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 11, 2025

Abstract

Automatic differentiation is a key component in deep learning. This topic is well studied and excellent surveys such as Baydin et al. (2018) 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 Chinchalkar (1994), Bartholomew-Biggs et al. (2000), Baydin et al. (2018) and Margossian

---

[*]These authors contributed equally to this work

(2019) review the algorithms for automatic differentiation and its wide applications. In particular, Baydin et al. (2018) 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 (Paszke et al., 2017) and Tensorflow (Abadi et al., 2016), 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 Baydin et al. (2018) for the consistency with past works. Ours is thus an extension of Baydin et al. (2018) 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 Baydin et al. (2018). We consider the same example function

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

---

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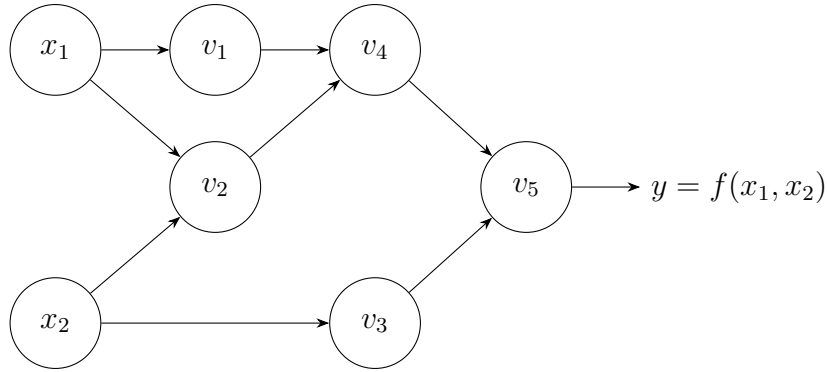[5] `https://github.com/karpathy/micrograd`
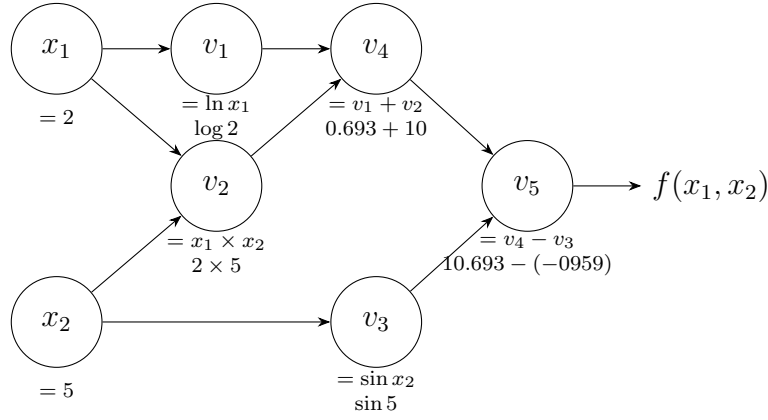
2.1   Forward Mode

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, in the following table, we have a forward procedure.

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

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.

x_1 → v_1 → v_4

$= 2$

$v_1 = \ln x_1$
$\log 2$

$v_4 = v_1 + v_2$
$0.693 + 10$

$v_2 = x_1 \times x_2$
2×5

$v_5 = v_4 - v_3$
$10.693 - (-0959)$

v_5 → $f(x_1, x_2)$

x_2 → v_3

$= 5$

$v_3 = \sin x_2$
$\sin 5$

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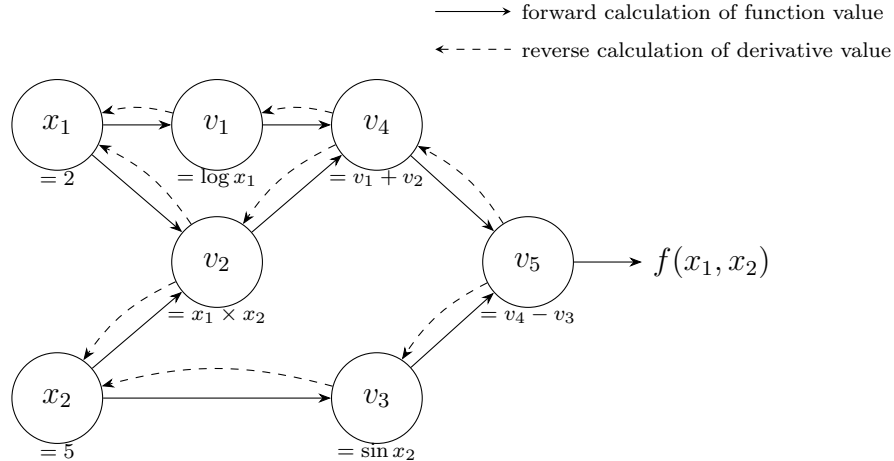
$$\bar{v} = \frac{\partial y}{\partial v}$$

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. For simplicity, we consider the forward mode. The reverse mode can be designed in a similar way. Consider a function $f : R^n \rightarrow R$ 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

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

due to the layered structure. For 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 in the following composite function

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

with

$$g(h_1, h_2) = h_1 - h_2,$$
$$h_1(x_1, x_2) = \log x_1 + x_1 x_2,$$
$$h_2(x_1, x_2) = \sin(x_2).$$

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

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

where the notation

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

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 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 we have a table recording the order to get $f(x_1, x_2)$:

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

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:

$$
v_1 = \log x_1,
$$
$$
v_2 = x_1 \times x_2,
$$
$$
v_3 = \sin x_2,
$$
$$
v_4 = v_1 \times v_2,
$$
$$
v_5 = v_4 - v_3.
$$

The expression in each node is from an operation to expressions in 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, at node v we should

8

store v's parents. Additionally, we need to record the operator applied on the node's parents and the resulting value. For example, the construction of the node
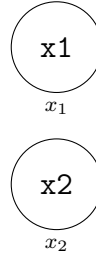
$$v_2 = x_1 \times x_2,$$

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.

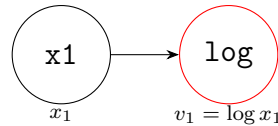| member	data type	example for `Node` v_2
numerical value	`float`	10
parent nodes	`List[Node]`	$[x_1, x_2]$
child nodes	`List[Node]`	$[v_4]$
operator	`string`	`"mul"` (for $\times$)

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

$$\texttt{y = sub(add(log(x1), mul(x1, x2)),sin(x2)).}$$

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`.
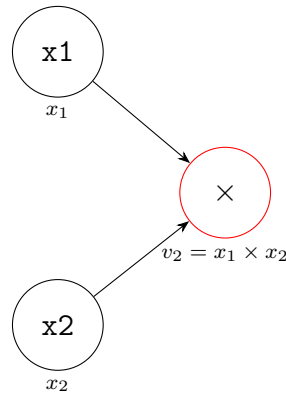

The implementation of our `log` function should create a `Node` instance to store $\log(x_1)$. Therefore, what we have is a wrapping function that does more than the log operation; see details in Section 3.3. The created node is the v_1 node in our computational graph. Next, we discuss details of the node creation. From the current log function and the input node x_1, we know contents of the following members:
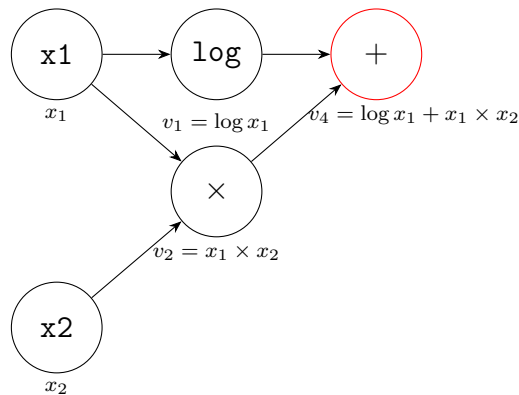
- parent nodes: $[x_1]$
- operator: `"log"`
- numerical value: $\log 2$

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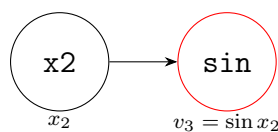
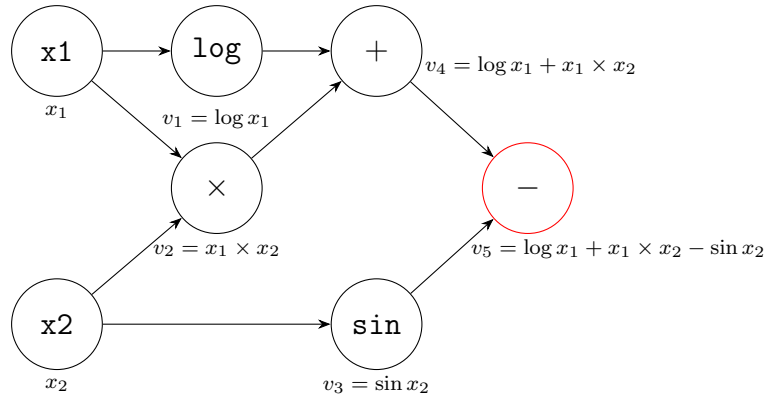We move on to apply `mul(node1, node2)` on nodes `x1` and `x2`.

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`"

```
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
```

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

$$y = \mathtt{sub(add(log(x1), mul(x1, x2)),sin(x2))}$$

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

4 Topological Order and Partial Derivatives

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 respectively sets of nodes and edges, we define

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

and

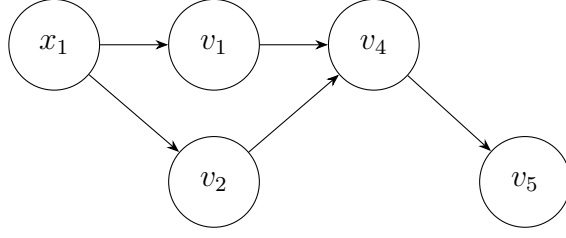$$E_R = \{(u, v) \in E \mid u \in V_R, \ v \in V_R\}.$$

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

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

12

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

```python
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))
```

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 input. This way explores all nodes reachable from the input node. After that, we append the input 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 input in calling the above function is x_1, which is also the root node of G_R. The left-most path of the depth-first search is

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

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

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

---

[6]We do not get into details, but a proof can be found in Kleinberg and Tardos (2005).

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,

$$x_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_5$$

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

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

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

$$\frac{\partial \text{ newNode}}{\partial \text{ Node1}} = \frac{\partial \text{ newNode}}{\partial \text{ Node2}} = 1,$$

15

so the red line is replaced by

$$\texttt{newNode.grad\_wrt\_parents = [1., 1.].}$$

- For `log(node)`, we have
$$\frac{\partial\ \texttt{newNode}}{\partial\ \texttt{Node}} = \frac{1}{\texttt{Node.value}},$$
so the red line becomes

$$\texttt{newNode.grad\_wrt\_parents = [1/node.value].}$$

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(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 Implementation of Reverse Mode

In Section 2.2, we introduced the basic concept of reverse mode automatic differentiation. Now we show how to implement it. Recall that in reverse mode, we denote

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

as the derivative of the output y with respect to the variable v. This is different from forward mode, where we computed $\dot{v} = \partial v / \partial x_1$. The key advantage of reverse mode is that we can compute $\partial y / \partial x_i$ for all input variables x_i in a single pass.

5.1 Reverse Topological Order

In forward mode, we needed a topological ordering that ensured each node came before its children. For reverse mode, we need the opposite: each node in the ordering should come before its parents. This makes sense because we start from the output $y = v_5$ and propagate gradients backward to the inputs.

Now, for reverse mode starting from the output node y, we define the set of nodes that can reach the output:

$$V_B = \{v \in V \mid y \text{ is reachable from } v\}$$

and correspondingly

$$E_B = \{(u, v) \in E \mid u \in V_B, \ v \in V_B\}.$$

Then $G_B \equiv \langle V_B, E_B \rangle$ is a subgraph of G containing all nodes that contribute to the output.
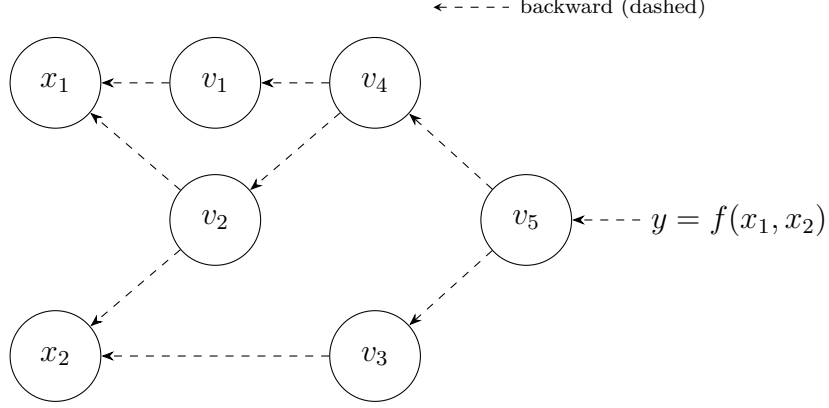
To perform reverse mode, we conceptually reverse all edges in G_B. We define the reversed edge set as

$$E_B^{\text{rev}} = \{(v, u) \mid (u, v) \in E_B\}.$$

The reversed graph is then

$$G_B^{\text{rev}} \equiv \langle V_B, E_B^{\text{rev}} \rangle.$$

In this reversed graph, edges point from children to parents. The dashed arrows show the direction of gradient flow in reverse mode:

Listing 5: Finding a reverse topological ordering

```python
def reverse_topological_order(output_node):
    def add_parents(node):
        if node not in visited:
            visited.add(node)
            for parent in node.parent_nodes:
                add_parents(parent)
            ordering.append(node)
    ordering, visited = [], set()
    add_parents(output_node)
    return list(reversed(ordering))
```

.

To find a reverse topological ordering of G_B, we can apply a topological ordering algorithm to G_B^{rev}. The key insight is that we can reuse the existing topological ordering algorithm from Section 4.1. Since our `Node` class stores both parent and child information, we can traverse the graph in either direction. For reverse mode, we traverse via child edges, which correspond to the forward edges in the reversed graph G_B^{rev}.

5.2 Computing Gradients in Reverse Mode

Once we have the reverse topological ordering, we can compute all $\bar{v}_i$ values. The computation follows the chain rule in reverse. For a node v with children nodes, we have

$$\bar{u} = \sum_{v \in u\text{'s children}} \bar{v} \cdot \frac{\partial v}{\partial u} \tag{10}$$

for each parent node u of v. Let us trace through our example. We start by setting

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

18

This is called "seeding" the gradient at the output. Next, we process v_5's parents: v_4 and v_3. From $v_5 = v_4 - v_3$, we know

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

These values were already stored in v_5's member `grad_wrt_parents` when we created the node. Using equation (10), we compute:

$$\bar{v}_4 = \bar{v}_5 \cdot \frac{\partial v_5}{\partial v_4} = 1 \cdot 1 = 1,$$

$$\bar{v}_3 = \bar{v}_5 \cdot \frac{\partial v_5}{\partial v_3} = 1 \cdot (-1) = -1.$$

We continue with v_4. From $v_4 = v_1 + v_2$, we have

$$\bar{v}_1 = \bar{v}_4 \cdot \frac{\partial v_4}{\partial v_1} = 1 \cdot 1 = 1,$$

$$\bar{v}_2 = \bar{v}_4 \cdot \frac{\partial v_4}{\partial v_2} = 1 \cdot 1 = 1.$$

For $v_3 = \sin(x_2)$, we get

$$\bar{x}_2 = \bar{v}_3 \cdot \frac{\partial v_3}{\partial x_2} = (-1) \cdot \cos(x_2) = -1 \cdot \cos(5) \approx -0.284.$$

Now we process $v_2 = x_1 \times x_2$. Note that x_2 has already received a gradient contribution from v_3. We must *add* the new contribution from v_2:

$$\bar{x}_1 = \bar{v}_2 \cdot \frac{\partial v_2}{\partial x_1} = 1 \cdot x_2 = 1 \cdot 5 = 5,$$

$$\bar{x}_2 \leftarrow \bar{x}_2 + \bar{v}_2 \cdot \frac{\partial v_2}{\partial x_2} = -0.284 + 1 \cdot x_1 = -0.284 + 2 = 1.716.$$

The use of "$\leftarrow$" emphasizes that we are accumulating gradients. This is crucial because a variable can affect the output through multiple paths in the graph.

Finally, for $v_1 = \log(x_1)$, we add another contribution to $\bar{x}_1$:

$$\bar{x}_1 \leftarrow \bar{x}_1 + \bar{v}_1 \cdot \frac{\partial v_1}{\partial x_1} = 5 + 1 \cdot \frac{1}{x_1} = 5 + \frac{1}{2} = 5.5.$$

At the end, we have

$$\frac{\partial y}{\partial x_1} = \bar{x}_1 = 5.5 \quad \text{and} \quad \frac{\partial y}{\partial x_2} = \bar{x}_2 = 1.716.$$

Both partial derivatives are computed in a single backward pass!

5.3   Implementation of Reverse Mode

The implementation follows the procedure described above. The important steps are:

1. Initialize all `partial_derivative` members to zero.

2. Seed the output node with gradient 1.

3. For each node in reverse topological order, propagate its gradient to its parents. The gradient contribution to each parent is the current node's gradient (`v_bar`) multiplied by the local derivative (`local_grad`). We use "`+=`" to accumulate contributions, which is essential when a parent has multiple children.

Listing 6: The backward function for reverse mode

```python
def backward(output_node):
    ordering = reverse_topological_order(output_node)
    for node in ordering:
        node.partial_derivative = 0.0
    output_node.partial_derivative = 1.0
    for node in ordering:
        v_bar = node.partial_derivative
        for i, parent in enumerate(node.parent_nodes):
            local_grad = node.grad_wrt_parents[i]
            contrib = v_bar * local_grad
            parent.partial_derivative += contrib
```

Notice that we reuse the same `Node` class and the same member `partial_derivative` that we used for forward mode. However, the meaning is different:

- In forward mode: `partial_derivative` stores $\dot{v} = \partial v/\partial x_1$.

- In reverse mode: `partial_derivative` stores $\bar{v} = \partial y/\partial v$.

This is possible because we run either forward mode or reverse mode, but not both at the same time.

5.4   Summary of Reverse Mode

The procedure for reverse mode includes three steps:

1. Create the computational graph (same as forward mode)

2. Find a reverse topological order starting from the output node

3. Compute partial derivatives with respect to all inputs by propagating gradients backward

The main advantage of reverse mode is efficiency when we have many inputs but few outputs. In deep learning, we typically have millions of parameters (inputs) but only one scalar loss function (output). Therefore, reverse mode, also known as backpropagation in the neural network context, is the standard choice for training deep learning models.

Acknowledgements

References

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

M. Bartholomew-Biggs, S. Brown, B. Christianson, and L. Dixon. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124(1-2):171–190, 2000.

A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

S. Chinchalkar. The application of automatic differentiation to problems in engineering analysis. *Computer Methods in Applied Mechanics and Engineering*, 118(1-2):197–207, 1994.

J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN 0321295358.

C. C. Margossian. A review of automatic differentiation and its efficient implementation. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 9(4):e1305, 2019.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in Pytorch. 2017.