Topics

1. Pytorch basics: https://colab.research.google.com/github/wecacuee/ECE490-Neural-Networks/blob/master//notebooks/06-pytorch/NumpyTutorial-Pytorched.ipynb
2. Autograd Mathematics: https://colab.research.google.com/github/wecacuee/ECE490-Neural-Networks/blob/master/notebooks/03-autograd/AutogradNumpy.ipynb
3. Probability problems (below)

Probability definitions

Q1: Define Sample Space

Sample space is the set all possible of outcomes of an experiment, denoted by Ω.

For example, For 2-coin tosses the sample space is

$$\Omega_{\text{2-coin}} = \{HH, HT, TH, TT\}$$

For roll of a dice with 6-sides

$$\Omega_{\text{dice}} = \{1, 2, 3, 4, 5, 6\}$$

For weight measurements of an individual, the sample space is the set of all positive real numbers

$$\Omega_{\text{weight}} = \mathbb{R}^+$$

Q2: Define Event Space

An event is the set of outcomes that we might be interested in.

Event space is a set of subsets of the sample space.

or example, For 2-coin tosses the set of all subsets of the sample space in cluding the null set $\{\}$ and the full sample Ω

$$\mathcal{F}_{\text{2-coin}} = \{\{\}, \{HH\}\{HT\}, \{TH\}, \{TT\}, \{HH, HT\}, \dots, \underbrace{\{HH, HT, TH, TT\}}_{\Omega}\}$$

For weight measurements of an individual, the event space is be the set of all unions and intersections of intervals (open and closed) of sample space (positive real numbers).

$$\mathcal{F}_{\text{weight}} = \{\cup_i \cap_j [a_{ij}, b_{ij}] : a_{ij} < b_{ij}, a_{ij} \in \mathbb{R}, b_{ij} \in \mathbb{R}\}$$

Q3: Define Power set

The set of all possible subsets of a set Ω is called a power set and is denoted by 2^Ω.

For roll of a dice with 6-sides

$$2^\Omega = \{\{\}, \{HH\}\{HT\}, \{TH\}, \{TT\}, \{HH, HT\}, \ldots, \underbrace{\{HH, HT, TH, TT\}}_{\Omega}\}$$

For discrete sample space, event space is the power set of the sample space.

Q4: Define Probability measure

Probability measure is a function $P : \mathcal{F} \to [0, 1]$ that maps from event space to real numbers between $[0, 1]$ and satisfy the following Kolmogorov axioms

1. $P(E) \in [0, 1]$ for all $E \in \mathcal{F}$, where $\mathcal{F}$ is event space
2. $P(\Omega) = 1$, where Ω is sample space
3. For all disjoint set of events A_1, A_2 ($A_1 \cap A_2 = \phi$), the probability of union of events is the sum of individual event probabilities:

   $$P(A_1) + P(A_2) = P(A_1 \cup A_2)$$

   when $A_1 \cap A_2 = \phi$.

   In general, for a countably infinite set of event $A_1, A_2, \ldots A_n \ldots \infty$,

   $$P\left(\bigcup_{n=1}^{\infty} A_n\right) = \sum_{n=1}^{\infty} P(A_n)$$

   when $A_i \cap A_j = \infty$ for all $i \neq j$.

Q5: Define Probability space

The triple of sample space Ω, event space $\mathcal{F}$ and a probability measure $P : \mathcal{F} \to [0, 1]$ is called a probability space.

Q6: Define Random variable

A random variable is a function $X : \Omega \to \mathbb{Q}$ that maps from sample space Ω to a space of integers $\mathbb{Z}$ or real numbers $\mathbb{R}$ (in general a measurable space), such that a preimage $X^{-1}(B) \in \Omega$ of any set of numbers $B \in \mathbb{Q}$ exists in the sample space.

For example, a 2-coin toss:

$$\Omega = \{HH, HT, TH, TT\}$$

A random variable maps the elements of sample space to a number,

$$X(HH) = 0, X(HT) = 1, X(TH) = 2, X(TT) = 3$$

By slight abuse of notation, the random variable also maps events to a set of numbers $X : \mathcal{F} \to B$,

$$X(\{HT, TH, TT\}) = \{1, 2, 3\}$$

Q7: What is the difference between discrete and continuous random variable

Discrete random variable: When the random variable maps the sample space to integers, then the random variable is discrete.

Continuous random variable: When the random variable maps the sample space to real numbers then the random variable is continuous.

Q8: Define Probability mass function (PMF)

For a discrete random variable (RV) the Probability mass function (PMF) is a function that assigns probability value to every discrete value of the random variable, such that

$$\sum_{x \in \Omega} P(X = x) = 1.$$

For example, a die roll

$$\Omega = \{1, \ldots, 6\}$$

$$P(X = 1) = 1/6, P(X = 2) = 1/6, \ldots, P(X = 6) = 1/6$$

PMF is denoted as multiple symbols $P(X = x) = P_X(x) = P(x)$

Q9: Define probability density function (PDF)

For a continuous random variable $X : \Omega \to \mathbb{R}$, the probability density function (PDF) is a function $f_X : \mathbb{R} \to [0, \infty)$ such that:

1. $f_X(x) \geq 0$ for all $x \in \mathbb{R}$
2. $\int_{\mathbb{R}} f_X(x)dx = 1$
3. $P(a \leq X \leq b) = P(X \in [a, b]) = \int_a^b f_X(x)dx$

Q10: Define joint probability mass function

$$P(X=x,Y=y) = P((X=x)\cap(Y=y)) = P((X=x) \text{ AND } (Y=y))$$

Q11: Define joint probability density function

For two continuous random variable X and Y, the joint probability density function (PDF) is a function $f_{X,Y} : (\mathbb{R},\mathbb{R}) \to [0,\infty)$ such that:

1. $f_{X,Y}(x,y) \geq 0$ for all $x, y \in \mathbb{R}$
2. $\int_{\mathbb{R}}\int_{\mathbb{R}} f_{X,Y}(x,y)dxdy = 1$
3. $P(a \leq X \leq b, c \leq Y \leq d) = P(X \in [a,b], Y \in [c,d]) = \int_c^d \int_a^b f_{X,Y}(x,y)dxdy$

Q12: Define cumulative distribution function

A cumulative distribution function (CDF) is $F_X(x)$ is defined as

$$F_X(x) = P(X \leq x).$$

For a discrete random variable, CDF is the sum of probability mass function

$$F_X(x) = P(X \leq x) = \sum_{a \leq x} P_X(a)$$

For a continuous random variable, CDF is the integral of probability density function

$$F_X(x) = P(X \leq x) = \int_{-\infty}^{x} f_X(z)dz$$

Q13: Define conditional probability

Conditional probability of event A given event B is defined as

$$P(A|B) = \frac{P(A,B)}{P(B)}$$

when $P(B) \neq 0$.

Q14: State Bayes theorem

For any two events, A and B

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Q15: State Bayes theorem in terms of likelihood, prior, evidence and posterior

For an observable event D and a hidden event θ, the posterior $P(\theta|D)$ can be estimated using Bayes theorem in terms of likelihood $P(D|\theta)$, prior $P(\theta)$ and evidence $P(D)$ as

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$$

Q16: Define statistical independence

Two random variables X and Y are said to be independent, denoted as $X \perp Y$ if any of the following equivalent condition hold for all x, y :

1. $$P(X = x, Y = y) = P(X = x)P(Y = y)$$
2. $$P(X = x|Y = y) = P(X = x)$$
3. $$P(Y = y|X = x) = P(Y = y)$$

Q17: Define conditional independence

Two random variables X and Y are said to be conditionally independent given random variable Z, denoted as $X \perp Y|Z$ if for all x, y, z :

$$P(X = x, Y = y|Z = z) = P(X = x|Z = z)P(Y = y|Z = z)$$

Q18: Identically independently distributed (IID)

The random variables (RVs) $X_1, X_2, \ldots, X_n$ are identically independently distributed if they are mutually independent $X_i \perp X_j$ and have the same probability distributions $P_{X_i}(x_i) = P_{X_j}(x_j)$.

Q19: Expectation of a function of a random variable

The expectation of a function $g(X)$ of a discrete random variable X is defined as:

$$\mathbb{E}_X[g(X)] = \sum_{x \in \mathbb{Z}} P(X = x)g(x)$$

The expectation of a function $g(X)$ of a continuous random variable X is defined as:

$$\mathbb{E}_X[g(X)] = \int_{x \in \mathbb{R}} f_X(x)g(x)dx$$

Q20: What is the difference between sample mean and expectation

Sample mean of n samples is

$$\mu(X_1, \dots, X_n) = \frac{1}{n}\sum_{i=1}^{n} X_i$$

Expectation of a discrete random variable is

$$\mathbb{E}_X[X] = \sum_{x\in\Omega_X} P(X = x)x$$

Sample mean converges to the expectation when n with high probability:

$$\lim_{n\to\infty} \mu(X_1, \dots, X_n) = E_X[X]$$

Q21: Define variance of a function of a random variable

The expectation of a function $g(X)$ of a random variable X is given by

$$\mathbb{V}_X[g(X)] = \mathbb{E}_X\left[(g(X) - \mathbb{E}_X[g(X)])^2\right]$$

Q22: Define a covariance matrix

For random vector $\mathrm{X} = [X_1, X_2, \dots, X_n]$, the covariance matrix of X is defined as:

$$\mathbb{V}_X[\mathrm{X}] = \mathbb{E}_X\left[(\mathrm{X} - \mathbb{E}_X[\mathrm{X}])\,(\mathrm{X} - \mathbb{E}_X[\mathrm{X}])^\top\right]$$

Q23:

Given the dataset $\mathcal{D} = \{(\mathrm{x}_1, y_1), \dots, (\mathrm{x}_n, y_n)\}$, a model $\hat{y}_i = f(\mathrm{x}_i; \theta)$, and a loss function $l(y_i, \hat{y}_i)$, show that the following optimization problem can be interpreted as maximum likelihood estimation. In the process show that for the interpretation, we need the IID (independently, identically distributed) assumption over the dataset. List any other assumptions that you need for the interpretation.

$$\theta^* = \arg\min_{\theta}\sum_{i=1}^{n} l(y_i, f(\mathrm{x}_i; \theta))$$

A23:

Let the x_i and y_i be random vectors for all i. Model the probability distribution as a negative log of the loss function:

$$P((\mathrm{x}_i, y_i)|\theta) = \frac{1}{Z}\exp(-l(y_i, f(\mathrm{x}_i; \theta)).$$

If the samples are IID, then we can write the probability of the entire dataset as products of sample probabilities

$$P(\mathcal{D}|\theta) = \prod_{i=1}^{n} P((\mathrm{x}_i, y_i)|\theta)$$

$$P(\mathcal{D}|\theta) = \prod_{i=1}^{n} \frac{1}{Z}\exp(-l(y_i, f(\mathrm{x}_i; \theta)).$$

A product of exponents is the summation of their powers,

$$P(\mathcal{D}|\theta) = \frac{1}{Z}\exp(-\sum_{i=1}^{n} l(y_i, f(\mathrm{x}_i; \theta)).$$

Denote

$$L(\mathcal{D}; \theta) = \sum_{i=1}^{n} l(y_i, f(\mathrm{x}_i; \theta).$$

The original optimization problem can be written as:

$$\theta^* = \arg\ \min_{\theta} L(\mathcal{D}; \theta)$$

Taking negative exponent on both sides turns the problem into a maximization problem because $\exp(-y)$ is a monotonically decreasing function.

$$\theta^* = \arg\ \max_{\theta} \exp(-L(\mathcal{D}; \theta))$$

This problem is the same as maximizing the likelihood $P(\mathcal{D}|\theta)$, hence maximum likelihood estimate.

Q24:

Given the dataset $\mathcal{D} = \{(\mathrm{x}_1, y_1), \dots, (\mathrm{x}_n, y_n)\}$, a model $\hat{y}_i = f(\mathrm{x}_i; \theta)$, a regularizer $R(\theta)$ and a loss function $l(y_i, \hat{y}_i)$, show that the following optimization problem can be interpreted as maximum-a-posteriori estimation. In the process show that for the interpretation, we need the IID (independently, identically distributed) assumption over the dataset. List any other assumptions that you need for the interpretation.

$$\theta^* = \arg\ \min_{\theta} \sum_{i=1}^{n} l(y_i, f(\mathrm{x}_i; \theta)) + \lambda R(\theta),$$

where λ is some positive constant that balances between the loss function and the regularizer.

A24:

Let the x_i and y_i be random vectors for all i. Model the probability distribution as a negative log of the loss function:

$$P((\mathrm{x}_i, y_i)|\theta) = \frac{1}{Z}\exp(-l(y_i, f(\mathrm{x}_i; \theta)).$$

If the samples are IID, then we can write the probability of the entire dataset as products of sample probabilities

$$P(\mathcal{D}|\theta) = \prod_{i=1}^{n} P((\mathrm{x}_i, y_i)|\theta)$$

$$P(\mathcal{D}|\theta) = \prod_{i=1}^{n} \frac{1}{Z}\exp(-l(y_i, f(\mathrm{x}_i; \theta)).$$

A product of exponents is the summation of their powers,

$$P(\mathcal{D}|\theta) = \frac{1}{Z}\exp(-\sum_{i=1}^{n} l(y_i, f(\mathrm{x}_i; \theta)).$$

Denote

$$L(\mathcal{D}; \theta) = \sum_{i=1}^{n} l(y_i, f(\mathrm{x}_i; \theta).$$

The original optimization problem can be written as:

$$\theta^* = \arg\ \min_\theta L(\mathcal{D}; \theta) + \lambda R(\theta)$$

Taking negative exponent on both sides turns the problem into a maximization problem because $\exp(-y)$ is a monotonically decreasing function.

$$\theta^* = \arg\ \max_\theta \exp(-L(\mathcal{D}; \theta)) \exp(-\lambda R(\theta))$$

The first term is the same as maximizing the likelihood $P(\mathcal{D}|\theta)$. If we interpret the second term as a prior:

$$P(\theta) = \frac{1}{Z'}\exp(-\lambda R(\theta)),$$

then we can rewrite the original optimization problem as

$$\theta^* = \arg\ \max_\theta P(\mathcal{D}|\theta)P(\theta)$$

By Bayes theorem $P(\mathcal{D}|\theta)P(\theta) = P(\theta|\mathcal{D})P(\mathcal{D})$, hence we can write the optimization problem as maximizing the posterior

$$\theta^* = \arg\ \max_\theta P(\theta|\mathcal{D})P(\mathcal{D}).$$

We can ignore the evidence term $P(\mathcal{D})$, because it is independent of θ the optimization variable. The original problem reduces to maximizing the posterior, hence maximum a posteriori:

$$\theta^* = \arg \max_{\theta} P(\theta|\mathcal{D})$$

Q25: Define L-p norm for $p = \{1, 2, \ldots\}$

$$\|\mathrm{x}\|_p = \left(|x_1|^p + |x_2|^p + \cdots + |x_n|^p\right)^{\frac{1}{p}}$$

Q26: Find the minimum point for the following regularized least square problem and

$$\mathrm{w}^* = \arg \min_{\mathrm{w}} \|\mathrm{y} - \mathrm{X}\mathrm{w}\|^2 + \lambda\|\mathrm{w}\|^2,$$

where $\mathrm{w} \in \mathbb{R}^n$, $\mathrm{y} \in \mathbb{R}^m$, $\mathrm{X} \in \mathbb{R}^{m\times n}$ and $\lambda \in \mathbb{R}^+$

A26:

Let $f(\mathrm{w}) = \|\mathrm{y} - \mathrm{X}\mathrm{w}\|^2 + \lambda\|\mathrm{w}\|^2$

Write $f(\mathrm{w})$ in terms of inner product,

$$f(\mathrm{w}) = (\mathrm{y} - \mathrm{X}\mathrm{w})^\top(\mathrm{y} - \mathrm{X}\mathrm{w}) + \lambda\mathrm{w}^\top\mathrm{w}$$

Expand and collect the terms,

$$f(\mathrm{w}) = \mathrm{w}^\top(\mathrm{X}^\top\mathrm{X} + \lambda I_n)\mathrm{w} - 2\mathrm{y}^\top\mathrm{X}\mathrm{w} + \mathrm{y}^\top\mathrm{y}$$

Taking the derivative of $f(\mathrm{w})$ we get,

$$\frac{\partial}{\partial \mathrm{w}} f(\mathrm{w}) = 2\mathrm{w}^\top(\mathrm{X}^\top\mathrm{X} + \lambda I_n) - 2\mathrm{y}^\top\mathrm{X}.$$

At the maximum point w^* the derivative of $f(\mathrm{w})$ is zero,

$$\frac{\partial}{\partial \mathrm{w}} f(\mathrm{w})\bigg|_{\mathrm{w}^*} = 0_n^\top,$$

Equating the derivative to zero at w^*, we can solve for w^*,

$$2\mathrm{w}^{*\top}(\mathrm{X}^\top\mathrm{X} + \lambda I_n) - 2\mathrm{y}^\top\mathrm{X} = 0_n^\top.$$

Rearranging we get,

$$\mathrm{w}^* = (\mathrm{X}^\top\mathrm{X} + \lambda I_n)^{-1}\mathrm{X}^\top\mathrm{y}$$

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

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

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

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

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

Similarly,

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

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

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):
    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

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

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

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

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

Similarly,

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

Case 2: VJP for matrix-vector multiplication

Let

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

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

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

Let

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

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

Define matrix derivative of scalar to be:

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

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

.

Note that

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

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

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

Let

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

Then

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

Returning to our original quest for

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

Note that

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

We can group the terms inside a single transpose.

Which results in

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

The derivative with respect to b is simpler:

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

Case 3: VJP for matrix-matrix multiplication

Let

$$\mathrm{F}(\mathrm{A}, \mathrm{B}) = \mathrm{AB}$$

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

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

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

$$\mathrm{B} = \begin{bmatrix} \mathrm{b}_1 & \mathrm{b}_2 & \dots & \mathrm{b}_p \end{bmatrix}$$
$$\mathrm{F} = \begin{bmatrix} \mathrm{f}_1 & \mathrm{f}_2 & \dots & \mathrm{f}_p \end{bmatrix}.$$

Then for all i

$$\mathrm{f}_i = \mathrm{Ab}_i$$

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

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

and for all $i \neq j$

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

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

$$\frac{\partial}{\partial \mathrm{A}} l(\mathrm{F}(\mathrm{A}, \mathrm{B})) = \frac{\partial}{\partial \mathrm{A}} l(\mathrm{f}_1, \mathrm{f}_2, \dots, \mathrm{f}_p) = \frac{\partial l}{\partial \mathrm{f}_1} \frac{\partial \mathrm{f}_1}{\partial \mathrm{A}} + \frac{\partial l}{\partial \mathrm{f}_2} \frac{\partial \mathrm{f}_2}{\partial \mathrm{A}} + \dots + \frac{\partial l}{\partial \mathrm{f}_p} \frac{\partial \mathrm{f}_p}{\partial \mathrm{A}}$$

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

It turns out that some of outer products can be compactly written as matrix-matrix multiplication: $$ \bfb_1\frac{\p l}{\p \bff_1}

- \bfb_2\frac{\p l}{\p \bff_2}
- \dots
- \bfb_p\frac{\p l}{\p \bff_p} =

$$\begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \dots & \mathbf{b}_p \end{bmatrix}$$

$$\begin{bmatrix} \frac{\partial l}{\partial \mathbf{f}_1} \\ \frac{\partial l}{\partial \mathbf{f}_2} \\ \vdots \\ \frac{\partial l}{\partial \mathbf{f}_p} \end{bmatrix}$$

= \bfB \left(\frac{\p l}{\p \bfF}\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]:
```python
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
    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 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]:
```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)
```

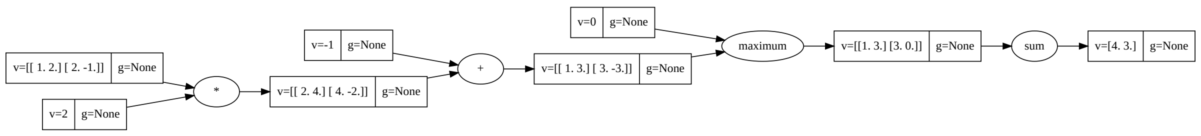
In [20]:
```
Tensor([1, 2]).sum()
```

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

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

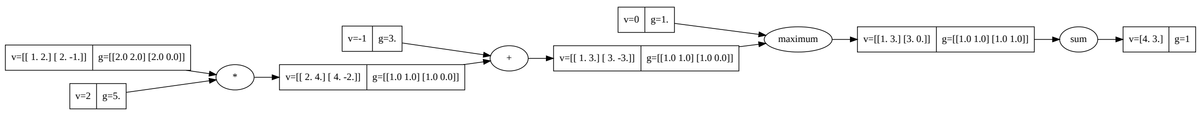
Out[69]:



v=[[1. 2.] [2. -1.]] | g=None
v=2 | g=None
*
v=[[2. 4.] [4. -2.]] | g=None
v=-1 | g=None
+
v=[[1. 3.] [3. -3.]] | g=None
v=0 | g=None
maximum
v=[[1. 3.] [3. 0.]] | g=None
sum
v=[4. 3.] | g=None

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

Out[70]:



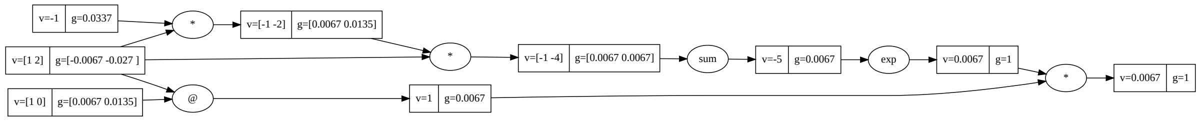
v=[[1. 2.] [2. -1.]] | g=[[2.0 2.0] [2.0 0.0]]
v=2 | g=5.
*
v=[[2. 4.] [4. -2.]] | g=[[1.0 1.0] [1.0 0.0]]
v=-1 | g=3.
+
v=[[1. 3.] [3. -3.]] | g=[[1.0 1.0] [1.0 0.0]]
v=0 | g=1.
maximum
v=[[1. 3.] [3. 0.]] | g=[[1.0 1.0] [1.0 1.0]]
sum
v=[4. 3.] | g=1

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]:



v=-1 | g=0.0337
v=[1 2] | g=[-0.0067 -0.027]
v=[1 0] | g=[0.0067 0.0135]
*
@
v=[-1 -2] | g=[0.0067 0.0135]
*
v=1 | g=0.0067
v=[-1 -4] | g=[0.0067 0.0067]
sum
v=-5 | g=0.0067
exp
v=0.0067 | g=1
*
v=0.0067 | g=1

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 [ ]:

```
```