

```
In [1]: # Refs:
# 1. https://github.com/karpathy/micrograd/tree/master/micrograd
# 2. https://github.com/mattjj/autodidact
# 3. https://github.com/mattjj/autodidact/blob/master/autograd/numpy/numpy_v
from collections import namedtuple
import numpy as np

def unbroadcast(target, g, axis=0):
    """Remove broadcasted dimensions by summing along them.
    When computing gradients of a broadcasted value, this is the right thing
    do when computing the total derivative and accounting for cloning.
    """
    while np.ndim(g) > np.ndim(target):
        g = g.sum(axis=axis)
    for axis, size in enumerate(target.shape):
        if size == 1:
            g = g.sum(axis=axis, keepdims=True)
    if np.iscomplexobj(g) and not np.iscomplex(target):
        g = g.real()
    return g

Op = namedtuple('Op', ['apply',
                        'vjp',
                        'name',
                        'nargs'])
```

Vector Jacobian Product for addition

$$\mathbf{f}(\mathbf{a}, \mathbf{b}) = \mathbf{a} + \mathbf{b}$$

where $\mathbf{a}, \mathbf{b}, \mathbf{f} \in \mathbb{R}^n$

Let $l(\mathbf{f}(\mathbf{a}, \mathbf{b})) \in \mathbb{R}$ be the eventual scalar output. We find $\frac{\partial l}{\partial \mathbf{a}}$ and $\frac{\partial l}{\partial \mathbf{b}}$ for Vector Jacobian product.

$$\frac{\partial}{\partial \mathbf{a}} l(\mathbf{f}(\mathbf{a}, \mathbf{b})) = \frac{\partial l}{\partial \mathbf{f}} \frac{\partial}{\partial \mathbf{a}} (\mathbf{a} + \mathbf{b}) = \frac{\partial l}{\partial \mathbf{f}} (\mathbf{I}_{n \times n} + \mathbf{0}_{n \times n}) = \frac{\partial l}{\partial \mathbf{f}}$$

Similarly,

$$\frac{\partial}{\partial \mathbf{b}} l(\mathbf{f}(\mathbf{a}, \mathbf{b})) = \frac{\partial l}{\partial \mathbf{f}}$$

```
In [2]: def add_vjp(dldf, a, b):
        dlda = unbroadcast(a, dldf)
        dlbd = unbroadcast(b, dldf)
        return dlda, dlbd
```

```
add = Op(
    apply=np.add,
    vjp=add_vjp,
    name='+',
    nargs=2)
```

VJP for element-wise multiplication

$$f(\alpha, \beta) = \alpha\beta$$

where $\alpha, \beta, f \in \mathbb{R}$

Let $l(f(\alpha, \beta)) \in \mathbb{R}$ be the eventual scalar output. We find $\frac{\partial l}{\partial \alpha}$ and $\frac{\partial l}{\partial \beta}$ for Vector Jacobian product.

$$\frac{\partial}{\partial \alpha} l(f(\alpha, \beta)) = \frac{\partial l}{\partial f} \frac{\partial}{\partial \alpha} (\alpha\beta) = \frac{\partial l}{\partial f} \beta$$

$$\frac{\partial}{\partial \beta} l(f(\alpha, \beta)) = \frac{\partial l}{\partial f} \frac{\partial}{\partial \beta} (\alpha\beta) = \frac{\partial l}{\partial f} \alpha$$

```
In [3]: def mul_vjp(dldf, a, b):
        dl_da = unbroadcast(a, dldf * b)
        dl_db = unbroadcast(b, dldf * a)
        return dl_da, dl_db

mul = Op(
    apply=np.multiply,
    vjp=mul_vjp,
    name='*',
    nargs=2)
```

VJP for matrix-matrix, matrix-vector and vector-vector multiplication

Case 1: VJP for vector-vector multiplication

$$f(\mathbf{a}, \mathbf{b}) = \mathbf{a}^\top \mathbf{b}$$

where $f \in \mathbb{R}$, and $\mathbf{b}, \mathbf{a} \in \mathbb{R}^n$

Let $l(f(\mathbf{a}, \mathbf{b})) \in \mathbb{R}$ be the eventual scalar output. We find $\frac{\partial l}{\partial \mathbf{a}}$ and $\frac{\partial l}{\partial \mathbf{b}}$ for Vector Jacobian product.

$$\frac{\partial}{\partial \mathbf{a}} l(f(\mathbf{a}, \mathbf{b})) = \frac{\partial l}{\partial f} \frac{\partial}{\partial \mathbf{a}} (\mathbf{a}^\top \mathbf{b}) = \frac{\partial l}{\partial f} \mathbf{b}^\top$$

Similarly,

$$\frac{\partial}{\partial \mathbf{b}} l(f(\mathbf{a}, \mathbf{b})) = \frac{\partial l}{\partial f} \mathbf{a}^\top$$

Case 2: VJP for matrix-vector multiplication

Let

$$\mathbf{f}(\mathbf{A}, \mathbf{b}) = \mathbf{A}\mathbf{b}$$

where $\mathbf{f} \in \mathbb{R}^m$, $\mathbf{b} \in \mathbb{R}^n$, and $\mathbf{A} \in \mathbb{R}^{m \times n}$

Let $l(\mathbf{f}(\mathbf{A}, \mathbf{b})) \in \mathbb{R}$ be the eventual scalar output. We want to find $\frac{\partial l}{\partial \mathbf{A}}$ and $\frac{\partial l}{\partial \mathbf{b}}$ for Vector Jacobian product.

Let

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}$$

, where each $\mathbf{a}_i^\top \in \mathbb{R}^{1 \times n}$ and $a_{ij} \in \mathbb{R}$.

Define matrix derivative of scalar to be:

$$\frac{\partial l}{\partial \mathbf{A}} = \begin{bmatrix} \frac{\partial l}{\partial a_{11}} & \frac{\partial l}{\partial a_{12}} & \dots & \frac{\partial l}{\partial a_{1n}} \\ \frac{\partial l}{\partial a_{21}} & \frac{\partial l}{\partial a_{22}} & \dots & \frac{\partial l}{\partial a_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial l}{\partial a_{m1}} & \frac{\partial l}{\partial a_{m2}} & \dots & \frac{\partial l}{\partial a_{mn}} \end{bmatrix} = \begin{bmatrix} \frac{\partial l}{\partial \mathbf{a}_1} \\ \frac{\partial l}{\partial \mathbf{a}_2} \\ \vdots \\ \frac{\partial l}{\partial \mathbf{a}_m} \end{bmatrix}$$

$$\frac{\partial}{\partial \mathbf{A}} l(\mathbf{f}(\mathbf{a}, \mathbf{b})) = \frac{\partial l}{\partial \mathbf{f}} \frac{\partial}{\partial \mathbf{A}} (\mathbf{A}\mathbf{b})$$

.

Note that

$$\mathbf{A}\mathbf{b} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{b} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b} \\ \mathbf{a}_2^\top \mathbf{b} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{b} \end{bmatrix}$$

Since $\mathbf{a}_i^\top \mathbf{b}$ is a scalar, it is easier to find its derivative with respect to the matrix \mathbf{A} .

$$\frac{\partial}{\partial \mathbf{A}} \mathbf{a}_i^\top \mathbf{b} = \begin{bmatrix} \frac{\partial \mathbf{a}_i^\top \mathbf{b}}{\partial \mathbf{a}_1} \\ \frac{\partial \mathbf{a}_i^\top \mathbf{b}}{\partial \mathbf{a}_2} \\ \vdots \\ \frac{\partial \mathbf{a}_i^\top \mathbf{b}}{\partial \mathbf{a}_i} \\ \vdots \\ \frac{\partial \mathbf{a}_i^\top \mathbf{b}}{\partial \mathbf{a}_m} \end{bmatrix} = \begin{bmatrix} \mathbf{0}_n^\top \\ \mathbf{0}_n^\top \\ \vdots \\ \mathbf{b}^\top \\ \vdots \\ \mathbf{0}_n^\top \end{bmatrix} \in \mathbb{R}^{m \times n}$$

Let

$$\frac{\partial l}{\partial \mathbf{f}} = \begin{bmatrix} \frac{\partial l}{\partial f_1} & \frac{\partial l}{\partial f_2} & \cdots & \frac{\partial l}{\partial f_m} \end{bmatrix}$$

Then

$$\frac{\partial l}{\partial \mathbf{f}} \frac{\partial}{\partial \mathbf{A}} \mathbf{a}_i^\top \mathbf{b} = \begin{bmatrix} \frac{\partial l}{\partial f_1} & \frac{\partial l}{\partial f_2} & \cdots & \frac{\partial l}{\partial f_m} \end{bmatrix} \begin{bmatrix} \mathbf{0}_n^\top \\ \mathbf{0}_n^\top \\ \vdots \\ \mathbf{b}^\top \\ \vdots \\ \mathbf{0}_n^\top \end{bmatrix} = \frac{\partial l}{\partial f_i} \mathbf{b}^\top \in \mathbb{R}^{1 \times n}$$

Returning to our original quest for

$$\frac{\partial}{\partial \mathbf{A}} l(\mathbf{f}(\mathbf{A}, \mathbf{b})) = \frac{\partial l}{\partial \mathbf{f}} \frac{\partial}{\partial \mathbf{A}} \mathbf{A}\mathbf{b} = \frac{\partial l}{\partial \mathbf{f}} \frac{\partial}{\partial \mathbf{A}} \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b} \\ \mathbf{a}_2^\top \mathbf{b} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{b} \end{bmatrix} = \begin{bmatrix} \frac{\partial l}{\partial \mathbf{f}} \frac{\partial}{\partial \mathbf{A}} \mathbf{a}_1^\top \mathbf{b} \\ \frac{\partial l}{\partial \mathbf{f}} \frac{\partial}{\partial \mathbf{A}} \mathbf{a}_2^\top \mathbf{b} \\ \vdots \\ \frac{\partial l}{\partial \mathbf{f}} \frac{\partial}{\partial \mathbf{A}} \mathbf{a}_m^\top \mathbf{b} \end{bmatrix} = \begin{bmatrix} \frac{\partial l}{\partial f_1} \mathbf{b}^\top \\ \frac{\partial l}{\partial f_2} \mathbf{b}^\top \\ \vdots \\ \frac{\partial l}{\partial f_m} \mathbf{b}^\top \end{bmatrix}$$

Note that

$$\begin{bmatrix} \frac{\partial l}{\partial f_1} \mathbf{b}^\top \\ \frac{\partial l}{\partial f_2} \mathbf{b}^\top \\ \vdots \\ \frac{\partial l}{\partial f_m} \mathbf{b}^\top \end{bmatrix} = \begin{bmatrix} \frac{\partial l}{\partial f_1} \\ \frac{\partial l}{\partial f_2} \\ \dots \\ \frac{\partial l}{\partial f_m} \end{bmatrix} \mathbf{b}^\top = \left(\frac{\partial l}{\partial \mathbf{f}} \right)^\top \mathbf{b}^\top$$

We can group the terms inside a single transpose.

Which results in

$$\frac{\partial}{\partial \mathbf{A}} l(\mathbf{f}(\mathbf{A}, \mathbf{b})) = \left(\mathbf{b} \frac{\partial l}{\partial \mathbf{f}} \right)^\top$$

The derivative with respect to \mathbf{b} is simpler:

$$\frac{\partial}{\partial \mathbf{b}} l(\mathbf{f}(\mathbf{A}, \mathbf{b})) = \frac{\partial l}{\partial \mathbf{f}} \frac{\partial}{\partial \mathbf{b}} (\mathbf{A}\mathbf{b}) = \frac{\partial l}{\partial \mathbf{f}} \mathbf{A}$$

Case 3: VJP for matrix-matrix multiplication

Let

$$\mathbf{F}(\mathbf{A}, \mathbf{B}) = \mathbf{A}\mathbf{B}$$

where $\mathbf{F} \in \mathbb{R}^{m \times p}$, $\mathbf{B} \in \mathbb{R}^{n \times p}$, and $\mathbf{A} \in \mathbb{R}^{m \times n}$

Let $l(\mathbf{F}(\mathbf{A}, \mathbf{B})) \in \mathbb{R}$ be the eventual scalar output. We want to find $\frac{\partial l}{\partial \mathbf{A}}$ and $\frac{\partial l}{\partial \mathbf{B}}$ for Vector Jacobian product.

Note that a matrix-matrix multiplication can be written in terms horizontal stacking of matrix-vector multiplications. Specifically, write \mathbf{F} and \mathbf{B} in terms of their column vectors:

$$\mathbf{B} = [\mathbf{b}_1 \quad \mathbf{b}_2 \quad \dots \quad \mathbf{b}_p]$$

$$\mathbf{F} = [\mathbf{f}_1 \quad \mathbf{f}_2 \quad \dots \quad \mathbf{f}_p].$$

Then for all i

$$\mathbf{f}_i = \mathbf{A}\mathbf{b}_i$$

From the VJP of matrix-vector multiplication, we can write

$$\frac{\partial l}{\partial \mathbf{f}_i} \frac{\partial}{\partial \mathbf{A}} \mathbf{f}_i = \frac{\partial l}{\partial \mathbf{f}_i} \frac{\partial}{\partial \mathbf{A}} (\mathbf{A}\mathbf{b}_i) = \left(\mathbf{b}_i \frac{\partial l}{\partial \mathbf{f}_i} \right)^\top \in \mathbb{R}^{m \times n}$$

and for all $i \neq j$

$$\frac{\partial l}{\partial \mathbf{f}_j} \frac{\partial}{\partial \mathbf{A}} (\mathbf{A} \mathbf{b}_i) = \mathbf{0}_{m \times n}$$

Instead of writing $l(\mathbf{F})$, we can also write $l(\mathbf{f}_1, \mathbf{f}_2, \dots, \mathbf{f}_p)$, then by chain rule of functions with multiple arguments, we have,

$$\begin{aligned} \frac{\partial}{\partial \mathbf{A}} l(\mathbf{F}(\mathbf{A}, \mathbf{B})) &= \frac{\partial}{\partial \mathbf{A}} l(\mathbf{f}_1, \mathbf{f}_2, \dots, \mathbf{f}_p) = \frac{\partial l}{\partial \mathbf{f}_1} \frac{\partial \mathbf{f}_1}{\partial \mathbf{A}} + \frac{\partial l}{\partial \mathbf{f}_2} \frac{\partial \mathbf{f}_2}{\partial \mathbf{A}} + \dots + \frac{\partial l}{\partial \mathbf{f}_p} \frac{\partial \mathbf{f}_p}{\partial \mathbf{A}} \\ \frac{\partial}{\partial \mathbf{A}} l(\mathbf{F}(\mathbf{A}, \mathbf{B})) &= \left(\mathbf{b}_1 \frac{\partial l}{\partial \mathbf{f}_1} \right)^\top + \left(\mathbf{b}_2 \frac{\partial l}{\partial \mathbf{f}_2} \right)^\top + \dots + \left(\mathbf{b}_p \frac{\partial l}{\partial \mathbf{f}_p} \right)^\top = \left(\mathbf{b}_1 \frac{\partial l}{\partial \mathbf{f}_1} + \right. \end{aligned}$$

It turns out that some of outer products can be compactly written as matrix-matrix multiplication:

$$\mathbf{b}_1 \frac{\partial l}{\partial \mathbf{f}_1} + \mathbf{b}_2 \frac{\partial l}{\partial \mathbf{f}_2} + \dots + \mathbf{b}_p \frac{\partial l}{\partial \mathbf{f}_p} = [\mathbf{b}_1 \quad \mathbf{b}_2 \quad \dots \quad \mathbf{b}_p] \begin{bmatrix} \frac{\partial l}{\partial \mathbf{f}_1} \\ \frac{\partial l}{\partial \mathbf{f}_2} \\ \vdots \\ \frac{\partial l}{\partial \mathbf{f}_p} \end{bmatrix} = \mathbf{B} \left(\frac{\partial l}{\partial \mathbf{F}} \right)^\top$$

Hence,

$$\frac{\partial}{\partial \mathbf{A}} l(\mathbf{F}(\mathbf{A}, \mathbf{B})) = \frac{\partial l}{\partial \mathbf{F}} \mathbf{B}^\top$$

The vector Jacobian product for \mathbf{B} can be found by applying the above rule to $\mathbf{F}_2(\mathbf{A}, \mathbf{C}) = \mathbf{F}^\top(\mathbf{A}, \mathbf{B}) = \mathbf{B}^\top \mathbf{A}^\top = \mathbf{C} \mathbf{A}^\top$ where $\mathbf{C} = \mathbf{B}^\top$ and $\mathbf{F}_2 = \mathbf{F}^\top$.

$$\frac{\partial}{\partial \mathbf{C}} l(\mathbf{F}_2(\mathbf{A}, \mathbf{C})) = \frac{\partial l}{\partial \mathbf{F}_2} \mathbf{A}$$

Take transpose of both sides

$$\frac{\partial}{\partial \mathbf{C}^\top} l(\mathbf{F}_2^\top(\mathbf{A}, \mathbf{C})) = \mathbf{A}^\top \frac{\partial l}{\partial \mathbf{F}_2^\top}$$

Put back, $\mathbf{C} = \mathbf{B}^\top$ and $\mathbf{F}_2 = \mathbf{F}^\top$,

$$\frac{\partial}{\partial \mathbf{B}} l(\mathbf{F}(\mathbf{A}, \mathbf{B})) = \mathbf{A}^\top \frac{\partial l}{\partial \mathbf{F}}$$

```

In [4]: def matmul_vjp(dldF, A, B):
        G = dldF
        if G.ndim == 0:
            # Case 1: vector-vector multiplication
            assert A.ndim == 1 and B.ndim == 1
            dldA = G*B
            dldB = G*A
            return (unbroadcast(A, dldA),
                    unbroadcast(B, dldB))

        assert not (A.ndim == 1 and B.ndim == 1)

        # 1. If both arguments are 2-D they are multiplied like conventional mat
        # 2. If either argument is N-D, N > 2, it is treated as a stack of matrices
        # residing in the last two indexes and broadcast accordingly.
        if A.ndim >= 2 and B.ndim >= 2:
            dldA = G @ B.swapaxes(-2, -1)
            dldB = A.swapaxes(-2, -1) @ G
        if A.ndim == 1:
            # 3. If the first argument is 1-D, it is promoted to a matrix by prepending
            # a 1 to its dimensions. After matrix multiplication the prepended
            A_ = A[np.newaxis, :]
            G_ = G[np.newaxis, :]
            dldA = G @ B.swapaxes(-2, -1)
            dldB = A_.swapaxes(-2, -1) @ G_ # outer product
        elif B.ndim == 1:
            # 4. If the second argument is 1-D, it is promoted to a matrix by appending
            # a 1 to its dimensions. After matrix multiplication the appended
            B_ = B[:, np.newaxis]
            G_ = G[:, np.newaxis]
            dldA = G_ @ B_.swapaxes(-2, -1) # outer product
            dldB = A.swapaxes(-2, -1) @ G
        return (unbroadcast(A, dldA),
                unbroadcast(B, dldB))

matmul = Op(
    apply=np.matmul,
    vjp=matmul_vjp,
    name='@',
    nargs=2)

```

```

In [5]: def exp_vjp(dldf, x):
        dldx = dldf * np.exp(x)
        return (unbroadcast(x, dldx),)

exp = Op(
    apply=np.exp,
    vjp=exp_vjp,
    name='exp',
    nargs=1)

```

```

In [6]: def log_vjp(dldf, x):
        dldx = dldf / x
        return (unbroadcast(x, dldx),)

log = Op(

```

```

    apply=np.log,
    vjp=log_vjp,
    name='log',
    nargs=1)

```

```

In [7]: def sum_vjp(dldf, x, axis=None, **kwargs):
        if axis is not None:
            dldx = np.expand_dims(dldf, axis=axis) * np.ones_like(x)
        else:
            dldx = dldf * np.ones_like(x)
        return (unbroadcast(x, dldx),)

sum_ = Op(
    apply=np.sum,
    vjp=sum_vjp,
    name='sum',
    nargs=1)

```

```

In [18]: def maximum_vjp(dldf, a, b):
        dlda = dldf * np.where(a > b, 1, 0)
        dlbd = dldf * np.where(a > b, 0, 1)
        return unbroadcast(a, dlda), unbroadcast(b, dlbd)

maximum = Op(
    apply=np.maximum,
    vjp=maximum_vjp,
    name='maximum',
    nargs=2)

```

```

In [19]: NoOp = Op(apply=None, name='', vjp=None, nargs=0)
class Tensor:
    __array_priority__ = 100
    def __init__(self, value, grad=None, parents=(), op=NoOp, kwargs={}, requires_grad=True):
        self.value = np.asarray(value)
        self.grad = grad
        self.parents = parents
        self.op = op
        self.kwargs = kwargs
        self.requires_grad = requires_grad

    shape = property(lambda self: self.value.shape)
    ndim = property(lambda self: self.value.ndim)
    size = property(lambda self: self.value.size)
    dtype = property(lambda self: self.value.dtype)

    def __add__(self, other):
        cls = type(self)
        other = other if isinstance(other, cls) else cls(other)
        return cls(add.apply(self.value, other.value),
                    parents=(self, other),
                    op=add)
    __radd__ = __add__

    def __mul__(self, other):
        cls = type(self)

```



```

        other = other if isinstance(other, cls) else cls(other)
        return cls(mul.apply(self.value, other.value),
                    parents=(self, other),
                    op=mul)
__rmul__ = __mul__

def __matmul__(self, other):
    cls = type(self)
    other = other if isinstance(other, cls) else cls(other)
    return cls(matmul.apply(self.value, other.value),
                parents=(self, other),
                op=matmul)

def exp(self):
    cls = type(self)
    return cls(exp.apply(self.value),
                parents=(self, ),
                op=exp)

def log(self):
    cls = type(self)
    return cls(log.apply(self.value),
                parents=(self, ),
                op=log)

def __pow__(self, other):
    cls = type(self)
    other = other if isinstance(other, cls) else cls(other)
    return (self.log() * other).exp()

def __div__(self, other):
    return self * (other**(-1))

def __sub__(self, other):
    return self + (other * (-1))

def __neg__(self):
    return self*(-1)

def sum(self, axis=None):
    cls = type(self)
    return cls(sum_.apply(self.value, axis=axis),
                parents=(self, ),
                op=sum_,
                kwargs=dict(axis=axis))

def maximum(self, other):
    cls = type(self)
    other = other if isinstance(other, cls) else cls(other)
    return cls(maximum.apply(self.value, other.value),
                parents=(self, other),
                op=maximum)

def __repr__(self):
    cls = type(self)
    return f"{cls.__name__}(value={self.value}, op={self.op.name})" if s

```

```

        #return f"{cls.__name__}(value={self.value}, parents={self.parents},

def backward(self, grad):
    self.grad = grad if self.grad is None else (self.grad+grad)
    if self.requires_grad and self.parents:
        p_vals = [p.value for p in self.parents]
        assert len(p_vals) == self.op.nargs
        p_grads = self.op.vjp(grad, *p_vals, **self.kwargs)
        for p, g in zip(self.parents, p_grads):
            p.backward(g)

```

```
In [20]: Tensor([1, 2]).sum()
```

```
Out[20]: Tensor(value=3, op=sum)
```

```
In [68]: try:
    from graphviz import Digraph
except ImportError as e:
    import subprocess
    subprocess.call("pip install --user graphviz".split())

def trace(root):
    nodes, edges = set(), set()
    def build(v):
        if v not in nodes:
            nodes.add(v)
            for p in v.parents:
                edges.add((p, v))
                build(p)
    build(root)
    return nodes, edges

def draw_dot(root, format='svg', rankdir='LR'):
    """
    format: png | svg | ...
    rankdir: TB (top to bottom graph) | LR (left to right)
    """
    assert rankdir in ['LR', 'TB']
    nodes, edges = trace(root)
    dot = Digraph(format=format, graph_attr={'rankdir': rankdir}) #, node_attr=

    for n in nodes:
        vstr = np.array2string(np.asarray(n.value), precision=4)
        gradstr = np.array2string(np.asarray(n.grad), precision=4)
        dot.node(name=str(id(n)), label = f"{{v={vstr} | g={gradstr}}}", shape='rect')
        if n.parents:
            dot.node(name=str(id(n)) + n.op.name, label=n.op.name)
            dot.edge(str(id(n)) + n.op.name, str(id(n)))

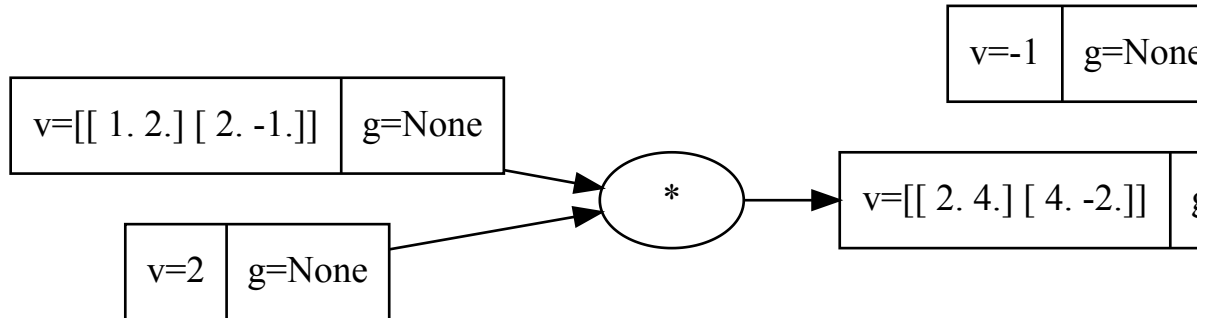
    for n1, n2 in edges:
        dot.edge(str(id(n1)), str(id(n2)) + n2.op.name)

    return dot

```

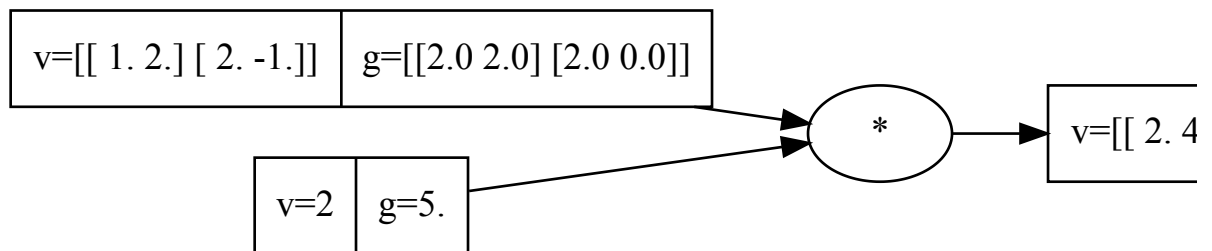
```
In [69]: # a very simple example
x = Tensor([[1.0, 2.0],
            [2.0, -1.0]])
y = (x * 2 - 1).maximum(0).sum(axis=-1)
draw_dot(y)
```

Out[69]:



```
In [70]: y.backward(np.ones_like(y))
draw_dot(y)
```

Out[70]:



```
In [73]: def f_np(x):
b = [1, 0]
return (x @ b)*np.exp((-x*x).sum(axis=-1))

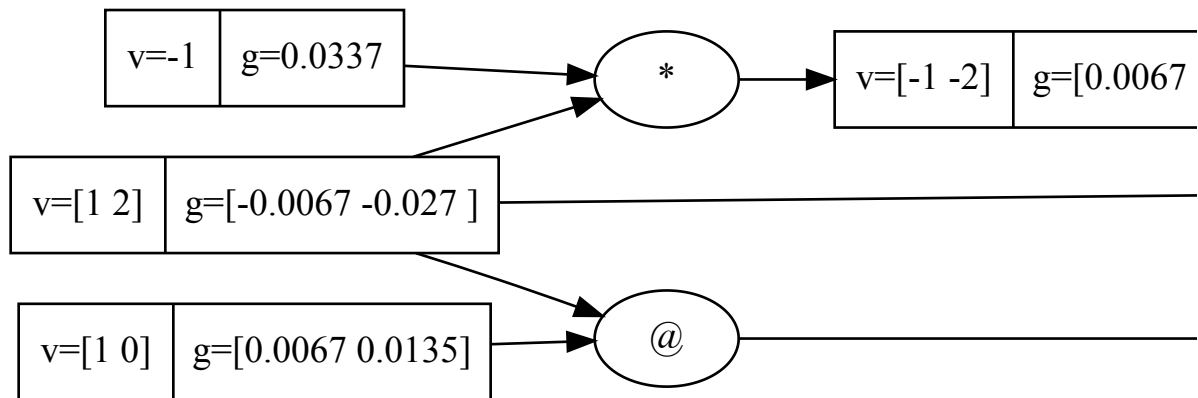
def f_T(x):
b = [1, 0]
return (x @ b)*(-x*x).sum(axis=-1).exp()

def grad_f(x):
xT = Tensor(x)
y = f_T(xT)
y.backward(np.ones_like(y.value))
return xT.grad
```

```
In [74]: xT = Tensor([1, 2])
out = f_T(xT)
out.backward(1)
print(xT.grad)
draw_dot(out)
```

[-0.00673795 -0.02695179]

Out[74]:



```

In [57]: def numerical_jacobian(f, x, h=1e-10):
    n = x.shape[-1]
    eye = np.eye(n)
    x_plus_dx = x + h * eye # n x n
    num_jac = (f(x_plus_dx) - f(x)) / h # limit definition of the formula #
    if num_jac.ndim >= 2:
        num_jac = num_jac.swapaxes(-1, -2) # m x n
    return num_jac

# Compare our grad_f with numerical gradient
def check_numerical_jacobian(f, jac_f, nD=2, **kwargs):
    x = np.random.rand(nD)
    print(x)
    num_jac = numerical_jacobian(f, x, **kwargs)
    print(num_jac)
    print(jac_f(x))
    return np.allclose(num_jac, jac_f(x), atol=1e-06, rtol=1e-4) # m x n

## Throw error if grad_f is wrong
assert check_numerical_jacobian(f_np, grad_f)

[0.4717993  0.90549333]
[ 0.19560853 -0.30124125]
[ 0.19560835 -0.30124165]

```

In []:

In []: