node` instance. However, different from $\log(x_1)$, the node created here stores two parents (instead of one)

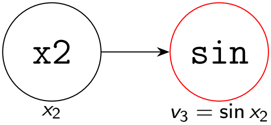
Creating the Computational Graph XI

- `add(log(x1), mul(x1, x2))`



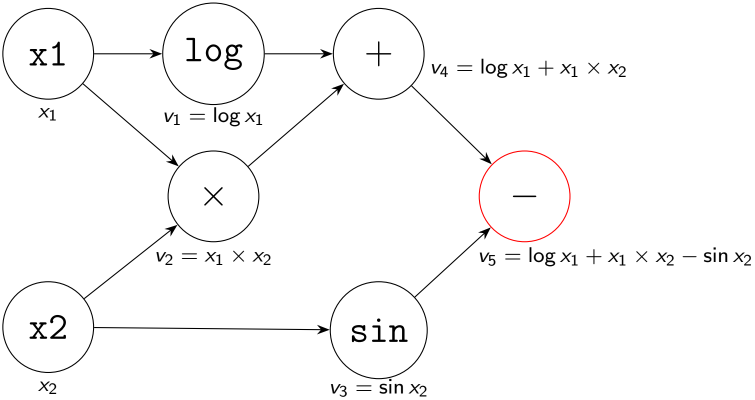
Creating the Computational Graph XII

- `sin(x2)`



Creating the Computational Graph XIII

- `sub(add(log(x1), mul(x1, x2)), sin(x2))`



Creating the Computational Graph XIV

- We can conclude that
  - each function generates exactly one `Node` instance;
  - however, the generated nodes differ in the operator, the number of parents, etc.

Wrapping Functions I

- We mentioned that a function like "`mul`" does more than calculating the product of two numbers. Here we show more details
- These customized functions "`add`", "`mul`" and "`log`" in the previous pages are *wrapping* functions
- Wrapping functions "wrap" numerical operations with additional codes
- Each must maintain the relation between the constructed node and its parents/children
- This way, the information of graph can be preserved

Wrapping Functions II

- For example, you may expect the following in the source code

```
def mul(node1, node2):
    value = node1.value * node2.value
    parent_nodes = [node1, node2]
    newNode = Node(value, parent_nodes, "mul")
    node1.child_nodes.append(newNode)
    node2.child_nodes.append(newNode)
    return newNode
```

- The created node is added to the "child nodes" lists of the two input nodes: `node1` and `node2`.

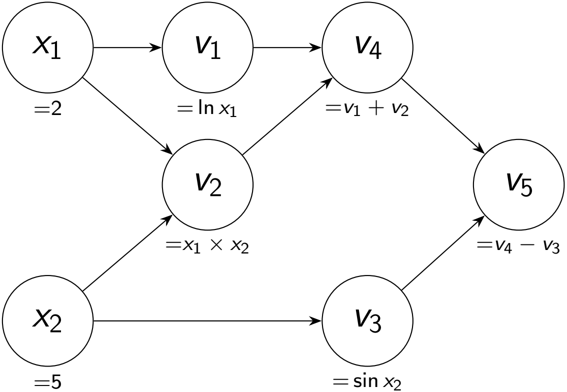
Wrapping Functions III

- As we mentioned earlier, when `node1` and `node2` were created, their lists of child nodes were empty. Each time a child node is created, it is appended to the list of its parent(s).
- The output of the function should be the created node. This setting enables the nested function call
- Then, calling `y = sub(...)` finishes the function evaluation. At the same time, we build the computational graph

Finding the Topological Order I

- We want to use the information in the graph to compute $\partial v_5 / \partial x_1$

Finding the Topological Order II



Finding the Topological Order III

- Recall that $\partial v / \partial x_1$ is denoted by $\dot{v}$
- From chain rule,

$$\dot{v}_5 = \frac{\partial v_5}{\partial v_4} \dot{v}_4 + \frac{\partial v_5}{\partial v_3} \dot{v}_3 \tag{2}$$

- We can see that

$$\frac{\partial v_5}{\partial v_4} \quad \text{and} \quad \frac{\partial v_5}{\partial v_3}$$

can be calculated at v_5 because we have information between v_5 and its parents v_4 and v_3. We will show details later

Finding the Topological Order IV

- Thus, the task we focus on now is to calculate $\dot{v}_4$ and $\dot{v}_3$
- For $\dot{v}_4$, we further have

$$\dot{v}_4 = \frac{\partial v_4}{\partial v_1}\dot{v}_1 + \frac{\partial v_4}{\partial v_2}\dot{v}_2, \tag{3}$$

  so $\dot{v}_1$ and $\dot{v}_2$ are needed
- On the other hand, we have $\dot{v}_3 = 0$ since the expression for v_3

$$\sin(x_2)$$

  is not a function of x_1

Finding the Topological Order V

- From this example, we find that

$$v \text{ is not reachable from } x_1 \Rightarrow \dot{v} = 0$$

- We say a node v is reachable from a node u if there exists a path from u to v in the graph
- Therefore, now we only care about nodes reachable from x_1
- From (2) and (3), we see that nodes reachable from x_1 must be properly ordered so that, for example, in (2), $\dot{v}_4$ and $\dot{v}_3$ are ready before calculating $\dot{v}_5$

Finding the Topological Order VI

- To consider nodes reachable from x_1, from the whole computational graph $G = \langle V, E \rangle$, where V and E are respectively sets of nodes and edges, we define

$$V_R = \{v \in V \mid v \text{ is reachable from } x_1\},$$

$$E_R = \{(u, v) \in E \mid u \in V_R,\ v \in V_R\}$$

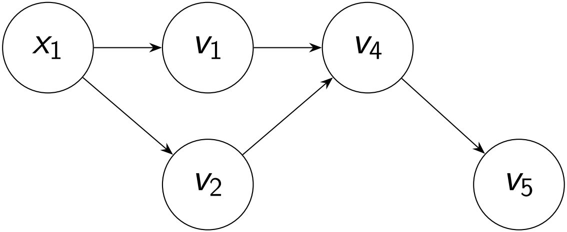
- Then,

$$G_R \equiv \langle V_R, E_R \rangle$$

is a subgraph of G

Finding the Topological Order VII

- For our example, G_R is the following subgraph



$$V_R = \{x_1, v_1, v_2, v_4, v_5\}$$
$$E_R = \{(x_1, v_1), (x_2, v_2), (v_1, v_4), (v_2, v_4), (v_4, v_5)\}$$

Finding the Topological Order VIII

- We aim to find a "suitable" ordering of V_R satisfying that each node $u \in V_R$ comes before all of its child nodes in the ordering
- By doing so, $\dot{u}$ can be used in the derivative calculation of its child nodes; see (3)
- For our example, a "suitable" ordering can be

$$x_1, v_1, v_2, v_4, v_5$$

- In graph theory, such an ordering is called a *topological ordering* of G_R

Finding the Topological Order IX

- Since G_R is a directed acyclic graph (DAG), a topological ordering must exist
- We may use depth first search (DFS) to traverse G_R to find the topological ordering
- Earlier we did not explain why a member "child nodes" is needed in the `Node` class. Here we see why
- To traverse G_R from x_1, we must access children of each node

Finding the Topological Order X

- Here is an implementation

```
def topological_order(rootNode):
    def add_children(node):
        if node not in visited:
            visited.add(node)
            for child in node.child_nodes:
                add_children(child)
            ordering.append(node)
    ordering, visited = [], set()
    add_children(rootNode)
    return list(reversed(ordering))
```

Finding the Topological Order XI

- The root node of G_R is x_1. We put it as the input of the `add_children` function
- The subroutine recursively explores all nodes reachable from the input node and appends the input node to the end
- Also, we must maintain a set of visited nodes to ensure that each node is included in the ordering exactly once

Finding the Topological Order XII

- For our example, the depth-first search has

$$x_1 \rightarrow v_1 \rightarrow v_4 \rightarrow v_5,$$

  so v_5 is added first. In the end, we get the following list

$$[v_5, v_4, v_1, v_2, x_1]$$

- Then, by reversing the list, a node always comes before its children
- Methods based on the topological ordering are called *tape-based* methods

Finding the Topological Order XIII

- They are used in some real-world implementations such as Tensorflow
- The ordering is regarded as a tape. We're going to read the nodes one by one from the beginning of the sequence (tape) to calculate the derivative value
- Based on the obtained ordering, let's see how to compute each $\dot{v}$

Computing the Partial Derivative I

- By the chain rule, we have

$$\dot{v} = \sum_{u \in v\text{'s parents}} \frac{\partial v}{\partial u} \dot{u}$$

- If we calculate the derivative according to the topological order, the second term

$$\dot{u} = \frac{\partial u}{\partial x_1}$$

should be readily available when we're computing $\dot{v}$

Computing the Partial Derivative II

- Therefore, all we need is to check the calculation of the first term
$$\frac{\partial v}{\partial u}$$
- At v, we know that u is one of its parent(s). We further know the operation involving v's parent(s)
- For example, we have $v_4 = v_1 \times v_2$, so
$$\frac{\partial v_4}{\partial v_1} = v_2 \text{ and } \frac{\partial v_4}{\partial v_2} = v_1$$
These values can be computed and stored when we construct the computational graph

Computing the Partial Derivative III

- Therefore, we add a member "gradient w.r.t. parents" to our `Node` class
- Also we add a member "partial derivative" to store the partial derivative with respect to x_1

member	data type	example for `Node` v_2
numerical value	`float`	10
parent nodes	`List[Node]`	$[x_1, x_2]$
child nodes	`List[Node]`	$[v_4]$
operator	`string`	`"mul"`
gradient w.r.t parents	`List[float]`	$[5, 2]$
partial derivative	`float`	5

Computing the Partial Derivative IV

- We update the `mul` function accordingly

```
def mul(node1, node2):
    value = node1.value * node2.value
    parent_nodes = [node1, node2]
    newNode = Node(value, parent_nodes, "mul")
    newNode.grad_wrt_parents = [node2.value,node1.value]
    node1.child_nodes.append(newNode)
    node2.child_nodes.append(newNode)
    return newNode
```

Computing the Partial Derivative V

- As shown above, we must compute

$$\frac{\partial\ \texttt{newNode}}{\partial\ \texttt{parentNode}}$$

  for each parent node in constructing a new child node
- Here are some examples other than the `mul` function

Computing the Partial Derivative VI

- `add(node1, node2)`: we have

$$\frac{\partial \texttt{ newNode}}{\partial \texttt{ node1}} = \frac{\partial \texttt{ newNode}}{\partial \texttt{ node2}} = 1,$$

so the red line is replaced by

```
newNode.grad_wrt_parents = [1., 1.]
```

Computing the Partial Derivative VII

- `log(node)`: we have

$$\frac{\partial\ \texttt{newNode}}{\partial\ \texttt{node}} = \frac{1}{\texttt{node.value}},$$

so the red line becomes

```
newNode.grad_wrt_parents = [1/node.value]
```

Computing the Partial Derivative VIII

- Now, we know how to get each term in the chain rule for calculating $\dot{v}$:

$$\dot{v} = \sum_{u \in v\text{'s parents}} \frac{\partial v}{\partial u} \dot{u}$$

- Therefore if we follow the topological ordering, all $\dot{v}$ (i.e., partial derivatives with respect to x_1) can be calculated

Computing the Partial Derivative IX

- An implementation to compute the partial derivatives is as follows

```
def forward(rootNode):
    rootNode.partial_derivative = 1
    ordering = topological_order(rootNode)
    for node in ordering[1:]:
        partial_derivative = 0
        for i in range(len(node.parent_nodes)):
            dnode_dparent = node.grad_wrt_parents[i]
            dparent_droot = node.parent_nodes[i].partial_derivative
            partial_derivative += dnode_dparent * dparent_droot
        node.partial_derivative = partial_derivative
```

- We store the resulting value in the member `partial_derivative` of each node

Summary I

- The procedure for forward mode includes three steps:
  1. Create the computational graph
  2. Find a topological order of the graph associated with x_1
  3. Compute the partial derivative with respect to x_1 along the topological order
- We discuss not only how to run each step but also what information we should store
- This is a minimal implementation to show you all details of the forward mode