自动微分

自动求函数的微分（梯度，高阶梯度）在数值优化领域是相当重要的。目前包括符号微分，数值微分以及自动微分方法。数值微分强调一开始直接代入数值近似求解。符号微分强调直接对代数进行求解，最后才代入问题数值，求微分的解析解。而自动微分将符号微分法应用于最基本的算子，比如常数，幂函数，指数函数，对数函数，三角函数等，然后代入数值，保留中间结果，最后再应用于整个函数。所以自动微分方法可以看成是数值方法和符号方法（解析方法）的混合。

自动微分有两种形式：前向模式 (forward mode) 和反向模式 (reverse mode)。前向模式是在计算图前向传播的同时计算微分。反向模式需要对计算图进行一次正向计算，得出输出值，再进行反向传播。

当输出的维度大于输入的时候，适宜使用前向模式微分；当输出维度远远小于输入的时候，适宜使用反向模式微分。在深度学习里由于变量超多，loss 只有一个，因此都是采用的反向模式的微分。下面来具体介绍这两种方法。

现有的自动微分工具

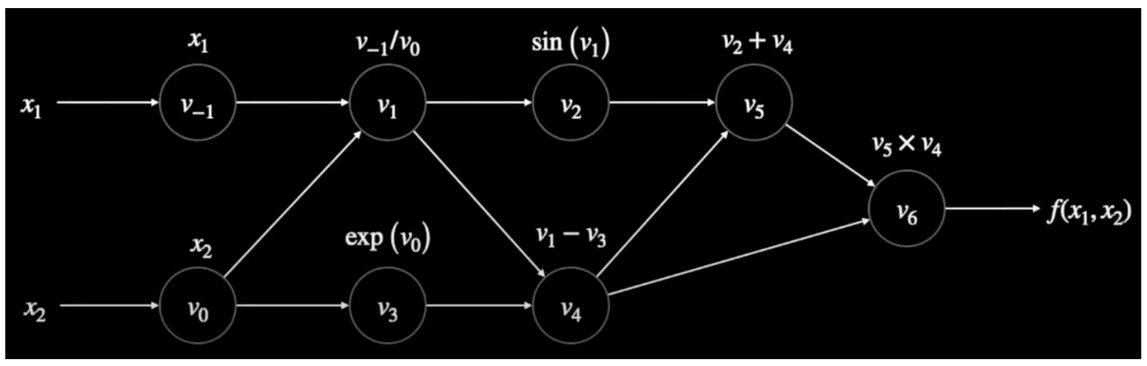
- 推荐使用 Pytorch，参考 https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html

正向模式

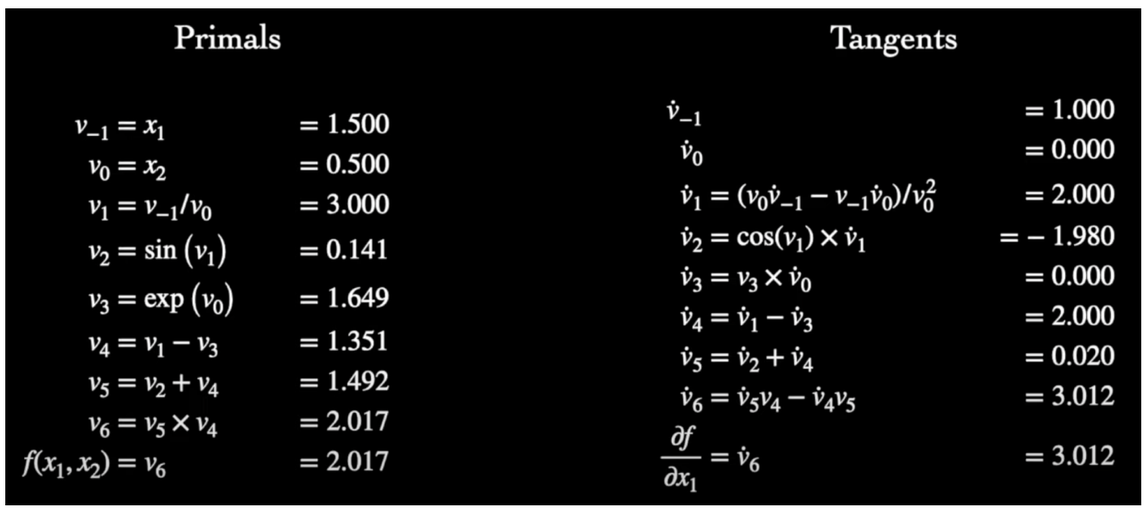
看下面一个例子：

$$f(x_1, x_2) = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2}\right] \times \left[\frac{x_1}{x_2} - e^{x_2}\right]$$

我们要计算 $x_1 = 1.5, x_2 = 0.5$ 时的偏导数 $\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}$。整个计算图如下图所示：



给定 x_1, x_2 的值我们就能根据这个图算出最后的函数值 $f(x_1, x_2)$。在前向计算的时候，我们可以同步的算出每个节点对 x_1 或者 x_2 的偏导数（一次只能求一种，要么对 x_1 求偏导，要么对 x_2 求偏导）。如下图所示，其中所有导数都是变量对 x_1 求的偏导数的值。



具体实现方式可以采用运算符重载的方式实现。那么只需要前向把计算图算完，对应的偏导数就求出来了。但需要注意的是，一次前向计算只能计算函数对一个自变量的偏导。在这个例子中就是一次只能计算 $\frac{\partial f}{\partial x_1}$。如果有 m 个输出，那么前向计算一次就能计算，$\frac{\partial f_1}{\partial x_1}, \frac{\partial f_2}{\partial x_1}, \ldots, \frac{\partial f_m}{\partial x_1}$。这相当于一次前向计算就能计算雅可比矩阵的一列：

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & & & \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

所以，对于 $f: R^n \to R^m$ 的偏导数计算。前向计算非常适合自变量很少，但是输出很多的情况，即 $n << m$ 的情况。实现正向模式的自动微分通常采用运算符重载的方法，如下所示，很好理解。

In [ ]:
```
import numpy as np
```

In [ ]:
```
class Variable:
    def __init__(self, val, dot=0.0):
        self.val = val
        self.dot = dot

    def __add__(self, other):
        res = Variable(self.val + other.val)
        res.dot = self.dot + other.dot
        return res

    def __sub__(self, other):
        res = Variable(self.val - other.val)
        res.dot = self.dot - other.dot
        return res

    def __mul__(self, other):
        res = Variable(self.val * other.val)
        res.dot = self.dot * other.val + self.val * other.dot
        return res

    def __truediv__(self, other):
        res = Variable(self.val / other.val)
        res.dot = (self.dot * other.val - self.val * other.dot) / (other.val ** 2)
        return res

    def __repr__(self):
        return "值为:{:.3f}, 偏导数为:{:.3f}".format(self.val, self.dot)

def sin(var):
    res = Variable(np.sin(var.val))
    res.dot = np.cos(var.val) * var.dot
    return res

def exp(var):
    res = Variable(np.exp(var.val))
    res.dot = np.exp(var.val) * var.dot
    return res
```

In [ ]:
```
def f(v_1, v0):
    v1 = v_1 / v0
```

```
    v2 = sin(v1)
    v3 = exp(v0)
    v4 = v1 - v3
    v5 = v2 + v4
    v6 = v5 * v4
    return v6
```

计算 $\left.\frac{\partial f}{\partial x_1}\right|_{x_1=1.5,x_2=0.5}$

通过一次前向运算即可得到：

In [ ]:
```
v_1 = Variable(1.5, 1.0)
v0 = Variable(0.5, 0.0)
v6 = f(v_1, v0)
print(v6)
```

值为:2.017, 偏导数为:3.012

计算 $\left.\frac{\partial f}{\partial x_2}\right|_{x_1=1.5,x_2=0.5}$

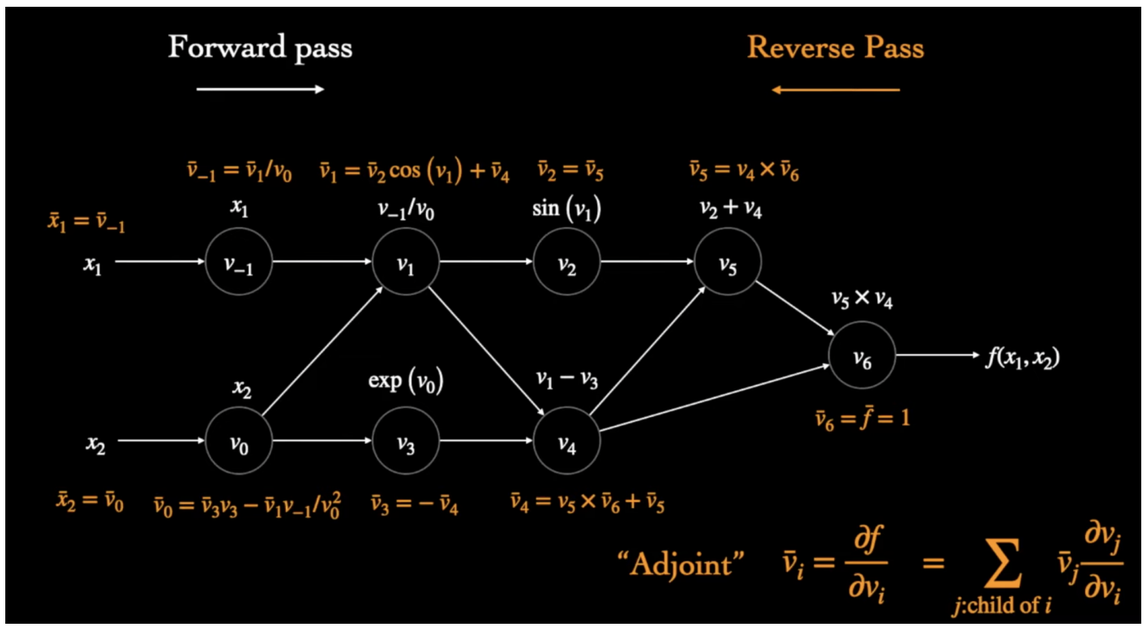
通过一次前向运算即可得到：

In [ ]:
```
v_1 = Variable(1.5, 0.0)
v0 = Variable(0.5, 1.0)
v6 = f(v_1, v0)
print(v6)
```

值为:2.017, 偏导数为:-13.724

逆向模式

而在深度学习中，神经网络的参数是非常多的，相当于 n 很多，但最终输出只有一个 loss，$m=1$。此时，$f:R^n\to R^1$，用正向模式划不来，相当于要前向计算 n 次才能得到函数 f 的梯度。下面我们来介绍逆向模式，逆向模式可以一次计算雅可比矩阵的一行，对于 $m=1$ 的情况就是直接输出梯度。计算流程如下所示：



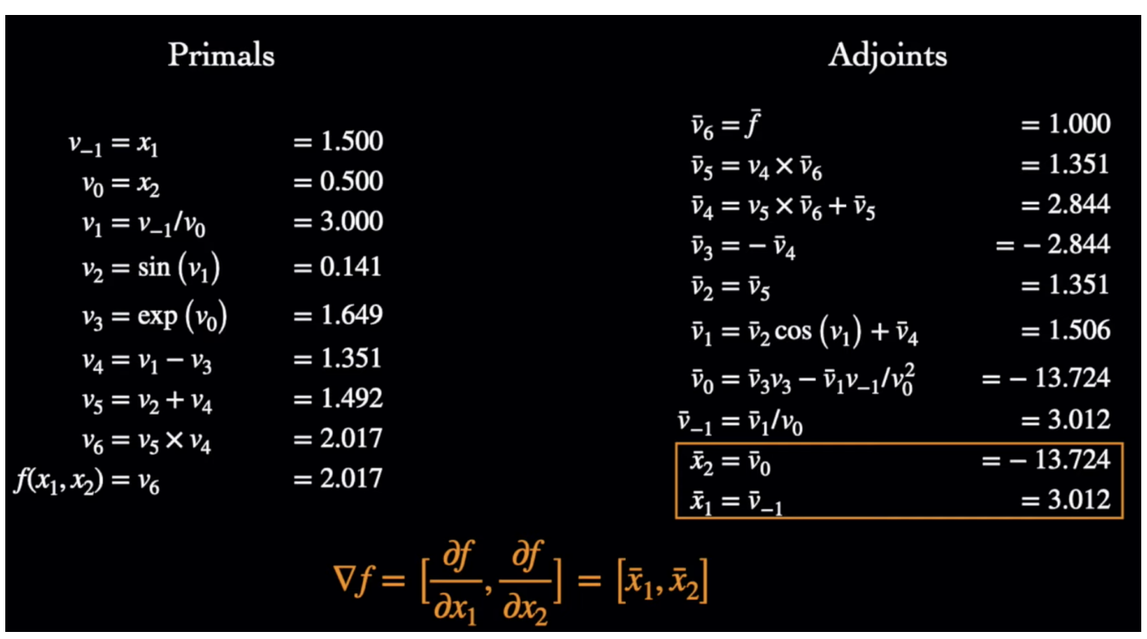
首先通过前向计算，计算所有节点的值。然后，反向计算每个节点上，函数对该节点的偏导数。对于 v6 节点来说就是计算

$$\frac{\partial f}{\partial v_6}=1$$

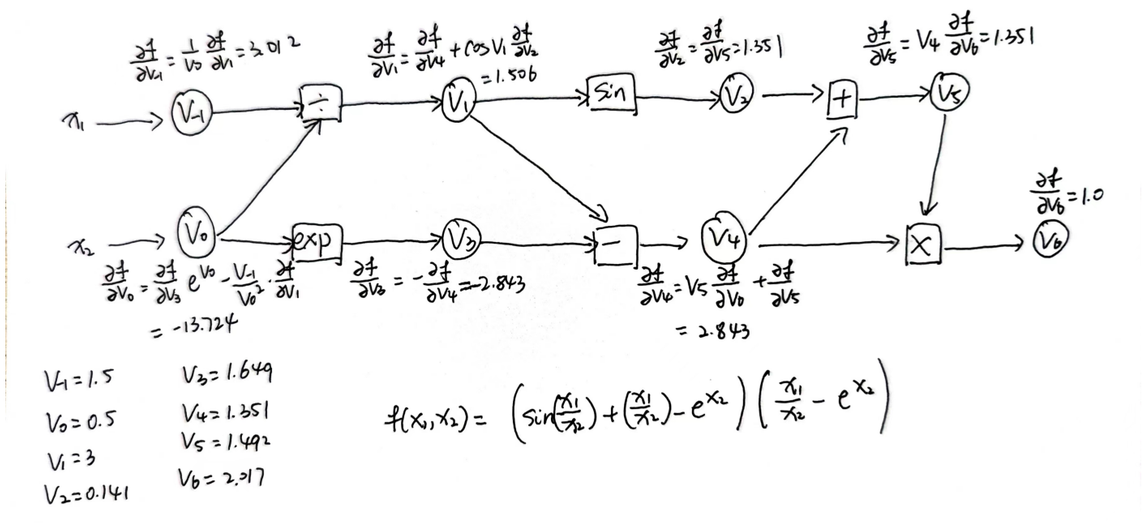
对于 v5 节点来说就是计算

$$\frac{\partial f}{\partial v_5}=v_4*\frac{\partial f}{\partial v_6}$$

然后一直往前算，直到算到 $\frac{\partial f}{\partial v_0}$。整个计算过程如下图所示，这里用 $\bar{v}_6$ 表示 $\partial f/\partial v_6$



上面这个图是视频里的原图，$\frac{\partial f}{\partial v_0}=\bar{v}_0$ 的计算过程有点错误。我这又重新画了一下，更加清晰。



所以，逆向模式有两个步骤，先前向计算每个节点的值，再逆向计算函数对每个节点的偏导数。对于 $f:R^n\to R^m$ 的偏导数计算。逆向模式非常适合自变量很多，但是输出很少的情况，即 $n>>m$ 的情况。Pytorch 中正是采用的这种方法，反向传播就是这么来的！先正向，后逆向情况的代码略有有些复杂。需要记录每一步的操作。具体的看代码。

In [ ]:
```python
from typing import List, NamedTuple, Callable, Dict, Optional
import numpy as np

# 全局记录每个变量的名字和所有的操作
class Tape(NamedTuple):
    """用于记录一次操作. 包括输入输出和一个可调用的反向传播函数."""
    inputs: List[str]
    outputs: List[str]
    # 计算所有的 partial_outputs / partial_inputs
    propagate: Callable

_NAME = 1

def fresh_name():
    global _NAME
    name = f"v{_NAME}"
    _NAME += 1
    return name

# 顺序记录前向计算中所有的操作: 加减乘除等
GRADIENT_TAPE: List[Tape] = []


# 清空所有的操作
def reset_tape():
    global _NAME
    _NAME = 1
    GRADIENT_TAPE.clear()


# 重载定义记录操作的变量
class Variable:
    def __init__(self, val: float, name: str=None):
        self.val = val
        self.name = name or fresh_name()

    def __repr__(self):
        return repr(self.val)

    # 返回一个常数
    @staticmethod
    def constant(val, name=None):
        var = Variable(val, name)
        print(f"{var.name} = {val}")
        return var

    # 重载的一些操作
    def __mul__(self, other):
        return ops_mul(self, other)

    def __truediv__(self, other):
        return ops_truediv(self, other)

    def __add__(self, other):
        return ops_add(self, other)

    def __sub__(self, other):
        return ops_sub(self, other)


def ops_mul(self: Variable, other: Variable):
    # forward
    x = Variable(self.val * other.val)
    print(f"{x.name} = {self.name} * {other.name}")

    # backward
    def propagate(df_doutputs):
        (df_dx,) = df_doutputs  # 下一层的 偏 f 偏 x
        dx_dself = other  # partial derivate of x = self*other
        dx_dother = self  # partial derivate of x = self*other
        df_dself = df_dx * dx_dself
        df_dother = df_dx * dx_dother
        dl_dinputs = [df_dself, df_dother]
        return dl_dinputs

    # 记录这个操作的输入输出
    tape = Tape(inputs=[self.name, other.name], outputs=[x.name], propagate=propagate)
    GRADIENT_TAPE.append(tape)
    return x

def ops_truediv(self: Variable, other: Variable):
    # forward
    x = Variable(self.val / other.val)
    print(f"{x.name} = {self.name} / {other.name}")

    # backward
    def propagate(df_doutputs):
        (df_dx,) = df_doutputs  # 下一层的 偏 f 偏 x
        dx_dself = Variable.constant(1.0) / other  # partial derivate of x = self/other
        dx_dother = Variable.constant(0.0) - self / (other * other)  # partial derivate of x = selfother
        df_dself = df_dx * dx_dself
        df_dother = df_dx * dx_dother
        dl_dinputs = [df_dself, df_dother]
        return dl_dinputs

    # 记录这个操作的输入输出
    tape = Tape(inputs=[self.name, other.name], outputs=[x.name], propagate=propagate)
    GRADIENT_TAPE.append(tape)
    return x

def ops_add(self, other):
    x = Variable(self.val + other.val)
    print(f"{x.name} = {self.name} + {other.name}")

    def propagate(df_doutputs):
        (df_dx,) = df_doutputs  # 上一个节点传来的 df / do
        dx_dself = Variable(1.0)
        dx_dother = Variable(1.0)
        df_dself = df_dx * dx_dself
        df_dother = df_dx * dx_dother
        return [df_dself, df_dother]

    # record the input and output of the op
    tape = Tape(inputs=[self.name, other.name], outputs=[x.name], propagate=propagate)
    GRADIENT_TAPE.append(tape)
    return x


def ops_sub(self, other):
    x = Variable(self.val - other.val)
    print(f"{x.name} = {self.name} - {other.name}")

    def propagate(df_doutputs):
        (df_dx,) = df_doutputs
        dx_dself = Variable(1.0)
```

```
        dx_dother = Variable(-1.0)
        df_dself = df_dx * dx_dself
        df_dother = df_dx * dx_dother
        return [df_dself, df_dother]

    # record the input and output of the op
    tape = Tape(inputs=[self.name, other.name], outputs=[x.name], propagate=propagate)
    GRADIENT_TAPE.append(tape)
    return x


def sin(self):
    x = Variable(np.sin(self.val))
    print(f"{x.name} = sin({self.name})")

    def propagate(df_doutputs):
        (df_dx,) = df_doutputs
        dx_dself = Variable(np.cos(self.val))
        df_dself = df_dx * dx_dself
        return [df_dself]

    # record the input and output of the op
    tape = Tape(inputs=[self.name], outputs=[x.name], propagate=propagate)
    GRADIENT_TAPE.append(tape)
    return x


def log(self):
    x = Variable(np.log(self.val))
    print(f"{x.name} = log({self.name})")

    def propagate(df_doutputs):
        (df_dx,) = df_doutputs
        dx_dself = Variable(1 / self.val)
        df_dself = df_dx * dx_dself
        return [df_dself]

    # record the input and output of the op
    tape = Tape(inputs=[self.name], outputs=[x.name], propagate=propagate)
    GRADIENT_TAPE.append(tape)
    return x

def exp(self):
    x = Variable(np.exp(self.val))
    print(f"{x.name} = exp({self.name})")

    def propagate(df_doutputs):
        (df_dx,) = df_doutputs
        dx_dself = Variable(x.val)
        df_dself = df_dx * dx_dself
        return [df_dself]

    # record the input and output of the op
    tape = Tape(inputs=[self.name], outputs=[x.name], propagate=propagate)
    GRADIENT_TAPE.append(tape)
    return x
```

In [ ]:

```
# 反向模式求解梯度，根据图片来理解
def grad(l, results):
    df_d = {} # map dL/dX for all values X
    df_d[l.name] = Variable(1.)
    print("df_d", df_d)

    def gather_grad(entries):
        return [df_d[entry] if entry in df_d else None for entry in entries]

    for entry in reversed(GRADIENT_TAPE):
        print(entry)
        df_doutputs = gather_grad(entry.outputs)
        df_dinputs = entry.propagate(df_doutputs)
        # 根据变量的名字，把梯度给累加上去
        for input, df_dinput in zip(entry.inputs, df_dinputs):
            if input not in df_d:
                df_d[input] = df_dinput
            else:
                df_d[input] += df_dinput

    for name, value in df_d.items():
        print(f'd{l.name}_d{name} = {value.name}')

    return gather_grad(result.name for result in results)
```

前向传播

In [ ]:

```
reset_tape()
v_1 = Variable.constant(1.5, name='v_1')
v0 = Variable.constant(0.5, name='v0')

def func(v_1, v0):
    v1 = v_1 / v0
    v2 = sin(v1)
    v3 = exp(v0)
    v4 = v1 - v3
    v5 = v2 + v4
    v6 = v5 * v4
    return v6

v6 = func(v_1, v0)

print(v6)
```

```
v_1 = 1.5
v0 = 0.5
v1 = v_1 / v0
v2 = sin(v1)
v3 = exp(v0)
v4 = v1 - v3
v5 = v2 + v4
v6 = v5 * v4
2.016646669428201
```

反向传播求解梯度，只能 backward 一次

In [ ]:

```
dx, dy = grad(v6, [v_1, v0])
print("df_dx1", dx)
print("df_dx2", dy)
```

```
df_d {'v6': 1.0}
Tape(inputs=['v5', 'v4'], outputs=['v6'], propagate=<function ops_mul.<locals>.propagate at 0x10ed321f0>)
v8 = v7 * v4
v9 = v7 * v5
Tape(inputs=['v2', 'v4'], outputs=['v5'], propagate=<function ops_add.<locals>.propagate at 0x10ed32160>)
v12 = v8 * v10
v13 = v8 * v11
v14 = v9 + v13
Tape(inputs=['v1', 'v3'], outputs=['v4'], propagate=<function ops_sub.<locals>.propagate at 0x10ed320d0>)
v17 = v14 * v15
v18 = v14 * v16
Tape(inputs=['v0'], outputs=['v3'], propagate=<function exp.<locals>.propagate at 0x10ed32040>)
v20 = v18 * v19
Tape(inputs=['v1'], outputs=['v2'], propagate=<function sin.<locals>.propagate at 0x10ec6af70>)
v22 = v12 * v21
v23 = v17 + v22
Tape(inputs=['v_1', 'v0'], outputs=['v1'], propagate=<function ops_truediv.<locals>.propagate at 0x10ec6ac10>)
v24 = 1.0
v25 = v24 / v0
v26 = 0.0
v27 = v0 * v0
v28 = v_1 / v27
v29 = v26 - v28
v30 = v23 * v25
v31 = v23 * v29
v32 = v20 + v31
dv6_dv6 = v7
dv6_dv5 = v8
dv6_dv4 = v14
dv6_dv2 = v12
dv6_dv1 = v23
dv6_dv3 = v18
dv6_dv0 = v32
dv6_dv_1 = v30
df_dx1 3.011843327673907
df_dx2 -13.723961509314076
```