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

December 14, 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

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

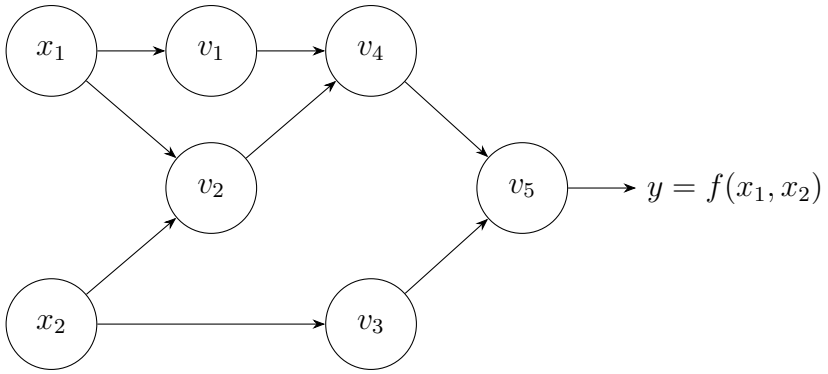
[1] `https://towardsdatascience.com/build-your-own-automatic-differentiation-program-6ecd585eec2a`
[2] `https://sidsite.com/posts/autodiff`
[3] `https://mdrk.io/introduction-to-automatic-differentiation-part2/`
[4] `https://github.com/dlsyscourse/lecture5/blob/main/5_automatic_differentiation_implementation.ipynb`
[5] `https://github.com/karpathy/micrograd`

2.1 Forward Mode

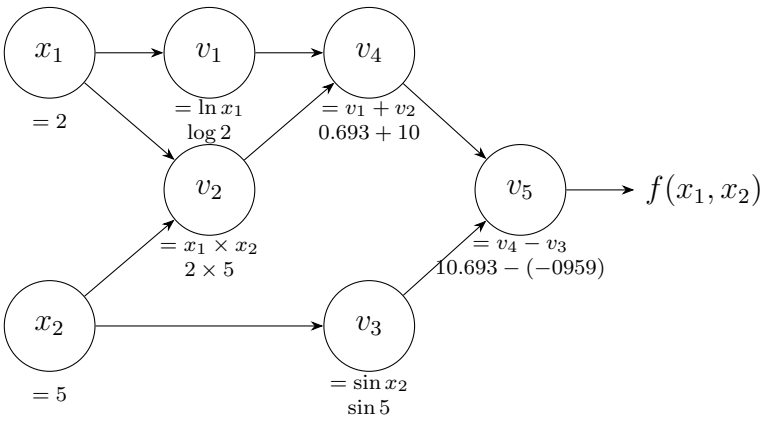
First, we discuss the forward mode. Before calculating the derivative, let us check how to calculate the function value. Assume that we want to calculate the function value at $(x_1, x_2) = (2, 5)$. Then, in the following table, we have a forward procedure.

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

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

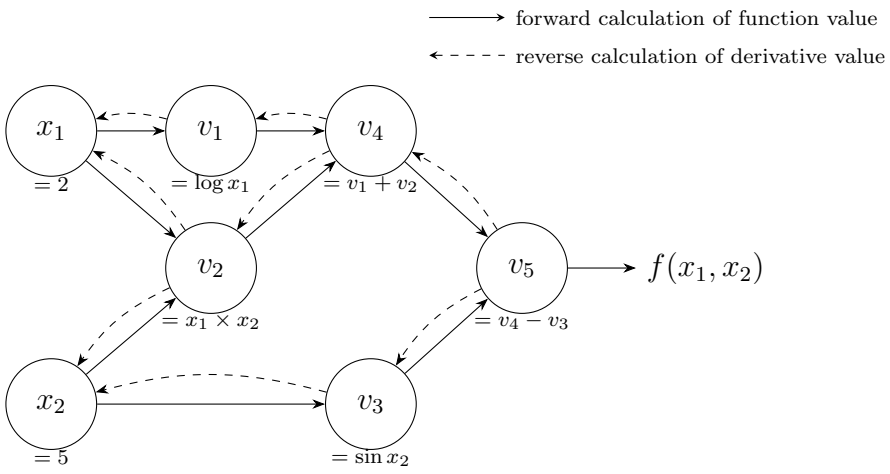
as the derivative of the function y with respect to the variable v. Note that earlier, in the forward mode, we considered

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

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

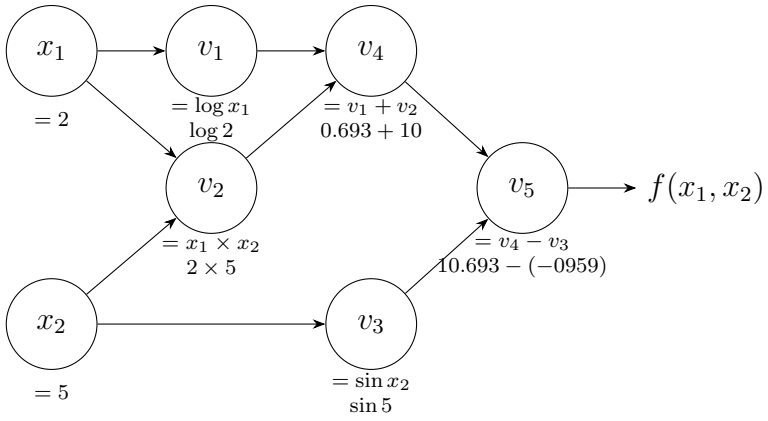
$$\frac{\partial g}{\partial h_i}\Big|_{\boldsymbol{h}=\boldsymbol{h}(\boldsymbol{x}_0)}$$

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

$$v_2 = x_1 \times x_2,$$

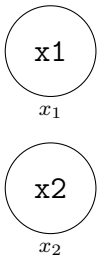
requires to store v_2's parent nodes $\{x_1, x_2\}$, the corresponding operator "$\times$" and the resulting value. Up to now, we can implement each node as a class `Node` with the following members.

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 $\times$)

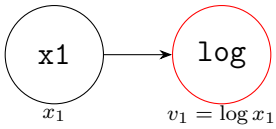
At this moment, it is unclear why we should store child nodes in our `Node` class. Later we will explain why such information is needed. Once the `Node` class is ready, starting from initial nodes (which represent x_i's), we use nested function calls to build the whole graph. In our case, the graph for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

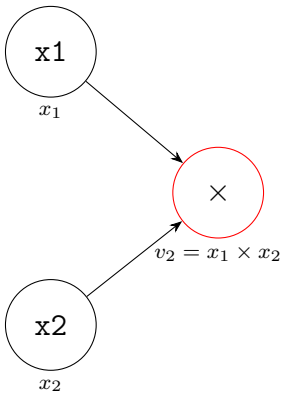


The implementation of our `log` function should create a `Node` instance to store $\log(x_1)$. Therefore, what we have is a wrapping function that does more than the log operation; see details in Section 3.3. The created node is the v_1 node in our computational graph. Next, we discuss details of the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

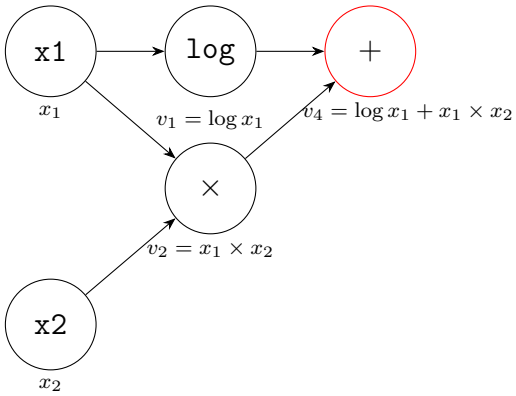
- parent nodes: $[x_1]$
- operator: `"log"`
- numerical value: $\log 2$

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.

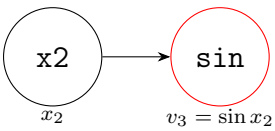
We move on to apply `mul(node1, node2)` on nodes `x1` and `x2`.



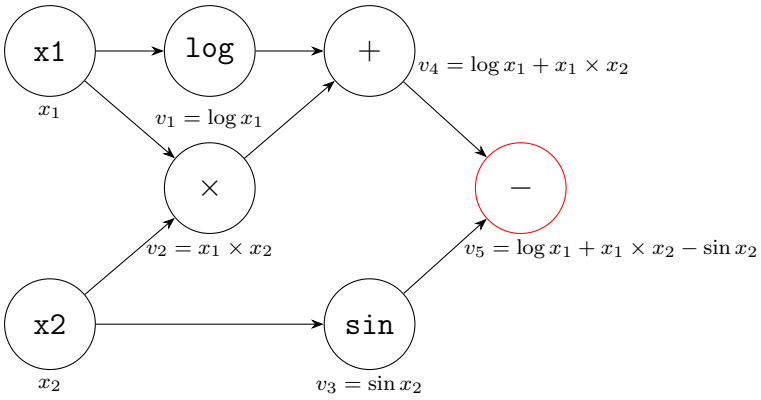
Similarly, the `mul` function generates a `Node` instance. However, different from `log(x1)`, the node created here stores two parents (instead of one). Then we apply the function call `add(log(x1), mul(x1, x2))`.



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,

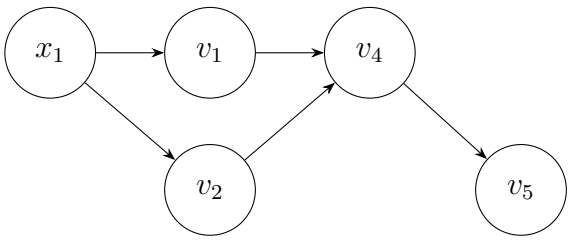
$$G_R \equiv \langle V_R, E_R\rangle$$

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 Kleinberg and Tardos (2005).

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

Why Reverse Mode? In Chapter 4, we implemented forward mode automatic differentiation. While intuitive, forward mode has a critical computational limitation when applied to machine learning.

Recall in section 2.1, forward mode computes the derivative of the output with respect to **one** input variable (e.g., $\partial y/\partial x_1$) in a single pass. If we have a function $f(x_1, \ldots, x_n)$ with n inputs and we want the gradient $\nabla f = [\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}]$, forward mode requires us to run the forward pass n times—once for each input variable. In modern deep learning, a neural network may have millions of parameters (x_i) but typically only a single output (the loss L). Using forward mode to train such a network would require millions of passes for a single gradient update, which is computationally infeasible.

In this chapter, we implement reverse mode by reversing the flow of information. We will need to:

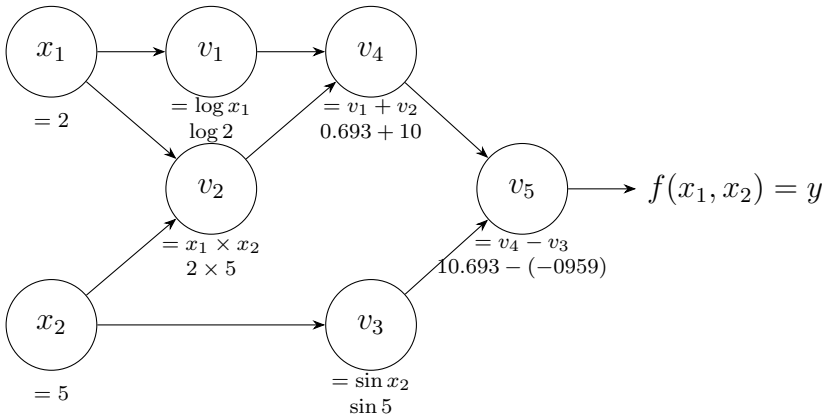
1. Construct the computational graph (Done in Chapter 3).
2. Find a topological order that visits children before parents (reverse topological order).
3. Compute partial derivatives by traversing this order.

Namely, 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. 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

We will start by going through the formulation of reverse mode automatic differentiation, and demonstrate that a topological order is needed. Then we will discuss what nodes are required in the ordering, and what nodes are irrelevant. We shall show again the computational graph example:



Recall that $\partial y/\partial v$ is denoted by $\bar{v}$. From the chain rule, one of our target derivatives $\bar{x}_2$ can be expanded to:

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

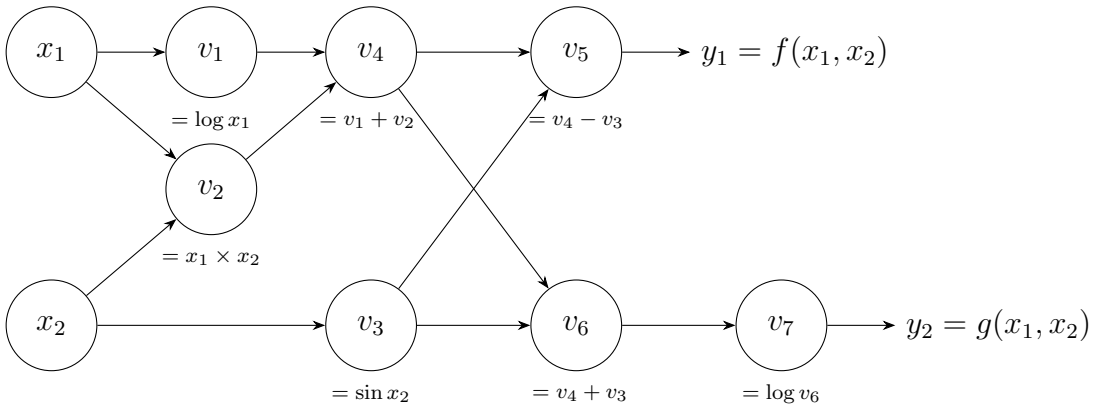
We are able to calculate the following:

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

This is because the node x_2 in the computational graph stores the information needed related to its children v_2 and v_3 to compute the derivative of its children w.r.t. itself. We defer the details on the calculation of 11. This implies a strict dependency: **the derivatives $\bar{v}_2, \bar{v}_3$ (from the child) must be available before we can compute $\bar{x}_2$.**

Ultimately, we see that an ordering is needed to ensure that $\bar{v}_4$ should be ready before $\bar{v}_2$, $\bar{v}_2$ should be ready before $\bar{x}_1$ and $\bar{x}_2$, and $\bar{v}_1$ is ready before $\bar{x}_1$. More generally, that the intermediate result $\bar{v}$ is ready before the parents of v require it. Namely, **we need an ordering that guarantees children come before the parents.**

We now know that an ordering is required, but should all nodes be considered in this ordering? The answer is NO, **only those that are the ancestors of our target output node should be considered**. To better illustrate this, we shall use 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

$$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4} + \bar{v}_6 \frac{\partial v_6}{\partial v_4} \tag{12}$$

Since we know from the graph, v_6 is not a ancestor of v_5, the output v_5 does not depend on v_6. Therefore, $\bar{v}_6 = \dfrac{\partial v_5}{\partial v_6} = 0$, and this is true for all nodes that are not ancestor of the target output node.

Therefore, we must consider all nodes that are ancestors of the target output node, and then create an ordering of these 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,

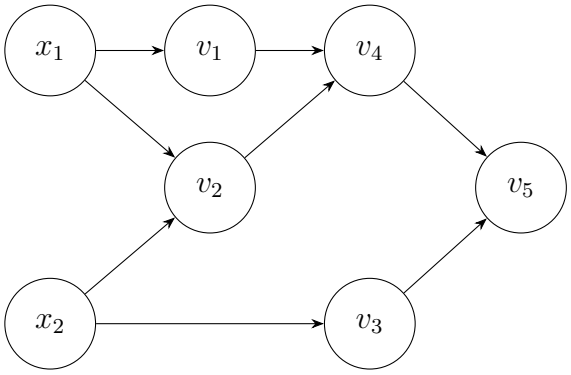
$$G_R \equiv \langle V_R, E_R \rangle$$

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 in Section 4.1, we desire an ordering where the children nodes to come before the parent nodes. **We can satisfy both the ordering requirement (Child before Parent) and the pruning requirement (Ancestors only) simultaneously using a Depth First Search (DFS) that traverses from the output backwards to the inputs**.

In `simpleautodiff`, we implement this in `topological_order_reverse`. Unlike the forward pass which traversed `child_nodes`, this function traverses `parent_nodes`:

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))
```

This algorithm works as follows:

1. **Start at the Output:** We call the function on the target node v_5.
2. **Traverse Parents:** The DFS recursively visits the parents of the current node. This ensures that we only visit nodes that contribute to v_5 (automatically pruning unrelated nodes like v_6, which are not ancestors of the target output node).
3. **Post-Order Gathering:** A node is appended to `ordering` only after all its ancestors (its parents, and their parents, etc.) have been visited. This ensures ancestors (which include parents) come before the children in the list.
4. **Final Reversal:** The list `ordering` ends up containing nodes ordered from [Inputs $\rightarrow$ Output] (parents to children). By applying `reversed()`, we obtain the desired order: [Output $\rightarrow$ Inputs] (children to parents).

We must maintain a set of visited nodes, `visited`, to ensure that each node is included in the ordering exactly once. This problem is caused by a node having multiple children. Using the example graph (5.1) above, we see that x_2 can be traversed from v_2 or v_3, and we only want one instance of traversal to x_2.

In our example, 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.

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

5.2 Computing the Partial Derivative

Now that we have the order or computation, we shall discuss the details of the computation. A general form of our calculation is

$$\bar{v} = \sum_{u \in v\text{'s children}} \bar{u} \, \frac{\partial u}{\partial v} \tag{13}$$

The first term on the right, $\bar{u} = \partial y / \partial u$, comes from previous calculations based on the topological ordering. The second term, $\partial u / \partial v$, represents the local derivative of the operation at node u with respect to its parent v, which we know we can compute given information about v's children. For example, using the same example earlier, by the chain rule, we have the following:

$$\bar{x}_2 = \bar{v}_2 \frac{\partial v_2}{\partial x_2} + \bar{v}_3 \frac{\partial v_3}{\partial x_2} \tag{14}$$

For the local derivatives,

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

We have:

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

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

Then we know:

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

Which we can then plug in the function values in the computational graph computed in Chapter 3 to obtain its value.

Our implementation does not change the `Node` class in the forward mode implementation. 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 $\bar{v}$, and the "gradient w.r.t. parents" will maintain its original usage.

There is an implementation detail that differentiates this from forward mode. In forward mode, the node being created, v, knew its own derivatives with respect to its parents. In reverse mode, we are processing the parent, v, but the derivative information $\partial v / \partial u$ is stored in the child. Therefore, to compute $\bar{v}$, we must:

1. Iterate over the children of v.
2. For each child, u_i, access its `grad_wrt_parents` attribute.
3. Find the index corresponding to u_i to retrieve the specific value $\partial u_i / \partial v$.

This "Pull" mechanism, where a parent reaches out to its children to pull gradient information, is highlighted in red in the implementation in `simpleautodiff` as follows:

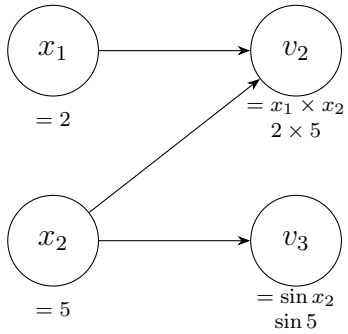
Listing 6: Evaluating derivatives

```
1  def reverse(rootNode):
2      rootNode.partial_derivative = 1
3      ordering = topological_order_reverse(rootNode)
4      for node in ordering[1:]:
5          partial_derivative = 0
6          for i in range(len(node.child_nodes)):
7              parent_idx_in_child = node.child_nodes[i].parent_nodes.index(node)
8              dchild_dnode = \
9                  node.child_nodes[i].grad_wrt_parents[parent_idx_in_child]
10             dy_dchild = node.child_nodes[i].partial_derivative
11             partial_derivative += dy_dchild * dchild_dnode
12         node.partial_derivative = partial_derivative
```

On line 3 and 4, we first generate the topological ordering, then we go traverse the computational graph using this ordering.

From line 6 to 12, is the meat of the calculation. We obtain the local derivative (e.g. $\partial u/\partial v$) from a child, multiply this by its corresponding intermediate result (e.g. $\bar{u}$). We do this for all children, and sum up each product from the children.

Let us trace the calculation for $\bar{x}_2$ using the example graph. We shall show a portion of the example graph here:



The topological order processes v_2 and v_3 before x_2. When the loop on line 4 reaches x_2, it subsequently compute from line 6 to 12:

1. It accesses child v_2. It retrieves $\bar{v}_2$ (already computed) and looks up $\partial v_2/\partial x_2 = x_1$.
   Term 1: $\bar{v}_2 \cdot x_1$.

2. It accesses child v_3. It retrieves $\bar{v}_3$ (already computed) and looks up $\partial v_3/\partial x_2 = \cos(x_2)$.
   Term 2: $\bar{v}_3 \cdot \cos(x_2)$.

3. It sums them: $\bar{x}_2 = \bar{v}_2 x_1 + \bar{v}_3 \cos(x_2)$, matching Equation (10).

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 each step is performed 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 reverse mode automatic differentiation.

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

The `AutoGrad` library offers an approach applicable to real-world automatic differentiation tasks. This chapter treats the implementation of AD for Machine Learning problems in an efficient and memory-saving manner. Section 6.1 will be concerned with the implementation of `AutoGrad` and a walkthrough, while Section 6.2 will reason for the applicability of `AutoGrad` in real-world scenarios.

6.1 Implementation features of AutoGrad

This is a breakdown of the functionality of `AutoGrad`. Along the walkthrough, we will go into details regarding some core functions, the `Node` class, the `Box` class and the decorator `@primitives`.

Conceptually, the implementation of `AutoGrad` is close to Summary points 1. and 2. of Section 5.3, while point 3. exhibits some heavier differences. The details are explained in the following chapters.

6.1.1 Implementation of Function Evaluation and the Computational Graph

A call of the main function for the user, `grad`, initializes building the computational graph, for which `Autograd` uses the `trace` function.

Listing 7: Tracing operations of chained functions

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

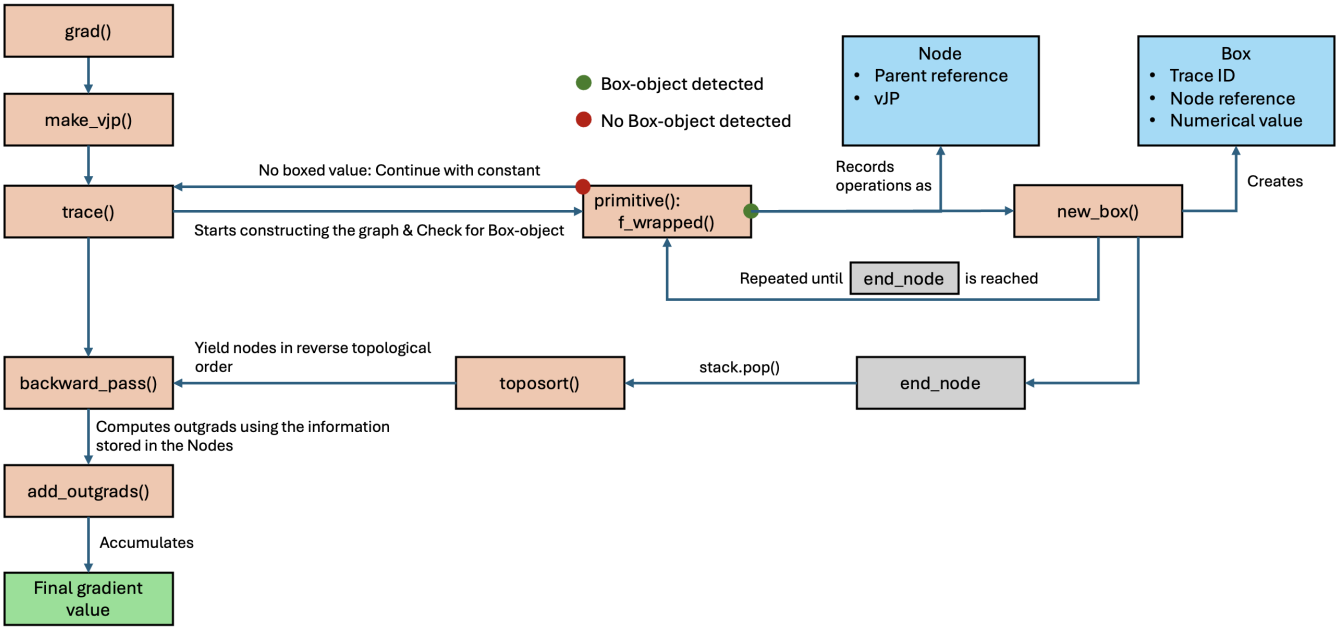


Figure 1: AutoGrad functionality.

```
        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
```

It might not be obvious from the code, maybe Figure 1 from above can support noticing that the function is not evaluated all at once in contrast to `simpleautodiff`'s approach. Instead, as soon as `grad()` is being called, the intermediate results from the applied mathematical operations are stored and re-used by the next operation. This repeats until the last operation has been called. At this point, the building of the computational graph is finished. Let's now look at the details of the algorithm, yet, for now we assume that AutoGrad simply knows when it is handling a relevant differentiable value.

The implementation starts by identifying the variables that are part of the same `_trace` context and wrapping those into `Box` classes. For each operation applied to the boxed inputs, `AutoGrad` creates a `VJPNode` instance where references to the parent nodes as well as the corresponding vector-Jacobian-Product-rules are stored. The vJP-rule is inferred directly from the mathematical operation (we will see later, how). Since at this point we are still missing the incoming gradient of the child which is a dependency for the local gradient, only the vJP-rule is stored, not a numerical value. Keep in mind that indeed the numerical intermediate values of the forward pass are stored in a `Box` instance: the numerical values of the parents are in the `VJPNode`, while the value resulting

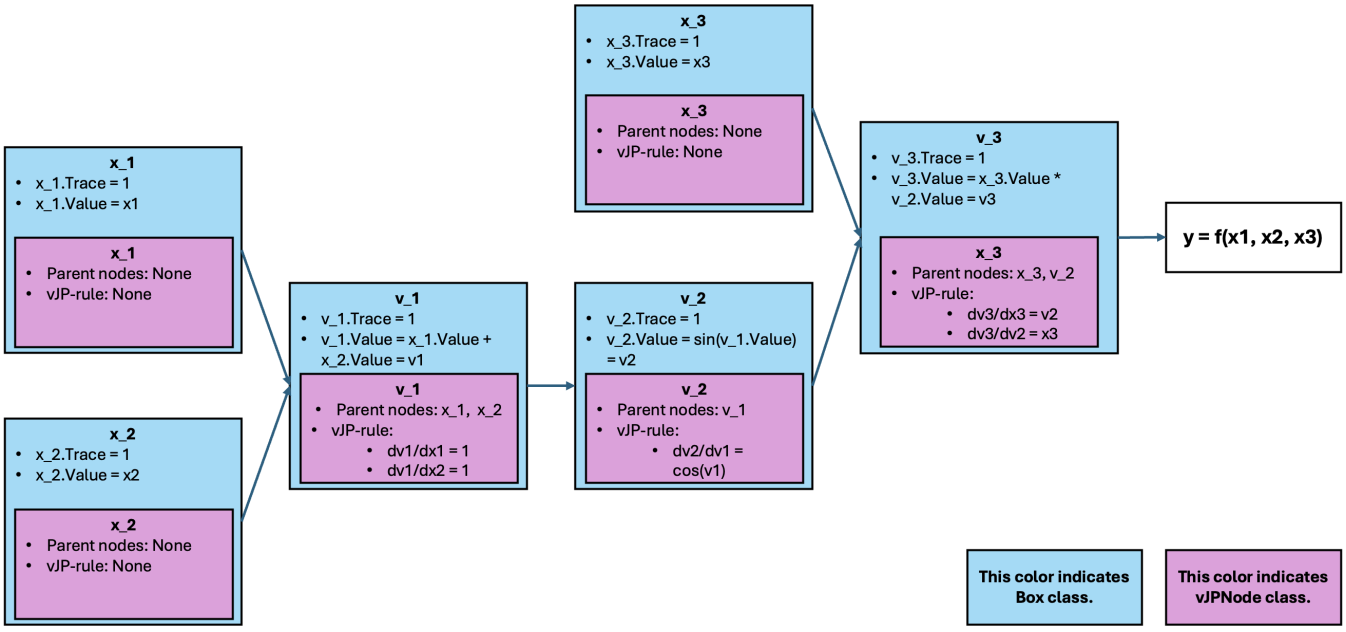


Figure 2: Implicit graph after calling `trace(fun)`.

from the operation is stored in the `Box`. Note that also a `VJPNode` reference is stored in the `Box` of which it produces the numerical value.

After all operations have been traced (i.e. the corresponding Nodes have been created), the `end_box` as indicated in Line 4 of the code-snippet above contains the leaf node of the graph. Having the complete set of Nodes obtained, we have the necessary information to draw a correct computational graph. In Figure 2 you see an illustrative visualization of the computational graph for the function $y = f(x_1, x_2, x_3) = sin(x_1 + x_2) \cdot x_3$.

An attribute mentioned above that becomes important for real-world scenarios is `_trace`. The trace reference is the identifier of the differentiation level. Calling `grad(grad(fun))` creates a computational graph for `fun` as well as a graph for `grad(fun)`. A more elaborate explanation and examples regarding applicability of higher-order-differentiation in certain scientific areas is provided in Section 6.2.2.

As mentioned, only elements that can contribute to the gradient, i.e. elements that are part of the same traced computation and depend on differentiable inputs, are wrapped in `Box` instances. In Section 6.1.3 we will explain how the `Box` instances get created.

The wrapping of values and functions as `@primitives` (Section 6.1.3) allows the user to keep writing math in plain Python. Before the function evaluation starts, the boxing of the inputs has already happened, see Line 3 in the code above.

To summarize `trace()` on a higher level, one could say it calls the wrapped version of the target function `fun`, and as the operations inside `fun` are being evaluated, `AutoGrad` records when each

operation occurs, what it does, and how their composition leads to the output value y.

6.1.2 Finding the Topological Order - Reverse Mode

After constructing the graph, the topological order must be established to enable backpropagation of gradients in the right order. It is crucial to have the gradient of a child ready before the computation of the parent's gradient, as was explained in Section 5.2. It was mentioned in Section 6.1.1 that it is sufficient to save the parents' nodes of each child node to implicitly create the graph.

The topological sorting of the nodes is done via the `toposort` function.

Listing 8: Topological ordering of the set of all nodes.

```
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 loop counts the children of each node, starting from the `end_node`. Parents get added to the `child_counts` dictionary until a root node is reached, then the next parent of the `end_node` (or the previous parent's upper parent) is added to the dictionary, and so on. Nodes that occur multiple times in the first loop get the corresponding number of children assigned. For simplicity of the algorithm, 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 `end_node` while the algorithm is running, searches for its parents, yields the other children of each parent, then moves on to the parent node 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 set of nodes needs to be stored and therefore does not contribute to memory space. Notably is as well that the second loop

uses a breadth-first-search-like (BFS) approach, whereas `simpleautodiff` uses DFS for the same task (see Section 4.1).

The `backward_pass` function is the function which calls `toposort` and receives the nodes. This is shown in Section 6.1.4.

6.1.3 Wrappers

For the function evaluation, it was mentioned in Section 6.1.1 that mathematical operations would be wrapped with the `@primitives` decorator to allow AutoGrad to track the operation, and differentiate between differentiable and non-differentiable functions. AutoGrad defines the wrappers and takes advantage of pre-defined rules for differentiable functions in the helper-file `numpy_vjps.py` and differentiates between strictly-non-differentiable `numpy` functions (e.g. `shape`, `floor`), unary operations (e.g. `log`, `arcsin`), binary operations (e.g. `multiply`, `power`) and more complicated operations (e.g. `kron`, `inner`).

Notably, differentiation rules for custom functions can also be created in `AutoGrad`.

Now, the above mentioned functions are the ones that we want to make use of for e.g. optimizing a loss-function of any arbitrary kind. While a human immediately detects the functions we are dealing with and can compute output by inserting inputs, Python does not natively know that those expressions are certain mathematical operations/transformations which can be differentiated. So, a requirement for differentiation in Python is that mathematical operations can be recognized as such in advance to ensure their correct categorization. Symbolic differentiation would find a formula and produce a new one for the derivative corresponding to the respective derivative rule. In AutoGrad, the `@primitives` decorator wraps the operations in the original function to make it traceable and to enable retrieving the vJP-rule from `numpy_vjps.py`. As an illustration, for a simple multiplication $x \cdot y$, we know that we are dealing with multiplication. Python only knows how to deal with the execution of multiplication natively, but the `@primitives` decorator allows inferring the vJP-rule:

Listing 9: vJP-rule for multiplication

```
defvjp(
    anp.multiply,
    lambda ans, x, y: unbroadcast_f(x, lambda g: y * g),
    lambda ans, x, y: unbroadcast_f(y, lambda g: x * g),
)
```

Inspecting the `f_wrapped` function:

Listing 10: Wrapping of mathematical operations

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

```
        if boxed_args:
            argvals = subvals(args, [(argnum, box._value) for argnum, box in boxed_args])
            if f_wrapped in notrace_primitives[node_constructor]:
                return f_wrapped(*argvals, **kwargs)
            parents = tuple(box._node for _, box in boxed_args)
            argnums = tuple(argnum for argnum, _ in boxed_args)
            ans = f_wrapped(*argvals, **kwargs)
            node = node_constructor(ans, f_wrapped, argvals, kwargs, argnums, parents)
            return new_box(ans, trace, node)
        else:
            return f_raw(*args, **kwargs)

    f_wrapped.fun = f_raw
    f_wrapped._is_autograd_primitive = True
    return f_wrapped
```

In Line 18-19, we can see that the function returns the wrapped function as well as the information that Python will deal with a wrapper in the proceeding. In the lines above, it detects whether the inputs of the functions are boxed or not. In the latter case, no differentiation is needed, and the output of the unwrapped function is simply returned in line 16. Otherwise, in Line 13, a new node is created which holds information about the operation that was applied, the parents of the node, the parents' numerical values and the raw numerical value.

6.1.4 Computing the VJP - Reverse Mode

The final step of automatic differentiation is in the `backward_pass` function.

Listing 11: Adding gradients of the children of each node in Line 7.

```
def backward_pass(g, end_node):
    outgrads = {end_node: (g, False)}
    for node in toposort(end_node):
        outgrad = outgrads.pop(node)
        ingrads = node.vjp(outgrad[0])
        for parent, ingrad in zip(node.parents, ingrads):
            outgrads[parent] = add_outgrads(outgrads.get(parent), ingrad)
    return outgrad[0]
```

We have noted earlier that the `yield` method used in the `toposort` function ensures efficiency of the implementation. Here, we can see why: The yielded nodes from `toposort` are being called to accumulate their gradients. Calling them in topological order ensures that the dependencies of local gradients are fulfilled. Let's look at the mathematical expression for the computation of a gradient $\bar{v}_1$, consisting of the incoming gradient $\bar{v}_2$ and the local gradient $\frac{\partial v_2}{\partial v_1}$:

$$\bar{v}_1 = \frac{\partial y}{\partial v_1} = \bar{v}_2 \cdot \frac{\partial v_2}{\partial v_1} = \frac{\partial y}{\partial v_2} \cdot \frac{\partial v_2}{\partial v_1}. \tag{15}$$

The incoming gradient is constituted in the code by `outgrad`. In the very first run of the loop, `outgrad` catches the seed gradient g, which typically equals 1. Note that this is the numerical value that the calculation of the local gradient of each parent is dependent on. The variable `ingrads` constitutes a list of gradients (singular for parents with only one child) where the incoming gradients and the local vJP-rules with the *incoming gradient as argument* for the vJP-rules of all children have already been multiplied, which was done in Line 5 by calling `node.vjp(outgrad[0])`. In math notation, `node.vjp(outgrad[0])` would look like

$$(\bar{v}_1, \bar{v}_2, \dots) = \left(\bar{v} \cdot \frac{\partial v}{\partial v_1}, \bar{v} \cdot \frac{\partial v}{\partial v_2}, \dots \right). \tag{16}$$

The loop in Line 6 and 7 accumulates the gradients $\bar{v}_j$ (if one child has several parents, otherwise the loop just consists of one iteration). Also, the `add_outgrads` adds up the gradients of the different children if one node has several ones. This will eventually result in the final gradient value, the output of `grad(fun)`.

6.2 From Toy to Machine: The Architecture of Applicability

While the pedagogical implementation in `simpleautodiff` is a great way to demonstrate the inner-workings of automatic differentiation, it falls short to more complex workflows. To transition from a "toy" implementation to a robust "machine" capable of empowering research, the architecture must handle the full expressivity of the host language, allow for more complex features like higher-order derivatives, and interact seamlessly with existing numerical libraries.

In this section, we explore the architectural decisions that enable Autograd to achieve this broad applicability. We examine how its "Define-by-Run" paradigm naturally accommodates dynamic control flow without the need for specialized syntax; how its recursive graph construction enables higher-order differentiation; how its modular wrapper design permits the effortless integration of custom operators; and how Autograd handles data types, such as scalars, complex numbers, and arbitrary container structures (like tuples of arrays).

6.2.1 Handling Dynamic Control Flow and Discussion on its Overhead

One of the biggest advantages of Autograd's design is its ability to handle dynamic control flow naturally. This capability stems directly from the "Define-by-Run" architecture discussed in Section 6.1.1. In this paradigm, we no longer need to specify the structure of the computational graph, instead the computational graph is constructed dynamically by tracing the execution of a Python function operation-by-operation, regenerated every time the function is executed, allowing it to adapt to the specific data flow of that iteration.

Our pedagogical implementation, `simpleautodiff`, surprisingly (for how simple it is), contains the fundamental mechanics of this paradigm, it wraps operations and creates the graph as the

gradient is being computed. As operations like `add(a, b)` or `mul(x, y)` are executed, `Node` objects are instantiated and linked in real-time, effectively generating the graph as the program runs. This allows `simpleautodiff` to support loops and conditionals that change the graph structure (e.g., iterating different numbers of times). However, it remains an *explicit* dynamic graph. The user must manually call specific naively-wrapped (not using decorators, users sees the wrap directly) versions of the operations, divorcing the mathematical definition from standard Python syntax (e.g. "+", "*"). Autograd evolves this into an *implicit* dynamic graph by wrapping native operations (via `numpy`), allowing the graph to be built transparently behind standard code. In addition, Autograd returns a function that evaluates the derivative, and does not require the inputs to this function before the evaluation contrary to `simpleautodiff`, this allows users to recompute the derivative through a simple interface.

Listing 12: Example of Dynamic Control Flow in simpleautodiff

```
def power_loop(x_val, n):
    # 1. Initialize the Input Node
    x = Node(x_val)

    # 2. Initialize the accumulator (start with x)
    result = x

    # 3. Dynamic Control Flow:
    # In each iteration, a new 'mul' Node is added to the graph.
    for i in range(n - 1):
        result = mul(result, x)
    return x, result
x_node, output_node = power_loop(2.0, 3)
reverse(output_node)
```

This "Define-by-Run" approach contrasts sharply with static graph construction frameworks such as Theano (The Theano Development Team et al. (2016)), where the user must define the entire graph structure before execution. In such systems, handling data-dependent control flow requires specialized operators (e.g., `scan`) or learning a separate mini-language, which steepens the learning curve.

In Autograd, standard Python control flow constructs—such as `for` loops, `while` loops, `if/else` branches, and recursion—are supported natively. Crucially, Autograd does not need to parse or understand these control flow structures. Instead, it simply tracks the sequence of primitive operations that are actually executed. For example, if a `while` loop runs ten times, Autograd sees a sequence of operations corresponding to ten unrolled iterations. If it runs twice on the next input, Autograd sees a shorter sequence. The tracer is agnostic to the control flow logic that generated the sequence.

On the discussion of the overhead, pre-defining the graph does allow for global optimizations

and lower runtime overhead by avoiding Python's interpreter during execution. Autograd accepts this overhead to gain flexibility. Moreover, This library was designed for specific "BLAS-Limited" workloads (Maclaurin (2016)), i.e. programs dominated by large-scale linear algebra operations, which are handled (via numpy) by ancient optimized FORTRAN libraries. As for GPU-bound workloads, specifically deep learning, which are dominated by numerical kernels, are handled in GPUs, Autograd itself does not support GPU, users can extend the library to do so (6.2.3). We can see that for common workloads like these, the programs are not bottlenecked by the overheads incurred by Autograd.

Notwithstanding, an absence of direct support for the GPU is a nuisance for modern use cases, where parallelization and "scaling laws" are often the slogans (e.g. LLMs). This has inspired libraries like JAX (Bradbury et al. (2018))(same creators from Autograd), and Pytorch Autograd (Paszke et al. (2017)). Both of the libraries are directly inspired by Autograd, and utilize the same "Define-by-run" architecture, this is a testament to this excellent design choice, and further proves the overheads are insubstantial in common use cases.

Example: Recurrent Neural Network (RNN)

We consider the loss function for a Recurrent Neural Network (RNN). RNNs require dynamic loss computation because they process sequences of variable lengths, causing the structure of the computation graph to change at runtime. In this example, the function `rnn_log_likelihood` is the objective. This approach removes the need for specialized control flow operators (e.g. `scan`) found in static graph frameworks.

Listing 13: RNN Log-Likelihood Loss Function

```
def rnn_log_likelihood(params, inputs, targets):
    # Computes the log-likelihood of the target sequence.
    # Autograd traces the 'rnn_predict' call (which contains the
    # recurrent loop) and the summation over time steps.
    logprobs = rnn_predict(params, inputs)
    loglik = 0.0
    num_time_steps, num_examples, _ = inputs.shape

    # Standard loop to aggregate loss over time
    for t in range(num_time_steps):
        loglik += np.sum(logprobs[t] * targets[t])

    return loglik / (num_time_steps * num_examples)
```

6.2.2 Higher-Order Differentiation

A distinctive feature of Autograd is its elegant way of handling higher-order derivatives. To use such a feature, one simply call `grad(grad(f))`. To understand it, we must first recognize that the

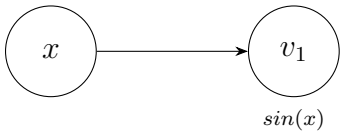
backward pass (6.1.4) is simply a sequence of operation invocations that generates the derivative of the target function $f(x)$, namely it executes a function that outputs the derivative, i.e. $\partial f/\partial x$. If we can trace such a function, we can generate a corresponding new computational graph, on which we can perform another backward pass, the output of this backward pass outputs the derivative of the derivative $\partial^2 f(x)/\partial x^2$. Given this idea, the function that defines the derivative of a operation/node, i.e. the Vector-Jacobian Product (VJP), must be itself composed of traceable operations (primitives).

For example, consider the operation $\dfrac{\partial^2 \sin(x)}{\partial x^2}$:

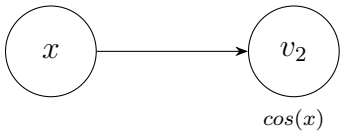
```
defvjp(anp.sin, lambda ans, x: lambda g: g * anp.cos(x))
```

The derivative of $sin(x)$ is defined as $g \cdot \cos(x)$, where g is the incoming gradient. Crucially, the operations "multiply" and "cos" used in this definition are themselves Autograd primitives. We shall illustrate using the 2 computational graphs below. The graph at the top is the computational graph created from the forward pass (i.e. trace) of $sin(x)$, the one below is the graph created from the forward pass of $cos(x)$ (backward pass of $sin(x)$), the derivative of $sin(x)$. As we compute $\partial \sin(x)/\partial x$ during the backward pass on the top graph, we will call the primitive function $cos(x)$, which defines the derivative of the node $sin(x)$, and as $cos(x)$ is called, the bottom graph is simultaneously created. Subsequently, a second backward pass is performed on the bottom graph, generating the second derivative $\partial^2 \sin(x)/\partial x^2$.

Computational graph of $\sin(x)$



Computational graph of $\cos(x)$



To manage these overlapping computations without confusion, Autograd uses a mechanism called the **Trace Stack**. Every variable being tracked is wrapped in a `Box` object containing a `_trace` attribute, which identifies exactly which `grad` call owns it.

When we call `grad(grad(f))`, two traces are active: an `Outer` trace and an `Inner` trace. The input x is wrapped twice: effectively `Box_Inner(Box_Outer(x))`.

1. **Forward Pass:** The `Inner` trace sees `Box_Inner(Box_Outer(x))`, and building the `Inner` graph (the computational graph of $\sin(\cdot)$).

2. **Inner Backward Pass:** The system executes the VJP: $g \cdot \cos(x)$. Crucially, the x used here is the `Box_Outer(x)` (since the inner box was peeled off during the forward pass).

3. **The Trace Mechanism:** When the operation $\cos(\cdot)$ encounters `Box_Outer(x)`, it checks the `_trace` attribute. Seeing that the box belongs to the `Outer` trace (which is still active), Autograd records this operation onto the `Outer` graph (the bottom graph).

Without the `_trace` attribute, the system encounters $\cos($`Box_Outer(x)`$)$ and faces a dilemma. If it ignores the box (treating it as a constant), no graph is recorded for the outer gradient, making the second derivative zero. If it records the operation onto the *current* (inner) trace, it corrupts the graph that is currently being backpropagated.

The `_trace` attribute explicitly identifies that x belongs to the **Outer Trace**. This signals the system to ignore x with respect to the currently executing inner backprop, but to **record** the operation onto the outer graph.

One can imagine, to compute higher order derivatives, it requires that every time we define a new gradient/Vector-Jacobian Product (VJP), we also define the gradients of the primitive functions we use to implement that gradient, and the gradients of those functions, so on and so forth. Fortunately, the recursion usually bottoms out fairly quickly (Maclaurin (2016)) (e.g. the sin(x), cos(x) family).

In contrast, our pedagogical implementation `simpleautodiff` is limited to first-order derivatives. During the forward pass In `simpleautodiff`, as we generate the computational graph, we compute and stores the numerical value of the local gradient (derivative of the operation w.r.t. its input) in the node's attribute `grad_wrt_parents`, instead of creating a vjp method on the node. Therefore, when we compute the derivative in the backward pass (the `reverse` function), we do not have the information about the operation that can be used to compute the local gradient. As a result, the backward pass cannot be traced. We show that in the wrapped operations in simpleautodiff, numerical values are stored in `grad_wrt_parents`:

Listing 14: Wrapped "mul" Operation

```
def mul(node1, node2):
    ...
    newNode.grad_wrt_parents = [node2.value,node1.value]
    ...
```

This ability to compute higher-order derivatives not only allow more complex optimization techniques in ML/DL that takes into account the hessian of a loss function, but also efficiently extends the library's utility beyond machine learning into other domains, such as physics simulations, optimization problems requiring curvature information (Hessians), or even ray tracing in computer graphics utilizing H2MC (Hessian Hamiltonian Monte Carlo).

6.2.3 Extensibility to Custom Operations

Autograd is designed as an extensible library of operators. This modularity allows users to easily define gradients for custom functions or integrate external numerical libraries (by wrapping operations with the primitive decorator) and extend the Autograd library.

As discussed in Section 6.1.3, this extensibility is achieved through the `@primitive` decorator and the `defvjp` (Define Vector-Jacobian Product) function. Just as the built-in multiplication operator registers its gradient rule, a user can register a new operation by wrapping a Python function as a primitive and defining its derivative of its output w.r.t. to its input. To Autograd, there is no distinction between a core numpy function and a user-defined primitive.

Example: Numerical Stability

The `logsumexp` operation can easily overflow if implemented naively. By defining a custom VJP, we can provide a mathematically precise gradient. The stable implementation uses the identity:

$$\log\left(\sum_i \exp(x_i)\right) = \log\left(\sum_i \exp(x_i - c)\right) + \log(\exp(c))$$

Listing 15: Defining a custom gradient for logsumexp

```
@primitive
def logsumexp(x):
    return np.max(x) + np.log(np.sum(np.exp(x - np.max(x))))

def logsumexp_vjp(ans, x):
    return lambda g: g * np.exp(x - ans) # 'ans' is the cached local output

# Register VJP, Autograd will use this Python function during the backward pass.
defvjp(logsumexp, logsumexp_vjp)
```

6.2.4 Handling Complex Data Types

Our pedagogical implementation, `simpleautodiff`, operates under a significant simplifying assumption: all variables are scalar numbers. Under such a significant constraint, the backward pass is straightforward because the accumulation of gradients solely relies on the Python operator (`+`).

However, a production-ready library like Autograd must support the more complex types of the Python ecosystem. Users expect to differentiate functions involving complex numbers, and arbitrary containers like tuples or dictionaries.

When a variable is used by multiple downstream operations (fan-out), the gradients from those branches must be summed in the backward pass. The naive approach fails immediately for container types. Python does not define a standard `+` operator for dictionaries or arbitrary objects, attempting to execute `dict1 + dict2` raises a `TypeError`.

To resolve this, Autograd introduces the VSpace (Vector Space) abstraction. It essentially treats all data types as a vector space, and defines the 2 fundamental operations in a vector space: vector addition and scalar multiplication. By this abstraction, we can perform, for example, the addition of

2 dictionaries. Consider the following example where a dictionary parameter is used in two separate branches of a calculation.

Listing 16: Branching logic requiring gradient accumulation on a dictionary

```
def f(params):
    # return x^2
    return params['x'] * params['x']
grads = grad(f)({'x': 3.0})
```

Autograd treats dictionaries as vector spaces where keys represent dimensions. The definition of addition: Iterates over keys and sums corresponding values recursively ($d_{new}[k] = d_1[k] + d_2[k]$). The "DictVSpace" class defines how to manipulate these structures. An example walkthrough is illustated below:

1. **Trace**: Accessing "x" twice creates two "extraction" nodes in the computational graph, both stemming from the same input node "x".

2. **Gradients**: The gradient of a extraction node is simply the identity function, namely it leaves the incoming gradient unchanged. In this case, the gradient returned from the extraction node is the gradient from the operation `params['x'] * params['x']`, which is `params['x'] = 3.0` but structured as {'x' = 3.0}

3. **Accumulation**: "DictVSpace" merges these gradients into a single gradient dictionary "{'x': 6.0}".

This abstraction allows "DictVSpace" to handle nested structures recursively. A dictionary can contain arrays (handled by "ArrayVSpace") or other dictionaries (handled by "DictVSpace"), enabling automatic differentiation of complex, nested parameter structures common in machine learning without manual flattening. The vspace transform happens during the backward pass:

Listing 17: Location of VSpace Transformation

```
def backward_pass(g, end_node):
    ...
    outgrads[parent] = add_outgrads(outgrads.get(parent), ingrad) # Accumulation
def add_outgrads(prev_g_flagged, g):
    ...
    vs = vspace(g) # Transformation into VSpace
    return vs.add(prev_g, g), True
class DictVSpace(ContainerVSpace):
    ...
    def _map(self, f, *args):
        return {k: f(vs, *[x[k] for x in args]) for k, vs in self.shape.items()}
    def _add(self, xs, ys):
        return self._map(lambda vs, x, y: vs._add(x, y), xs, ys)
```

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.

J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/jax-ml/jax`.

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.

D. Maclaurin. *Modeling, Inference and Optimization with Composable Differentiable Procedures*. PhD thesis, Harvard University, April 2016.

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.

The Theano Development Team, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, et al. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016. URL `https://arxiv.org/abs/1605.02688`.