Automatic differentiation

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_vjps.p
4. https://auto-ed.readthedocs.io/en/latest/mod2.html#ii-more-theory
5. https://github.com/lindseysbrown/Auto-eD/blob/master/docs/mod2.rst?plain=1

Latex macros

Chain rule

Scalar single-variable chain rule

Recall the limit definition of derivative of a function,

$$g'(x) = \lim_{h \to 0} \frac{g(x+h) - g(x)}{h}.$$

From the limit definition you can find the value of $g(x+h)$ as

$$\lim_{h \to 0} g(x+h) = \lim_{h \to 0} g(x) + g'(x)h.$$

You can use this rule to find the chain rule of finding the chaining of two functions together,

$$\begin{aligned}
\frac{\partial f(g(x))}{\partial x} &= \lim_{h \to 0} \frac{f(g(x+h)) - f(g(x))}{h} \\
&= \lim_{h \to 0} \frac{f(g(x) + g'(x)h) - f(g(x))}{h} \\
&= \lim_{h \to 0} \frac{f(g(x)) + f'(g(x))g'(x)h - f(g(x))}{h} \\
&= f'(g(x))g'(x)
\end{aligned}$$

Scalar two-variable chain rule

Consider a function of two variables $f(u(x), v(x))$. Find its derivative,

$$\begin{aligned}\frac{\partial f(u(x), v(x))}{\partial x} &= \lim_{h \to 0} \frac{f(u(x+h), v(x+h)) - f(u(x), v(x))}{h} \\ &= \lim_{h \to 0} \frac{f(u(x) + u'(x)h, v(x) + v'(x)h) - f(u(x), v(x))}{h}\end{aligned}$$

Now $f(u + \delta u, v + \delta v)$ should not be expanded in one step but in two steps. First keep $v + \delta v$ as it is, and expand with respect to $u + \delta u$

$$\lim_{\delta v, \delta u \to 0} f(u + \delta u, v + \delta v) = \lim_{\delta v, \delta u \to 0} f(u, v + \delta v) + f'_u(u, v + \delta v)\delta u,$$

and then do the same with $v + \delta v$,

$$\lim_{\delta v, \delta u \to 0} f(u + \delta u, v + \delta v) = \lim_{\delta v, \delta u \to 0} f(u, v) + f'_v(u, v)\delta v + f'_u(u, v + \delta v)\delta u,$$

We use $\lim_{\delta v \to 0} f'_u(u, v + \delta v) = \lim_{\delta v \to 0} f'_u(u, v)$ to get,

$$\lim_{\delta v, \delta u \to 0} f(u + \delta u, v + \delta v) = \lim_{\delta v, \delta u \to 0} f(u, v) + f'_v(u, v)\delta v + f'_u(u, v)\delta u.$$

Going back to the chain rule,

$$\begin{aligned}\frac{\partial f(u(x), v(x))}{\partial x} &= \lim_{h \to 0} \frac{f(u(x) + u'(x)h, v(x) + v'(x)h) - f(u(x), v(x))}{h} \\ &= \lim_{h \to 0} \frac{f(u(x), v(x)) + f'_v(u(x), v(x))v'(x)h + f'_u(u(x), v(x))u'(x)h - f(u(x), v(x))}{h} \\ &= \lim_{h \to 0} \frac{f'_v(u(x), v(x))v'(x)h + f'_u(u(x), v(x))u'(x)h}{h} \\ &= f'_v(u(x), v(x))v'(x) + f'_u(u(x), v(x))u'(x)\end{aligned}$$

Scalar valued vector function chain rule

Consider two functions $f(\mathbf{g}) : \mathrm{R}^m \to \mathrm{R}$, $\mathbf{g}(x) : \mathrm{R} \to \mathrm{R}^m$ that can be composed together $f(\mathbf{g}(x))$. We want to find the derivative of composition $f \circ g$ by chain rule.

Recall that the derivative (Jacobian) of $f(\mathbf{y})$ is a row vector,

$$\frac{\partial f(\mathbf{g})}{\partial \mathbf{g}} = \begin{bmatrix} \frac{\partial f}{\partial g_1} & \frac{\partial f}{\partial g_2} & \cdots & \frac{\partial f}{\partial g_m} \end{bmatrix}.$$

And the derivative (Jacobian) of $\mathbf{g}(x)$ is a column vector,

$$\frac{\partial \mathbf{g}(x)}{\partial x} = \begin{bmatrix} \frac{\partial g_1}{\partial x} \\ \frac{\partial g_2}{\partial x} \\ \vdots \\ \frac{\partial g_m}{\partial x} \end{bmatrix}.$$

Note that a vector function is a multi-variate scalar function

$$f(\mathbf{g}(x)) = f(g_1(x), g_2(x), \dots, g_m(x)).$$

We can apply the multi-variate scalar function chain rule,

$$\begin{aligned} \frac{\partial}{\partial x} f(\mathbf{g}(x)) &= f'_{g_1}(g_1(x), \dots, g_m(x))g'_1(x) + \cdots + f'_{g_m}(g_1(x), \dots, g_m(x))g'_m(x) \\ &= f'_{g_1}(\mathbf{g}(x))g'_1(x) + \cdots + f'_{g_m}(\mathbf{g}(x))g'_m(x). \end{aligned}$$

The derivatives of $\mathbf{g}$ can be separated from derivatives of f as vector multiplication,

$$\frac{\partial}{\partial x} f(\mathbf{g}(x)) = \begin{bmatrix} f'_{g_1}(\mathbf{g}(x)) & \cdots & f'_{g_m}(\mathbf{g}(x)) \end{bmatrix} \begin{bmatrix} g'_1(x) \\ \vdots \\ g'_m(x) \end{bmatrix}.$$

Hence the chain rule for vector derivatives works out for our definition of vector derivatives,

$$\frac{\partial}{\partial \mathbf{x}} f(\mathbf{g}(x)) = \frac{\partial f(\mathbf{g}(x))}{\partial \mathbf{g}} \frac{\partial \mathbf{g}(x)}{\partial x}.$$

Note that the order of multiplication matters, specifically

$$\frac{\partial}{\partial \mathbf{x}} f(\mathbf{g}(x)) \neq \frac{\partial \mathbf{g}(x)}{\partial x} \frac{\partial f(\mathbf{g}(x))}{\partial \mathbf{g}}.$$

This is a consequence of row-vector convention. If we chose a column-vector convention the result will be completely different.

General chain rule

Let the function be $\mathbf{f}(\mathbf{g}) : \mathbb{R}^m \to \mathbb{R}^n$ and $\mathbf{g}(\mathbf{x}) : \mathbb{R}^p \to \mathbb{R}^m$, then the derivative (Jacobian) of their composition $\mathbf{f} \circ \mathbf{g}$ is

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{f}(\mathbf{g}(\mathbf{x})) = \frac{\partial \mathbf{f}(\mathbf{g}(\mathbf{x}))}{\partial \mathbf{g}} \frac{\partial \mathbf{g}(\mathbf{x})}{\partial \mathbf{x}}$$

Computational complexity of Forward vs Reverse mode differentiation

Consider three functions, $\mathbf{h}(\mathbf{x}) : \mathbb{R}^m \to \mathbb{R}^n$, $\mathbf{g}(\mathbf{h}) : \mathbb{R}^n \to \mathbb{R}^p$ and $\mathbf{f}(\mathbf{g}) : \mathbb{R}^p \to \mathbb{R}^q$ chained together for composition $\mathbf{f}(\mathbf{g}(\mathbf{h}(\mathbf{x}))) : \mathbb{R}^m \to \mathbb{R}^q$. To find the derivative (Jacobian) the composite function, we use chain rule:

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{f}(\mathbf{g}(\mathbf{h}(\mathbf{x}))) = \frac{\partial \mathbf{f}}{\partial \mathbf{g}} \frac{\partial \mathbf{g}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{x}}$$

Computational complexity of matrix multiplication

Let's say you multiply two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$, total number of additions and multiplications (floating point operations) can be calculated by

$$C = AB = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_p \end{bmatrix},$$

where $\mathbf{a}_i^\top$ are the row-vectors of matrix A and $\mathbf{b}_i$ are the column vectors of matrix B. Then matrix C is written as

$$C = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b}_1 & \mathbf{a}_1^\top \mathbf{b}_2 & \cdots & \mathbf{a}_1^\top \mathbf{b}_p \\ \mathbf{a}_2^\top \mathbf{b}_1 & \mathbf{a}_2^\top \mathbf{b}_2 & \cdots & \mathbf{a}_2^\top \mathbf{b}_p \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_m^\top \mathbf{b}_1 & \mathbf{a}_m^\top \mathbf{b}_2 & \cdots & \mathbf{a}_m^\top \mathbf{b}_p \end{bmatrix}$$

We note that C matrix has pm elments and each element requires computing dot product of size n vectors,

$$\mathbf{a}_i^\top \mathbf{b}_j = a_{i1} b_{j1} + a_{i2} b_{j2} + \cdots + a_{in} b_{in}.$$

Each dot product requires n multiplications and $n - 1$ additions. Hence matrix multiplication which has pm dot products requires $pm(n + n - 1)$ (floating point) operations.

Matrix multiplication has a computation complexity of $O(pmn)$ for matrices of size $m \times n$ and $n \times p$.

Computational complexity of forward-mode differentiation

In forward diff, we compute computaional complexity from input side to the output side.

$$\frac{\partial}{\partial \mathbf{x}}\mathbf{f}(\mathbf{g}(\mathbf{h}(\mathbf{x}))) = \left(\frac{\partial \mathbf{f}}{\partial \mathbf{g}}\left(\frac{\partial \mathbf{g}}{\partial \mathbf{h}}\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right)\right)$$

The first two matrix multiplications $X_{p \times n} = \left(\frac{\partial \mathbf{g}}{\partial \mathbf{h}}\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right)$ are of the size $p \times m$ and $m \times n$, resulting in $O(pmn)$ complexity.

The second two matrix multiplications $\left(\frac{\partial \mathbf{f}}{\partial \mathbf{g}}X_{p \times n}\right)$ are of the size $q \times p$ and $p \times n$, resulting in $O(qpn)$ complexity.

The total computational complexity of forward differentiation is $O(qpn + pmn) = O((qp + pm)n)$.

For a longer chain of functions of Jacobians of shape $q_i \times p_i$ with $(p_i = q_{i-1})$.

$$\frac{\partial}{\partial \mathbf{x}}\mathbf{f}_n(\ldots\mathbf{f}_2(\mathbf{f}_1(\mathbf{x}))) = \frac{\partial \mathbf{f}_n}{\partial \mathbf{f}_{n-1}}_{q_n \times p_{n-1}} \cdots \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}_{q_1 \times p_1} \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}}_{q_0 \times p_0}$$

We get a computational complexity that looks like $O((\sum_{i=1}^{n} q_i p_i) p_0)$. Note that the size of input p_0 is the only common factor for the entire chain.

Computational complexity of reverse-mode diff

In reverse-mode diff, we compute computaional complexity from input side to the output side.

$$\frac{\partial}{\partial \mathbf{x}}\mathbf{f}(\mathbf{g}(\mathbf{h}(\mathbf{x}))) = \left(\left(\frac{\partial \mathbf{f}}{\partial \mathbf{g}}\frac{\partial \mathbf{g}}{\partial \mathbf{h}}\right)\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right)$$

The first two matrix multiplications $X_{q \times p} = \left(\frac{\partial \mathbf{f}}{\partial \mathbf{g}}\frac{\partial \mathbf{g}}{\partial \mathbf{h}}\right)$ are of the size $q \times p$ and $p \times m$, resulting in $O(qpm)$ complexity.

The second two matrix multiplications $\left(X_{q \times p} \frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right)$ are of the size $q \times p$ and $p \times n$, resulting in $O(qpn)$ complexity.

The total computational complexity of forward differentiation is $O(qpm + qmn) = O(q(pm + pn))$.

For a longer chain of functions of Jacobians of shape $q_i \times p_i$ with $(p_i = q_{i-1})$.

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{f}_n(\ldots \mathbf{f}_2(\mathbf{f}_1(\mathbf{x}))) = \frac{\partial \mathbf{f}_n}{\partial \mathbf{f}_{n-1}}_{q_n \times p_{n-1}} \cdots \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}_{q_1 \times p_1} \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}}_{q_0 \times p_0}$$

We get a computational complexity that looks like $O(q_n(\sum_{i=0}^{n-1} q_i p_i))$. Note that the size of output q_n is the only common factor for the entire chain.

Reverse-mode differentiation is called backpropagation

Reverse-mode differentiation is called backpropagation in neural networks. It is more popular because most of the times you compute the derivatives of the loss function which is a scalar function with output dimension as only 1. This makes reverse-mode differentiation clearly superior for loss function gradient.

Implementing numpy backpropagation for various operations

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

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

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

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)
    dldb = unbroadcast(b, dldf)
    return dlda, dldb

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

VJP for element-wise multiplication

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

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

Let $l(f(\alpha, \beta)) \in \mathrm{R}$ 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):
    dlda = unbroadcast(a, dldf * b)
    dldb = unbroadcast(b, dldf * a)
    return dlda, dldb

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 findfind $\frac{\partial l}{\partial \mathbf{A}}$ and $\frac{\partial l}{\partial \mathbf{b}}$ for Vector Jacobian product.

Let

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

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

Define matrix derivative of scalar to be:

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

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

.

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 $\mathbf{A}$.

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

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

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

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

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

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

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:

$$\mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_p \end{bmatrix}$$

$$\mathbf{F} = \begin{bmatrix} \mathbf{f}_1 & \mathbf{f}_2 & \cdots & \mathbf{f}_p \end{bmatrix}.$$

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 $l(\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_p)$, then by chain rule of functions with multiple arguments, we have,

$$\frac{\partial}{\partial \mathbf{A}} l(\mathbf{F}(\mathbf{A}, \mathbf{B})) = \frac{\partial}{\partial \mathbf{A}} l(\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_p) = \frac{\partial l}{\partial \mathbf{f}_1} \frac{\partial \mathbf{f}_1}{\partial \mathbf{A}} + \frac{\partial l}{\partial \mathbf{f}_2} \frac{\partial \mathbf{f}_2}{\partial \mathbf{A}} + \cdots + \frac{\partial l}{\partial \mathbf{f}_p} \frac{\partial \mathbf{f}_p}{\partial \mathbf{A}}$$

$$\frac{\partial}{\partial \mathbf{A}} l(\mathbf{F}(\mathbf{A}, \mathbf{B})) = \left(\mathbf{b}_1 \frac{\partial l}{\partial \mathbf{f}_1} \right)^\top + \left(\mathbf{b}_2 \frac{\partial l}{\partial \mathbf{f}_2} \right)^\top + \cdots + \left(\mathbf{b}_p \frac{\partial l}{\partial \mathbf{f}_p} \right)^\top = \left(\mathbf{b}_1 \frac{\partial l}{\partial \mathbf{f}_1} + \mathbf{b}_2 \frac{\partial l}{\partial \mathbf{f}_2} + \cdots + \mathbf{b}_p \frac{\partial l}{\partial \mathbf{f}_p} \right)^\top$$

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

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

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 matri
    # 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
```

```python
    if A.ndim == 1:
        # 3. If the first argument is 1-D, it is promoted to a matrix by pre
        #    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 ap
        #    a 1 to its dimensions. After matrix multiplication the appendec
        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]:
```python
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]:
```python
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]:
```python
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]:
```python
def maximum_vjp(dldf, a, b):
    dlda = dldf * np.where(a > b, 1, 0)
    dldb = dldf * np.where(a > b, 0, 1)
    return unbroadcast(a, dlda), unbroadcast(b, dldb)

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

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

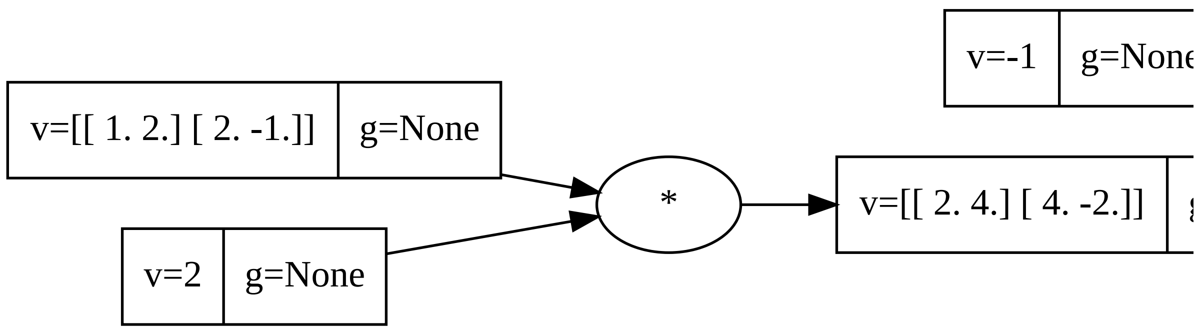
In [68]:
```
try:
    from graphviz import Digraph
```

```python
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_at

    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}}}", sha
        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]:

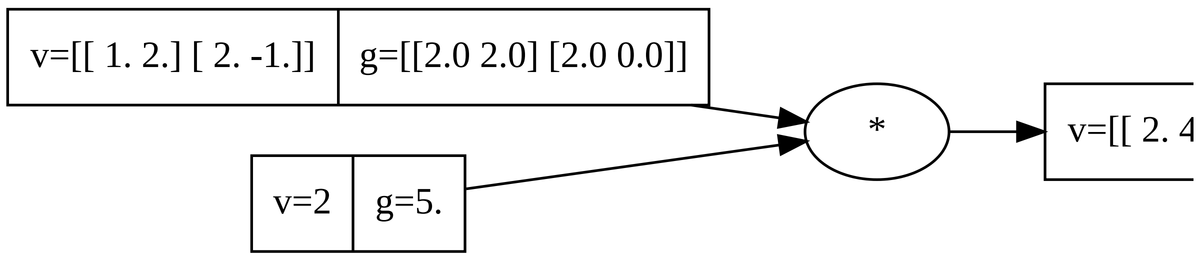
```python
# a very simple example
x = Tensor([[1.0, 2.0],
            [2.0, -1.0]])
y = (x * 2 - 1).maximum(0).sum(axis=-1)
draw_dot(y)
```

Out[69]:



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

Out[70]:

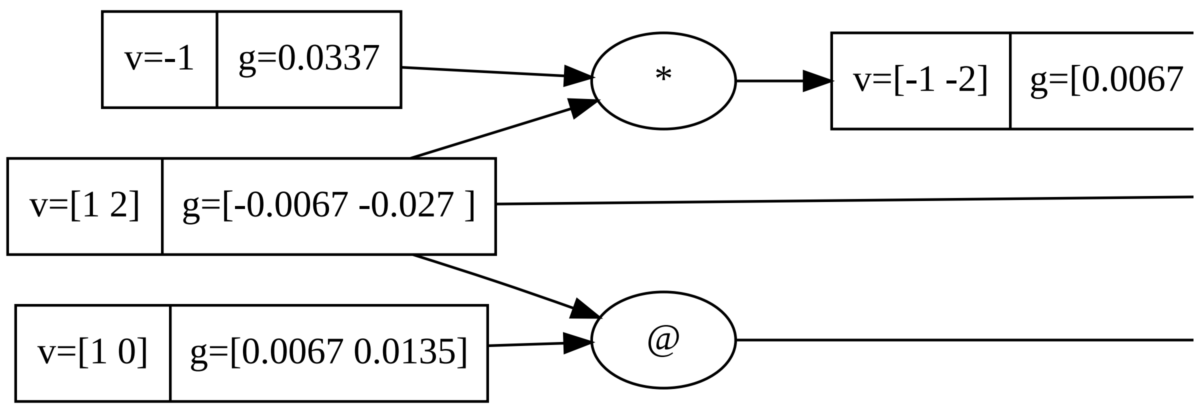


In [73]:
```
def f_np(x):
    b = [1, 0]
    return (x @ b)*np.exp((-x*x).sum(axis=-1))

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 [ ]: