深入理解并实现基本的自动微分（Automatic Differentiation）算法

杨其臻

Sep 18, 2025

自动微分（Automatic Differentiation，AD）是现代深度学习框架的核心技术，它使得梯度计算变得高效且精确。本文将带领您从第一性原理出发，深入探讨自动微分的两种核心模式：前向模式与反向模式。我们将从数学基础讲起，通过 Python 代码一步步实现一个基于计算图的反向模式自动微分微型库，并理解其如何高效地计算梯度。文章将避免使用图片和列表，专注于文字描述和代码解读，以确保内容的清晰性和专业性。
在机器学习和深度学习中，模型的训练过程本质上是通过梯度下降算法来优化参数。梯度计算是关键步骤，但传统方法存在显著局限性。手动求导虽然精确，但耗时且容易出错，尤其是当模型结构变化时，需要重新推导公式。数值微分（如有限差分法）虽然自动化，但计算成本高，且存在精度问题，如舍入误差和截断误差，无法达到机器精度。自动微分结合了数值微分的自动化性和符号微分的精确性，能够高效计算梯度，且精度极高。本文的目标是抛开大型框架如 PyTorch 或 TensorFlow，从零实现一个简单的自动微分引擎，以透彻理解其工作原理。

1　自动微分的基本思想

自动微分的核心思想是将任何复杂函数分解为一系列基本初等函数（Primitive Functions）的组合，例如加法、乘法、指数函数和对数函数等。这个过程可以通过计算图（Computational Graph）来表示，其中节点代表中间变量或值，边代表基本运算操作。计算图是一个有向无环图（DAG），它清晰地展示了函数计算的依赖关系。例如，函数 `f(x, y) = exp(x) + x * y` 可以分解为多个步骤：首先计算 `exp(x)`，然后计算 `x * y`，最后将两者相加。数学上，自动微分依赖于链式法则（Chain Rule），这是多元微积分中的基本原理，用于计算复合函数的导数。链式法则允许我们将梯度计算分解为每个基本操作的局部梯度乘积，从而高效地传播梯度。

2　两种模式的深度解析

自动微分主要有两种模式：前向模式和反向模式。每种模式有其适用场景和特点。

2.1　前向模式自动微分

前向模式自动微分从输入变量开始，沿着计算图向前推进，同时计算当前变量对某一个输入变量的导数。这种模式的核心是二元数（Dual Number）理论，二元数将函数值和高阶微分的计算合并在一起，通过扩展数字系统来同时跟踪值和导数。前向模式的特点是适合输入变量少、输出变量多的情况，因为计算梯度的时间与输入维度成正比。例如，对于一个函数 `f(x, y)`，前向模式会逐个计算对每个输入的偏导数。在实际计算中，我们初始

化输入变量的导数为 1（对于自身）或 0（对于其他），然后逐步应用链式法则向前传播。

2.2 反向模式自动微分

反向模式自动微分是本文的重点，因为它更适用于深度学习场景，其中输入维度高（如大量参数），输出维度低（如损失函数为标量）。反向模式分为两个阶段：首先进行前向计算（Forward Pass）得到所有中间变量的值，然后从最终输出开始，反向遍历计算图，应用链式法则计算输出对所有输入变量的导数。关键概念是雅可比向量积（Jacobian-vector Product），它允许我们高效地累积梯度。反向模式的特点是计算梯度的时间与输出维度成正比，这使得它在高维输入情况下非常高效。反向传播（Backpropagation）算法是反向模式的一个特例，广泛应用于神经网络训练。在反向过程中，每个节点接收来自后续节点的梯度，并将其乘以局部雅可比矩阵，然后累加到父节点的梯度上。

3 动手实现：构建一个微型自动微分库

现在，我们将动手实现一个基于反向模式的自动微分微型库。这个库的核心是一个 Tensor 类，它存储数据、梯度、父节点和操作信息。我们将重载基本运算符来实现前向计算和反向传播。

3.1 设计核心类：Tensor 类

首先，我们定义 Tensor 类，它具有以下属性：data 存储数值，grad 存储梯度，_prev 存储父节点（用于构建计算图），_op 存储产生该节点的操作。此外，我们实现 backward() 方法来触发反向传播。

```
class Tensor:
    def __init__(self, data, _children=(), _op=''):
        self.data = data
        self.grad = 0.0
        self._prev = set(_children)
        self._op = _op
        self._backward = lambda: None

    def backward(self):
        topo = []
        visited = set()
        def build_topo(v):
            if v not in visited:
                visited.add(v)
                for child in v._prev:
                    build_topo(child)
                topo.append(v)
        build_topo(self)
        self.grad = 1.0
```

```
        for v in reversed(topo):
            v._backward()
```

在这个代码中，Tensor 类初始化时设置数据、梯度、父节点和操作。backward 方法使用深度优先搜索（DFS）构建拓扑排序，然后反向遍历节点，调用每个节点的 _backward 方法来计算梯度。拓扑排序确保我们以正确的顺序处理节点，避免循环依赖。

3.2 实现基本运算操作

接下来，我们重载基本运算符，如加法、乘法、指数函数等。每个操作需要实现前向计算和反向传播规则。

```
def add(self, other):
    other = other if isinstance(other, Tensor) else Tensor(other)
    out = Tensor(self.data + other.data, (self, other), '+')
    def _backward():
        self.grad += out.grad
        other.grad += out.grad
    out._backward = _backward
    return out

Tensor.__add__ = add
```

加法操作的前向计算简单地将两个张量的数据相加。反向传播规则是：梯度从输出节点 out 传播回输入节点 self 和 other，由于加法操作的导数为 1，所以直接将 out.grad 累加到输入节点的梯度上。这体现了链式法则的应用。

```
def mul(self, other):
    other = other if isinstance(other, Tensor) else Tensor(other)
    out = Tensor(self.data * other.data, (self, other), '*')
    def _backward():
        self.grad += other.data * out.grad
        other.grad += self.data * out.grad
    out._backward = _backward
    return out

Tensor.__mul__ = mul
```

乘法操作的前向计算将数据相乘。反向传播规则更复杂：对于 self，梯度是 other.data * out.grad，因为乘法的偏导数是另一个变量的值；同样对于 other，梯度是 self.data * out.grad。这确保了梯度正确传播。

类似地，我们可以实现其他操作，如指数函数。

```
def exp(self):
```

```
    out = Tensor(math.exp(self.data), (self,), 'exp')
    def _backward():
        self.grad += out.data * out.grad
    out._backward = _backward
    return out

Tensor.exp = exp
```

指数函数的前向计算使用 math.exp。反向传播规则是：梯度是输出值乘以输出梯度，因为指数函数的导数是其自身。这通过 out.data * out.grad 实现。

3.3 核心引擎：反向传播算法

反向传播算法在 backward 方法中实现。它首先构建拓扑排序来确保节点按依赖顺序处理，然后从输出节点开始，梯度初始化为 1.0（因为输出对自身的导数为 1），反向调用每个节点的 _backward 方法。这个过程应用链式法则，将梯度累加到父节点。

3.4 代码演示与测试

让我们测试这个微型库。例如，计算函数 f(x) = exp(x) + x * x 在 x=2 处的梯度。

```
x = Tensor(2.0)
y = x.exp() + x * x
y.backward()
print(x.grad) # 应该输出 exp(2) + 2*2 的导数，即 exp(2) + 4
```

在这个测试中，我们创建张量 x，计算 y，然后调用 backward。梯度计算应该正确，我们可以与手动计算对比：f'(x) = exp(x) + 2*x，在 x=2 时，值为 exp(2) + 4，约等于 7.389 + 4 = 11.389。我们的库应该输出类似值。

4 现代框架中的自动微分

在现代深度学习框架中，自动微分有静态图和动态图两种实现方式。静态图（如 TensorFlow 1.x）先定义计算图再执行，图是固定的；动态图（如 PyTorch 和 TensorFlow 2.x Eager Mode）边定义边执行，图是即时构建的。我们的实现属于动态图模式。现代框架还进行了大量优化，如内核融合、高效内存管理和数据结构，以提高性能和可扩展性。

本文深入探讨了自动微分的核心思想、两种模式的区别，以及如何从零实现一个反向模式自动微分库。通过代码实现，我们理解了梯度计算如何通过计算图和链式法则高效完成。自动微分是深度学习的基础，掌握其原理有助于更深入地理解模型训练过程。展望未来，可以扩展实现高阶导数、处理控制流（如 if 和 while 语句），或应用于随机梯度估计等领域。鼓励读者在此基础上继续探索，以加深理解。

5 参考资料与延伸阅读

建议阅读一些经典资源，如 Baydin et al. 的论文「Automatic Differentiation in Machine Learning: a Survey」，以及书籍如「Deep Learning」by Ian Goodfellow et al.。在线教程如 PyTorch 官方文档也提供了丰富资料。