A Step-by-step Introduction to the Implementation of Automatic Differentiation

Yu-Hsueh Fang*[1], He-Zhe Lin*[1], Jie-Jyun Liu[1], and Chih-Jen Lin[1,2]

[1]National Taiwan University
{d11725001, r11922027, d11922012}@ntu.edu.tw
cjlin@csie.ntu.edu.tw
[2]Mohamed bin Zayed University of Artificial Intelligence
chihjen.lin@mbzuai.ac.ae

November 18, 2025

Abstract

Automatic differentiation is a key component in deep learning. This topic is well studied and excellent surveys such as **?** have been available to clearly describe the basic concepts. Further, sophisticated implementations of automatic differentiation are now an important part of popular deep learning frameworks. However, it is difficult, if not impossible, to directly teach students the implementation of existing systems due to the complexity. On the other hand, if the teaching stops at the basic concept, students fail to sense the realization of an implementation. For example, we often mention the computational graph in teaching automatic differentiation, but students wonder how to implement and use it. In this document, we partially fill the gap by giving a step by step introduction of implementing a simple automatic differentiation system. We streamline the mathematical concepts and the implementation. Further, we give the motivation behind each implementation detail, so the whole setting becomes very natural.

1 Introduction

In modern machine learning, derivatives are the cornerstone of numerous applications and studies. The calculation often relies on automatic differentiation, a classic method for efficiently and accurately calculating derivatives of numeric functions. For example, deep learning cannot succeed without automatic differentiation. Therefore, teaching students how automatic differentiation works is highly essential.

Automatic differentiation is a well-developed area with rich literature. Excellent surveys including **?**, **?**, **?** and **?** review the algorithms for automatic differentiation and its wide applications. In

*These authors contributed equally to this work

particular, **?** is a comprehensive work focusing on automatic differentiation in machine learning. Therefore, there is no lack of materials introducing the concept of automatic differentiation.

On the other hand, as deep learning systems now solve large-scale problems, it is inevitable that the implementation of automatic differentiation becomes highly sophisticated. For example, in popular deep learning systems such as PyTorch (**?**) and Tensorflow (**?**), at least thousands of lines of code are needed. Because of this, many places of teaching automatic differentiation for deep learning stop at the basic concepts. Then students fail to sense the realization of an implementation. For example, we often mention the computational graph in teaching automatic differentiation, but students wonder how to implement and use it. In this document, we aim to partially fill the gap by giving a tutorial on the basic implementation.

In recent years, many works[1,2,3,4,5] have attempted to discuss the basic implementation of automatic differentiation. However, they still leave room for improvement. For example, some are not self-contained – they quickly talk about implementations without connecting to basic concepts. Ours, which is very suitable for the beginners, has the following features:

- We streamline the mathematical concepts and the implementation. Further, we give the motivation behind each implementation detail, so the whole setting becomes very natural.
- We use the example from **?** for the consistency with past works. Ours is thus an extension of **?** into the implementation details.
- We build a complete tutorial including this document, slides and the source code at `https://www.csie.ntu.edu.tw/~cjlin/papers/autodiff/`.

2 Automatic Differentiation

There are two major modes of automatic differentiation. In this section, we introduce the basic concepts of both modes. Most materials in this section are from **?**. We consider the same example function

$$y = f(x_1, x_2) = \log x_1 + x_1 x_2 - \sin x_2.$$

2.1 Forward Mode

First, we discuss the forward mode. Before calculating the derivative, let us check how to calculate the function value. Assume that we want to calculate the function value at $(x_1, x_2) = (2, 5)$. Then,

[1] `https://towardsdatascience.com/build-your-own-automatic-differentiation-program-6ecd585eec2a`

[2] `https://sidsite.com/posts/autodiff`

[3] `https://mdrk.io/introduction-to-automatic-differentiation-part2/`

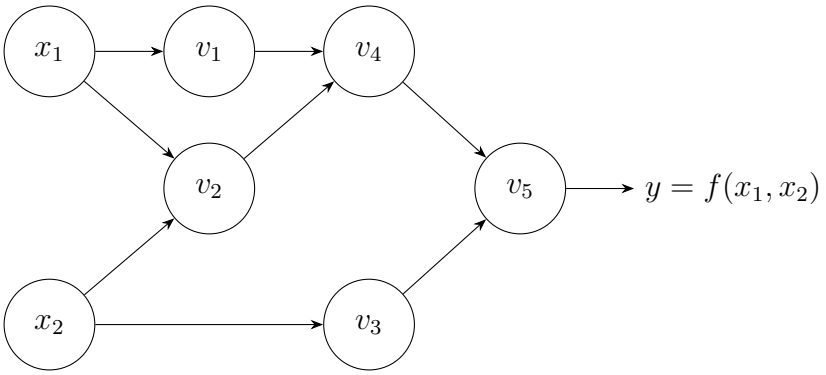
[4] `https://github.com/dlsyscourse/lecture5/blob/main/5_automatic_differentiation_implementation.ipynb`

[5] `https://github.com/karpathy/micrograd`

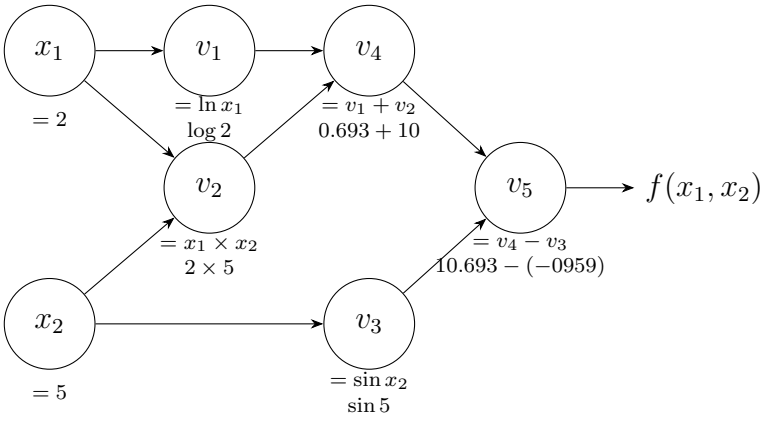
in the following table, we have a forward procedure.

x_1		$= 2$
x_2		$= 5$
v_1	$= \log x_1$	$= \log 2$
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	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

We use variables v_i to record the intermediate outcomes. First, we know log function is applied to x_1. Therefore, we have $\log(x_1)$ as a new variable called v_1. Similarly, there is a variable v_2, which is $x_1 \times x_2$. Each v_i is related to a simple operation. The initial value of this member is zero when a node is created. In the end, our function value at $(2, 5)$ is $y = v_5$. As shown in the table, the function evaluation is decomposed into a sequence of simple operations. We have a corresponding computational graph as follows:



Because calculating both v_1 and v_2 needs x_1, x_1 has two links to them in the graph. The following graph shows all the intermediate results in the computation.



The computational graph tells us the dependencies of variables. Thus, from the inputs x_1 and x_2 we can go through all nodes for getting the function value $y = v_5$ in the end.

Now, we have learned about the function evaluation. But remember, we would like to calculate the derivative. Assume that we target at the partial derivative $\partial y/\partial x_1$. Here, we denote

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

as the derivative of the variable v with respect to x_1. The idea is that by using the chain rule, we can obtain the following forward derivative calculation to eventually get $\dot{v}_5 = \partial f/\partial x_1$.

$\dot{x}_1$	$= \partial x_1/\partial x_1$	$= 1$
$\dot{x}_2$	$= \partial x_2/\partial x_1$	$= 0$
$\dot{v}_1$	$= \dot{x}_1/x_1$	$= 1/2$
$\dot{v}_2$	$= \dot{x}_1 \times x_2 + \dot{x}_2 \times x_1$	$= 1 \times 5 + 0 \times 2$
$\dot{v}_3$	$= \dot{x}_2 \times \cos x_2$	$= 0 \times \cos 5$
$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$\dot{y}$	$= \dot{v}_5$	$= 5.5$

The table starts from $\dot{x}_1$ and $\dot{x}_2$, which are $\partial x_1/\partial x_1 = 1$ and $\partial x_2/\partial x_1 = 0$. Based on $\dot{x}_1$ and $\dot{x}_2$, we can calculate other values. For example, let us check the partial derivative $\partial v_1/\partial x_1$. From

$$v_1 = \log x_1,$$

by the chain rule,

$$\dot{v}_1 = \frac{\partial v_1}{\partial x_1} \times \frac{\partial x_1}{\partial x_1} = \frac{1}{x_1} \times \frac{\partial x_1}{\partial x_1} = \frac{\dot{x}_1}{x_1}.$$

Therefore, we need $\dot{x}_1$ and x_1 for calculating $\dot{v}_1$ on the left-hand side. We already have the value of $\dot{x}_1$ from the previous step ($\partial x_1/\partial x_1 = 1$). Also, the function evaluation gives the value of x_1. Then, we can calculate $\dot{x}_1/x_1 = 1/2$. Clearly, the chain rule plays an important role here. The calculation of other $\dot{v}_i$ is similar.

2.2 Reverse Mode

Next, we discuss the reverse mode. We denote

$$\bar{v} = \frac{\partial y}{\partial v}$$

as the derivative of the function y with respect to the variable v. Note that earlier, in the forward mode, we considered

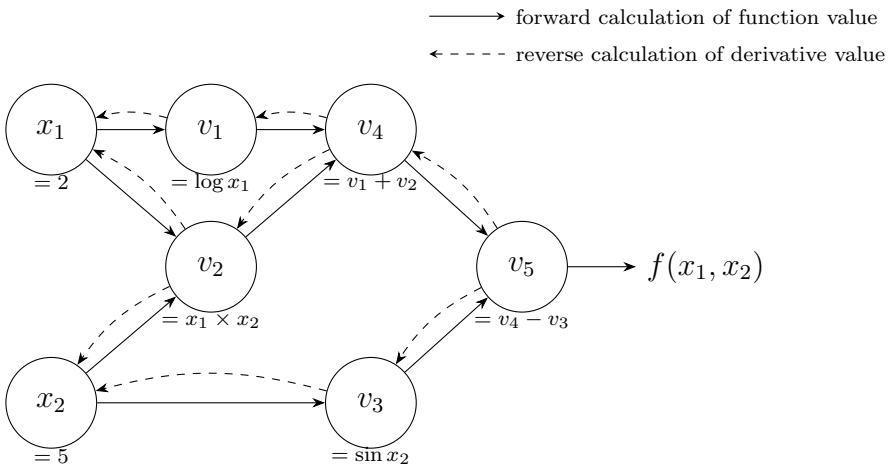
$$\dot{v} = \frac{\partial v}{\partial x_1},$$

so the focus is on the derivatives of all variables with respect to one input variable. In contrast, the reverse mode focuses on $\bar{v} = \partial y/\partial v$ for all v, the partial derivatives of one output with respect to

all variables. Therefore, for our example, we can use $\bar{v}_i$'s and $\bar{x}_i$'s to get both $\partial y/\partial x_1$ and $\partial y/\partial x_2$ at once. Now we illustrate the calculation of

$$\frac{\partial y}{\partial x_2}.$$

By checking the variable x_2 in the computational graph, we see that variable x_2 affects y by affecting v_2 and v_3.



This dependency, together with the fact that x_1 is fixed, means if we would like to calculate $\partial y/\partial x_2$, then it is equal to calculate

$$\frac{\partial y}{\partial x_2} = \frac{\partial y}{\partial v_2}\frac{\partial v_2}{\partial x_2} + \frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial x_2}. \tag{1}$$

We can rewrite Equation 1 as follows with our notation.

$$\bar{x}_2 = \bar{v}_2\frac{\partial v_2}{\partial x_2} + \bar{v}_3\frac{\partial v_3}{\partial x_2}. \tag{2}$$

If $\bar{v}_2$ and $\bar{v}_3$ are available beforehand, all we need is to calculate $\partial v_2/\partial x_2$ and $\partial v_3/\partial x_2$. From the operation between x_2 and v_3, we know that $\partial v_3/\partial x_2 = \cos(x_2)$. Similarly, we have $\partial v_2/\partial x_2 = x_1$. Then, the evaluation of $\bar{x}_2$ is done in two steps:

$$\bar{x}_2 \leftarrow \bar{v}_3\frac{\partial v_3}{\partial x_2}$$
$$\bar{x}_2 \leftarrow \bar{x}_2 + \bar{v}_2\frac{\partial v_2}{\partial x_2}.$$

These steps are part of the sequence of a reverse traversal, shown in the following table.

$\bar{x}_1$			$= 5.5$
$\bar{x}_2$			$= 1.716$
$\bar{x}_1$	$= \bar{x}_1 + \bar{v}_1 \frac{\partial v_1}{\partial x_1}$	$= \bar{x}_1 + \bar{v}_1 / x_1$	$= 5.5$
$\bar{x}_2$	$= \bar{x}_2 + \bar{v}_2 \frac{\partial v_2}{\partial x_2}$	$= \bar{x}_2 + \bar{v}_2 \times x_1$	$= 1.716$
$\bar{x}_1$	$= \bar{v}_2 \frac{\partial v_2}{\partial x_1}$	$= \bar{v}_2 \times x_2$	$= 5$
$\bar{x}_2$	$= \bar{v}_3 \frac{\partial v_3}{\partial x_2}$	$= \bar{v}_3 \times \cos x_2$	$= -0.284$
$\bar{v}_2$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5$	$= \bar{y}$	$= 1$	

To get the desired $\bar{x}_1$ and $\bar{x}_2$ (i.e., $\partial y/\partial x_1$ and $\partial y/\partial x_2$), we begin with

$$\bar{v}_5 = \frac{\partial y}{\partial v_5} = \frac{\partial y}{\partial y} = 1.$$

From the computational graph, we then get $\bar{v}_4$ and $\bar{v}_3$. Because v_4 affects y only through v_5, we have

$$\bar{v}_4 = \frac{\partial y}{\partial v_4} = \frac{\partial y}{\partial v_5}\frac{\partial v_5}{\partial v_4} = \bar{v}_5 \frac{\partial v_5}{\partial v_4} = \bar{v}_5 \times 1$$

The above equation is based on that we already know $\partial y/\partial v_5 = \bar{v}_5$. Also, the operation from v_4 to v_5 is an addition, so $\partial v_5/\partial v_4$ is a constant 1. By such a sequence, in the end, we obtain

$$\frac{\partial y}{\partial x_1} = \bar{x}_1 \text{ and } \frac{\partial y}{\partial x_2} = \bar{x}_2$$

at the same time.

3 Implementation of Function Evaluation and the Computational Graph

With the basic concepts ready in Section 2, we move to the implementation of the automatic differentiation. 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.

3.1 The Need to Calculate Function Values

We are calculating the derivative, so at the first glance, function values are not needed. However, we show that function evaluation is necessary due to the function structure and the use of the chain rule. To explain this, we begin with knowing that the function of a neural network is usually a nested composite function

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

owing to the layered structure. For an easy discussion, let us assume that $f(\boldsymbol{x})$ is the following general composite function

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

We can see that the example function considered earlier

$$f(x_1, x_2) = \log x_1 + x_1 x_2 - \sin x_2 \tag{3}$$

can be written as the following composite function

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

with

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

To calculate the derivative at $\boldsymbol{x} = \boldsymbol{x}_0$ using the chain rule, we have

$$\frac{\partial f}{\partial x_1}\Big|_{\boldsymbol{x}=\boldsymbol{x}_0} = \sum_{i=1}^{k} \left(\frac{\partial g}{\partial h_i}\Big|_{\boldsymbol{h}=\boldsymbol{h}(\boldsymbol{x}_0)} \times \frac{\partial h_i}{\partial x_1}\Big|_{\boldsymbol{x}=\boldsymbol{x}_0} \right),$$

where the notation

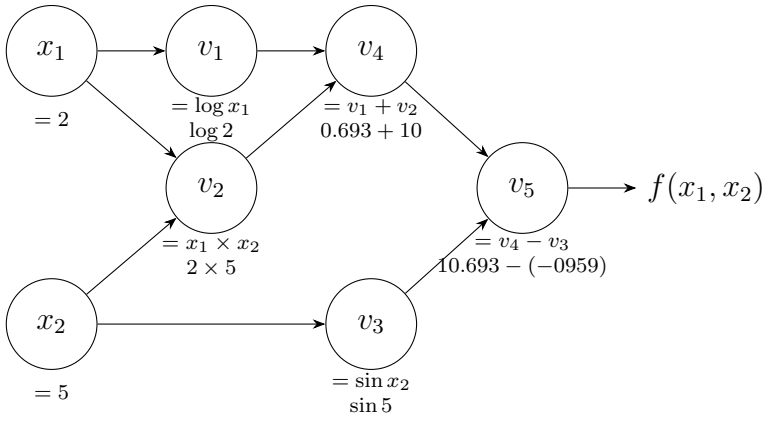
$$\frac{\partial g}{\partial h_i}\Big|_{\boldsymbol{h}=\boldsymbol{h}(\boldsymbol{x}_0)}$$

means the derivative of g with respect to h_i evaluated at $\boldsymbol{h}(\boldsymbol{x}_0) = \begin{bmatrix} h_1(\boldsymbol{x}_0) & \cdots & h_k(\boldsymbol{x}_0) \end{bmatrix}^T$. Clearly, we must calculate the values of the inner function values $h_1(\boldsymbol{x}_0), \dots, h_k(\boldsymbol{x}_0)$ 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 (3), recall that we have a table recording the order to get $f(x_1, x_2)$:

x_1		$= 2$
x_2		$= 5$
v_1	$= \log x_1$	$= \log 2$
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	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

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



Therefore, we must check how to build the graph.

3.2 Creating the Computational Graph

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:

$$
\begin{aligned}
v_1 &= \log x_1, \\
v_2 &= x_1 \times x_2, \\
v_3 &= \sin x_2, \\
v_4 &= v_1 \times v_2, \\
v_5 &= v_4 - v_3.
\end{aligned}
$$

The expression in each node is an operation on expressions from other nodes. Therefore, it is 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, we should store the'

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

$$v_2 = x_1 \times x_2,$$

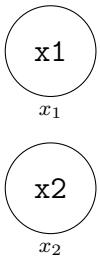
requires to store v_2's parent nodes $\{x_1, x_2\}$, the corresponding operator "$\times$" and the resulting value. Up to now, we can implement each node as a class `Node` with the following members.

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 $\times$)

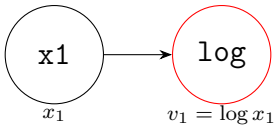
At this moment, it is unclear why we should store child nodes in our `Node` class. Later we will explain why such information is needed. Once the `Node` class is ready, starting from initial nodes (which represent x_i's), we use nested function calls to build the whole graph. In our case, the graph for $y = f(x_1, x_2)$ can be constructed via

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

let us see this process step by step and check what each function must do. First, our starting point is the root nodes created by the `Node` class constructor.



These root `Nodes` have empty members "parent nodes," "child nodes," and "operator" with only "numerical value" respectively set to x_1 and x_2. Then, we apply our implemented `log(node)` to the node `x1`.

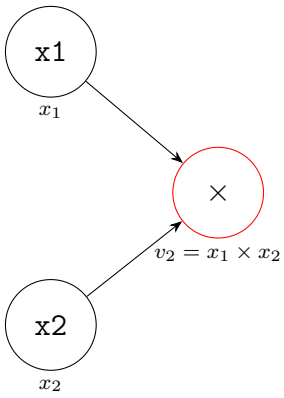


The implementation of our `log` function should create a `Node` instance to store $\log(x_1)$. Therefore, what we have is a wrapping function that does more than the log operation; see details in Section 3.3. The created node is the v_1 node in our computational graph. Next, we discuss details of the node creation. From the current log function and the input node x_1, we know contents of the following members:

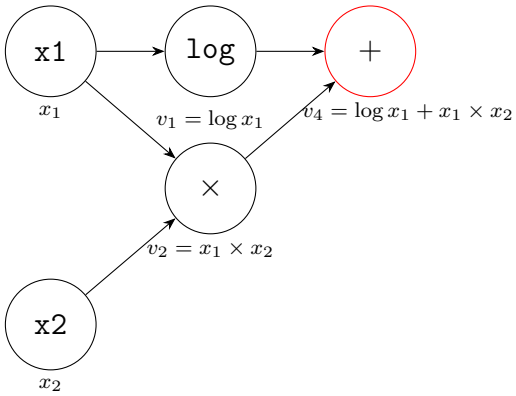
- parent nodes: $[x_1]$
- operator: `"log"`
- numerical value: $\log 2$

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. 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 details later in Section 3.3.

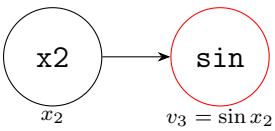
We move on to apply `mul(node1, node2)` on nodes `x1` and `x2`.



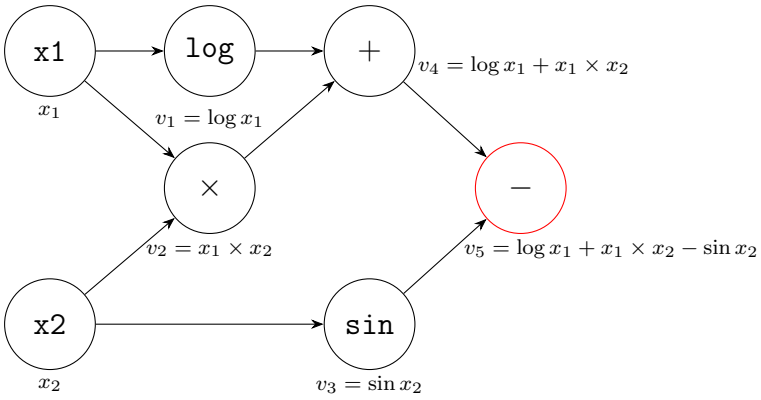
Similarly, the `mul` function generates a `Node` instance. However, different from `log(x1)`, the node created here stores two parents (instead of one). Then we apply the function call `add(log(x1), mul(x1, x2))`.



Next, we apply `sin(node)` to `x2`.



Last, applying `sub(node1,node2)` to the output nodes of `add(log(x1), mul(x1, x2))` and `sin(x1)` leads to



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

3.3 Wrapping Functions

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, which "wrap" numerical operations with additional codes. An important task is to maintain the relation between the constructed node and its parents/children. This way, the information of graph can be preserved.

For example, we may implement the following "`mul`" function.

Listing 1: The wrapping function "`mul`"

```
    newNode.grad_wrt_parents = [1, -1]
    node1.child_nodes.append(newNode)
    node2.child_nodes.append(newNode)
    return newNode

def mul(node1, node2):
    value = node1.value * node2.value
```

In this code, we add the created node to the "child nodes" lists of the two input nodes: `node1` and `node2`. As we mentioned earlier, when `node1` and `node2` were created, their lists of child nodes were unknown and left empty. Thus, in creating each node, we append the node to the list of its parent(s).

The output of the function should be the created node. This setting enables the nested function call. That is, calling

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

finishes the function evaluation. At the same time, we build the computational graph.

4 Topological Order and Partial Derivatives - Forward Mode

Once the computational graph is built, we want to use the information in the graph to compute

$$\frac{\partial y}{\partial x_1} = \frac{\partial v_5}{\partial x_1}.$$

4.1 Finding the Topological Order

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

$$\dot{v}_5 = \frac{\partial v_5}{\partial v_4}\dot{v}_4 + \frac{\partial v_5}{\partial v_3}\dot{v}_3. \tag{4}$$

We are able to calculate

$$\frac{\partial v_5}{\partial v_4} \quad \text{and} \quad \frac{\partial v_5}{\partial v_3}, \tag{5}$$

because the node v_5 stores the needed information related to its parents v_4 and v_3. We defer the details on calculating (5), so the focus is now on calculating $\dot{v}_4$ and $\dot{v}_3$. For $\dot{v}_4$, we further have

$$\dot{v}_4 = \frac{\partial v_4}{\partial v_1}\dot{v}_1 + \frac{\partial v_4}{\partial v_2}\dot{v}_2, \tag{6}$$

which, by the same reason, indicates the need of $\dot{v}_1$ and $\dot{v}_2$. On the other hand, we have $\dot{v}_3 = 0$ since v_3 (i.e., $\sin(x_2)$) is not a function of x_1. The discussion on calculating $\dot{v}_4$ and $\dot{v}_3$ leads us to find that

$$v \text{ is not reachable from } x_1 \text{ in the graph} \Rightarrow \dot{v} = 0. \tag{7}$$

We say a node v is reachable from a node u if there exists a path from u to v in the graph. From (7), now we only care about nodes reachable from x_1. Further, we must properly order nodes reachable from x_1 so that, for example, in (4.2), $\dot{v}_4$ and $\dot{v}_3$ are ready before calculating $\dot{v}_5$. Similarly, $\dot{v}_1$ and $\dot{v}_2$ should be available when calculating $\dot{v}_4$.

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

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

and

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

Then,

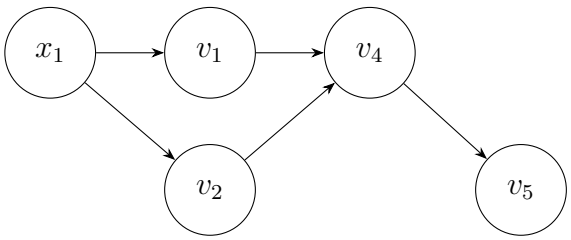
$$G_R \equiv \langle V_R, E_R\rangle$$

is a subgraph of G. For our example, G_R is the following subgraph with

$$V_R = \{x_1, v_1, v_2, v_4, v_5\}$$

and

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



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 (6). 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. Since G_R is a directed acyclic graph (DAG), a topological ordering must exist.[6] We may use depth first search (DFS) to traverse G_R to find the topological ordering. For the implementation, earlier we included a member "child nodes" in the `Node` class, but did not explain why. The reason is that to traverse G_R from x_1, we must access children of each node.

Based on the above idea, we can have the following code to find a topological ordering.

Listing 2: Using depth first search to find a topological ordering

```
def topological_order_forward(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))
```

The function `add_children` implements the depth-first-search of a DAG. From the input node, it sequentially calls itself by using each child as the argument. This way explores all nodes reachable from the input node. After that, we append the input node to the end of the output list, and during traversal, we append the current node to the output list after all its children nodes has been traversed. Also, we must maintain a set of visited nodes to ensure that each node is included in the ordering exactly once. For our example, the argument used in calling the above function is x_1, which is also the root node of G_R. The first path of the depth-first search is

$$x_1 \to v_1 \to v_4 \to v_5, \tag{8}$$

[6] We do not get into details, but a proof can be found in **?**.

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. One may wonder whether we can add a node to the list before adding its child nodes. This way, we have a simpler implementation without needing to reverse the list in the end. However, this setting may fail to generate a topological ordering. We obtain the following list for our example:

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

A violation occurs because v_2 does not appear before its child v_4. The key reason is that in a DFS path, a node may point to another node that was added earlier through a different path. Then this node becomes after one of its children. For our example,

$$x_1 \to v_2 \to v_4 \to v_5$$

is a path processed after the path in (8). Thus, v_2 is added after v_4 and a violation occurs. Reversing the list can resolve the problem. To this end, in the function `add_children`, we must append the input node in the end.

In automatic differentiation, methods based on the topological ordering are called *tape-based* methods. They are used in some real-world implementations such as Tensorflow. The ordering is regarded as a tape. We read the nodes one by one from the beginning of the sequence (tape) to calculate the derivative value.

Based on the obtained ordering, subsequently let us see how to compute each $\dot{v}$.

4.2 Computing the Partial Derivative

Earlier, by the chain rule, we have

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

In Section 4.1, we mentioned that

$$\dot{v}_4 \text{ and } \dot{v}_3$$

should be ready before calculating $\dot{v}_5$. For

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

we are able to calculate and store them when v_5 is created. The reason is that from

$$v_5(v_4, v_3) = v_4 - v_3,$$

we know

$$\frac{\partial v_5}{\partial v_4} = 1 \text{ and } \frac{\partial v_5}{\partial v_3} = -1.$$

A general form of our calculation is

$$\dot{v} = \sum_{u \in v\text{'s parents}} \frac{\partial v}{\partial u} \dot{u}. \tag{9}$$

The second term, $\dot{u} = \dfrac{\partial u}{\partial x_1}$, comes from past calculation due to the topological ordering. We can calculate the first term because u is one of v's parent(s) and we know the operation at v. 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 immediately computed and stored when we construct the computational graph. Therefore, we add one member "gradient w.r.t. parents" to our `Node` class. In addition, we need a member "partial derivative" to store the accumulated sum in the calculation of (9). In the end, this member stores the $\dot{v}_i$ value. Details of the derivative evaluation are in Listing 4. The complete list of members of our node class is in the following table.

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

We update the `mul` function accordingly.

Listing 3: The wrapping function "`mul`". The change from Listing 1 is in red color.

```
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
```

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:

- For `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.].
```

- For `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].
```

Now, we know how to get each term in (9), i.e., the chain rule for calculating $\dot{v}$. Therefore, if we follow the topological ordering, all $\dot{v}_i$ (i.e., partial derivatives with respect to x_1) can be calculated. An implementation to compute the partial derivatives is as follows. Here we store the resulting value in the member `partial_derivative` of each node.

Listing 4: Evaluating derivatives

```
def forward(rootNode):
    rootNode.partial_derivative = 1
    ordering = topological_order_forward(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
```

4.3 Summary

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 demonstrate the forward mode automatic differentiation.

5 Topological Order and Partial Derivatives - Reverse Mode

Recall in section 2.2, we mentioned the focus of the reverse mode automatic differentiation is on the derivatives of a single output variable w.r.t. all input variables. Therefore, after the computational graph is built, we want to use the information in the graph to compute both derivatives of the output variable y w.r.t. to each input variable:

$$\frac{\partial y}{\partial x_1} = \frac{\partial v_5}{\partial x_1}\,, \frac{\partial y}{\partial x_2} = \frac{\partial v_5}{\partial x_2}.$$

5.1 Finding the Topological Order

Recall that $\partial y/\partial v$ is denoted by $\bar{v}$. From the chain rule,

$$\bar{x}_2 = \bar{v}_2 \frac{\partial v_2}{\partial x_2} + \bar{v}_3 \frac{\partial v_3}{\partial x_2} \tag{10}$$

We are able to calculate

$$\frac{\partial v_2}{\partial x_2} \quad \text{and} \quad \frac{\partial v_3}{\partial x_2}, \tag{11}$$

because the node x_2 stores the needed information related to its children v_2 and v_3 in order to compute the derivative of its children w.r.t. itself. We defer the details on calculating, so the focus is now on calculating $\bar{v}_2$ and $\bar{v}_3$. For $\bar{v}_2$, we further have

$$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2} \tag{12}$$

where v_4 is the children of v_2 , and by the same reason as before, indicates the need of $\bar{v}_4$. At the same time, the intermediate result $\bar{v}_2$ is needed in the calculation of $\bar{x}_1$

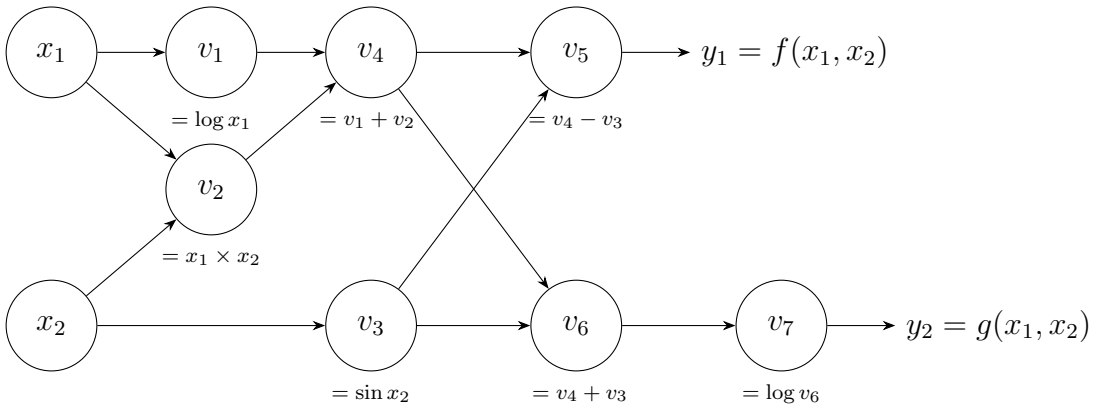
$$\bar{x}_1 = \bar{v}_1 \frac{\partial v_1}{\partial x_1} + \bar{v}_2 \frac{\partial v_2}{\partial x_1} \tag{13}$$

Similarly, this indicates the need of $\bar{v}_1$. We see that an ordering is needed to ensure that $\bar{v}_2$ is ready before $\bar{x}_1$ and $\bar{x}_2$, and $\bar{v}_1$ is ready before $\bar{x}_1$, or more generally, that the intermediate result $\bar{v}$ is ready before the parents of v require it.

On the other hand, the intermediate result $\bar{v}$ (e.g. 13) is defined by the derivatives of the output node w.r.t. v's children (i.e. $\bar{u}$ where u is a child of v) and the derivatives of the children w.r.t. to v. The nodes that are needed are the nodes reachable from the input nodes and those nodes need to be an ancestor of the output node v_5. This is because for a node to be considered first, it needs to be a node that uses the input variable, which will be reachable from the input node. Furthermore, if a node is not an ancestor of the target output node, then it entails that the output node does not define its function using that node, and therefore, any derivative of the output node, will not be related to the node.

However, the first condition is trivial, since all nodes in the computational graph should be reachable from the input node, as they are all defined using the input variables.

To better illustrate this we need an example with multiple outputs:



The graph above is extended from the previous example. Recall the goal is to compute the derivative of a single output variable w.r.t. all input variables, which in this case the target output variable is v_5. An example intermediate result $\bar{v}_4$ is defined as

$$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4} + \bar{v}_6 \frac{\partial v_6}{\partial v_4} \tag{14}$$

Since v_6 is not an ancestor of v_5, $\bar{v}_6 = \dfrac{\partial v_5}{\partial v_6} = 0$, and terms that contains v_6 can be removed from the equation.

Therefore, we must consider all nodes that are ancestors of the target output node, and then create a ordering of theses nodes where the children come before the parents.

In order to consider nodes that are ancestors of the target output node v_5, from the computational graph $G = \langle V, E \rangle$, where V and E are sets of nodes and edges, respectively. We define

$$V_R = \{v \in V \mid v \text{ is ancestor of } v_5\}$$

and

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

Then,

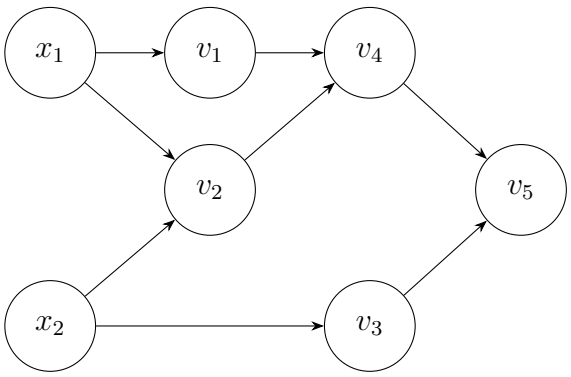
$$G_R \equiv \langle V_R, E_R \rangle$$

is a subgraph of G. For our example, G_R is the following subgraph with

$$V_R = \{x_1, x_2, v_1, v_2, v_3, v_4, v_5\}$$

and

$$E_R = \{(x_1, v_1), (x_1, v_2), (x_2, v_2), (x_2, v_3), (v_1, v_4), (v_2, v_4), (v_3, v_5), (v_4, v_5)\}.$$



We aim to find a "suitable" ordering of V_R satisfying that each node $u \in V_R$ comes before all of its parent nodes in the ordering. By doing so, $\bar{v}$ can be used in the derivative calculation of its child nodes; see (10). For our example, a "suitable" ordering can be

$$v_5, v_3, v_4, v_1, v_2, x_2, x_1.$$

In contrast to the topological ordering in the forward mode implementation 4.1, instead of an ordering where the parent nodes come before the children nodes, we desire the children nodes to come before the parent nodes. We may perform a depth first search (DFS) to traverse G_R to find the topological ordering.

Based on the above idea, we can have the following code to find a topological ordering.

Listing 5: Using depth first search to find a topological ordering

```
def topological_order_reverse(rootNode):
    def add_parent(node):
        if node not in visited:
            visited.add(node)
            for parent in node.parent_nodes:
                add_parent(parent)
            ordering.append(node)
    ordering, visited = [], set()
    add_parent(rootNode)
    return list(reversed(ordering))
```

The function `add_parent` implements the depth-first-search of a DAG. From the output node, it sequentially calls itself by using each parent as the argument. This way explores all nodes that are ancestors of the output node, and during traversal, we append the current node to the output list after all its children nodes has been traversed. After that, we append the output node to the end of the output list. Also, we must maintain a set of visited nodes to ensure that each node is included in the ordering exactly once. For our example, the argument used in calling the above function is v_5, which is a leaf node of G_R. The first path of the depth-first search is

$$v_5 \to v_4 \to v_1 \to x_1, \tag{15}$$

so x_1 is added first. In the end, we get the following list

$$[x_1, v_1, x_2, v_2, v_4, v_3, v_5].$$

Then, by reversing the list, a node always comes before its parent.

5.2 Computing the Partial Derivative

Earlier, by the chain rule, we have

$$\bar{x}_2 = \bar{v}_2 \frac{\partial v_2}{\partial x_2} + \bar{v}_3 \frac{\partial v_3}{\partial x_2} \tag{16}$$

In Section 5.1, we mentioned that

$$\bar{v}_2 \text{ and } \bar{v}_3$$

should be ready before calculating $\bar{x}_2$. For

$$\frac{\partial v_2}{\partial x_2} \text{ and } \frac{\partial v_3}{\partial x_2},$$

we are able to calculate them since we know the function definition of the children x_2. However, the values of these derivatives cannot be computed at the creation of x_2, since we do not know the children of x_2 at its time of creation. From

$$v_2(x_1, x_2) = x_1 \times x_2,$$

$$v_3(x_2) = \sin x_2,$$

we know

$$\frac{\partial v_3}{\partial x_2} = x_1 \text{ and } \frac{\partial v_2}{\partial x_2} = \cos x_2$$

A general form of our calculation is

$$\bar{v} = \sum_{u \in v\text{'s children}} \bar{u} \, \frac{\partial u}{\partial v} \tag{17}$$

The first term, $\bar{u} = \dfrac{\partial y}{\partial u}$, comes from past calculations based on the topological ordering. We can calculate the second term because u is one of v's children and we know the operations at u w.r.t. to v. For example, we have $v_4 = v_1 + v_2$, so

$$\frac{\partial v_4}{\partial v_2} = 1$$

In contrast to the forward mode implementation, these values cannot be immediately computed when we construct the graph. This reason is that we don't have the information regarding the children of a node at the time of its construction. Since we construct the computational graph in a forward pass fashion, we know about only the parents of a node when it's created. However, a

simple extension of the forward mode implementation can allow us to obtain the derivatives of the children of a node w.r.t. to the node at runtime.

Recall that we added 2 members "gradient w.r.t. parents" and "partial derivative" to our `Node` class. One is used to store the node's derivative w.r.t to its parents, the other is to store the intermediate result $\dot{u}$. The complete unchanged list of members of our node class is in the following table.

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

In the reverse mode implementation, we shall use the "partial derivative" member to store $\bar{v}$. At the same time, to obtain the derivatives of the children of a node w.r.t. to the node, we will, at runtime, obtain it from the children of the node by accessing the "gradient w.r.t. parents" member of its children.

We now have everything we need. Therefore, if we follow the topological ordering, all $\bar{v}$ (i.e., partial derivatives of y w.r.t. to v) can be calculated. An implementation to compute the partial derivatives is as follows. The action of obtaining the derivatives of the children of the current node w.r.t. to the node itself is highlighted in red.

Listing 6: Evaluating derivatives

```
def reverse(rootNode):
    rootNode.partial_derivative = 1
    ordering = topological_order_reverse(rootNode)
    for node in ordering[1:]:
        partial_derivative = 0
        for i in range(len(node.child_nodes)):
            parent_idx_in_child = node.child_nodes[i].parent_nodes.index(node)
            dchild_dnode = \
                node.child_nodes[i].grad_wrt_parents[parent_idx_in_child]
            dy_dchild = node.child_nodes[i].partial_derivative
            partial_derivative += dy_dchild * dchild_dnode
        node.partial_derivative = partial_derivative
```

5.3 Summary

The procedure for reverse mode includes three steps:

1. Create the computational graph

2. Find a topological order of the graph of the ancestors of y, where children comes before parents

3. Compute the partial derivative of y with respect to the current node v along the topological order. Once all input nodes $x_1, x_2, \ldots,$ are all traversed, you will have achieved the objective of obtaining the partial derivative of y with respect to the input nodes.

We discuss not only how to run each step but also what information cannot be obtained at the time of construction of the computational graph, and how we circumvent it by extending the forward mode implementation and obtaining these information at runtime. This is a minimal implementation to demonstrate the reverse mode automatic differentiation.

6 Study of Reverse Mode Automatic Differentiation in the Autograd-library

The AutoGrad-library offers a real-world, complete approach to automatic differentiation. This chapter treats how the chain rule is implemented to handle different data types in an efficient and memory-saving manner.

6.1 Implementation of Function Evaluation and the Computational Graph

The manual construction of the computation graph in AutoGrad is approached differently than in simpleautodiff. The simpleautodiff-library saves parents and children of each Node manually to maintain the topological order of the graph. Autograd uses the trace()-function to implicitly draw a computation graph:

Listing 7: Tracing operations of chained functions

```
def trace(start_node, fun, x):
    with trace_stack.new_trace() as t:
        start_box = new_box(x, t, start_node)
        end_box = fun(start_box)
        if isbox(end_box) and end_box._trace == start_box._trace:
            return end_box._value, end_box._node
        else:
            warnings.warn("Output␣seems␣independent␣of␣input.")
            return end_box, None
```

For each mathematical operation, a node (VJPNode()-class) is created that stores a reference to its value (but not the value itself, therefore memory-saving), the parent nodes and the Vector-Jacobian product. Due to having the information about the parents stored, the graph does not have to be built as a centralized object, but is implicitly built as a decentralized network of nodes and can

be reconstructed with its topological ordering. This dynamic construction does not require calling and re-adjusting a "Graph"-object for every operation. The result of the mathematical operation that each node represents is saved as a Box()-value. The Box()-class stores the resulting value and a reference to its node, that way the next operation will again use the values of the corresponding Box()-classes. After running through all operations, the final Box holds the output value and a reference to the terminal node. Why is the Box()-class crucial to the implementation? Each operation can either deal with constant or differentiable values. The Box()-class is the wrapper around each numerical value that stores the information about its origin (the parent nodes, and the trace to the graph), such that the next operator for the value can compute the derivative. More precisely, the trace to the graph is the identifier of the differentiation level. The wrapping allows the user to keep writing math in plain Python with variables which store the necessary differentiation information and links to the computational graph in the background. Before the function evaluation starts, the boxing of the inputs has already happened:

Listing 8: Boxing of the function, then evaluation

```
def trace(start_node, fun, x):
    with trace_stack.new_trace() as t:
        start_box = new_box(x, t, start_node)
        end_box = fun(start_box)
```

As the graph is under construction, the resulting variables are being boxed after each operation. This will in the end result in end_box() which holds the terminal node.

Box()-objects are identified within the primitive()-function:

Listing 9: Scanning for Box()-classes.

```
    def f_wrapped(*args, **kwargs):
        boxed_args, trace, node_constructor = find_top_boxed_args(args)
```

The function is evaluated simultaneously with the graph construction. As opposed to symbolic differentation, AD as implemented in AutoGrad (and also simpleautodiff) directly computes the numerical results of the operations and wraps them into boxes.

6.2 Topological Order and Partial Derivatives - Reverse Mode

6.2.1 Finding the Topological Order - Reverse Mode

Because of the fact that in the directed acyclic graph that is constructed, the gradients of the parents are dependent on the children's gradients, the topological order of the graph needs to be saved. It was explained in the section above that it is sufficient to save the parents' nodes of each child node to implicitly create the graph as well as establishing the topological order of the graph. The topological sorting of the nodes is done via the toposort()-function: Box()-objects are identified within the primitive()-function:

Listing 10: How topological ordering is done.

```
def toposort(end_node, parents=operator.attrgetter("parents")):
    child_counts = {}
    stack = [end_node]
    while stack:
        node = stack.pop()
        if node in child_counts:
            child_counts[node] += 1
        else:
            child_counts[node] = 1
            stack.extend(parents(node))

    childless_nodes = [end_node]
    while childless_nodes:
        node = childless_nodes.pop()
        yield node
        for parent in parents(node):
            if child_counts[parent] == 1:
                childless_nodes.append(parent)
            else:
                child_counts[parent] -= 1
```

The first while-loop counts the children of each node, starting from the end_node. Due to algorithmical reasons, the end_node gets exactly one child node assigned. During the second loop, the actual ordering happens. Taking advantage of the information about the number of children, the loop yields the node during the execution of the algorithm, searches for its parents, yields all other children of each parent, then moves on to the parent and repeats the same procedure. Using the yield method, the topological order is being generated as the function is running, which makes it time-efficient. Also, not the whole graph needs to be stored (as list()-object, for example) and therefore does not contribute to memory.

The backward_pass()-function is the one in which gradients are computed and backpropagated. The calculation of the partial derivative from which the gradients are computed, is assigned to ingrad.

Listing 11: The calculation of partial derivatives happens here.

```
        ingrads = node.vjp(outgrad[0])
```

6.2.2 Computing the Partial Derivative - Reverse Mode

6.3 Code examples

Acknowledgements

This work was supported by National Science and Technology Council of Taiwan grant 110-2221-E-002-115-MY3.

References