AI Saturdays Lagos Cohort 7 Practicals

Vectors, Matrices & Linear Algebra

Introduction to numpy
Optimized computational framework (CuPY)

Vectors

A vector is a 1D array of values

We use the notation $x \in \mathbb{R}^n$ to denote that x is an n-dimensional vector with real-valued entries

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

We use the notation x_i to denote the *i*th entry of x

By default, we consider vectors to represent *column* vectors, if we want to consider a row vector, we use the notation x^T

Matrices

A matrix is a 2D array of values

We use the notation $A \in \mathbb{R}^{m \times n}$ to denote a real-valued matrix with m rows and n columns

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{bmatrix}$$

We use A_{ij} to denote the entry in row i and column j

Use the notation $A_{i:}$ to refer to row i, $A_{:j}$ to refer to column j (sometimes we'll use other notation, but we will define before doing so)

Matrices and Linear Algebra

Matrices are:

1. The “obvious” way to store tabular data (particularly numerical entries, though categorical data can be encoded too) in an efficient manner

2. The foundation of linear algebra, how we write down and operate upon (multi-variate) systems of linear equations

Understanding both these perspectives is critical for virtually all data science analysis algorithms

Matrices as tabular data

Given the “Grades” table from our relation data lecture

Person ID	HW1 Grade	HW2 Grade
5	100	80
6	60	80
100	100	100

Natural to represent this data (ignoring primary key) as a matrix

$$A \in \mathbb{R}^{3\times 2} = \begin{bmatrix} 100 & 80 \\ 60 & 80 \\ 100 & 100 \end{bmatrix}$$

Row and Column Ordering

Matrices can be laid out in memory by row or by column

$$A = \begin{bmatrix} 100 & 80 \\ 60 & 80 \\ 100 & 100 \end{bmatrix}$$

Row major ordering: $100, 80, 60, 80, 100, 100$

Column major ordering: $100, 60, 100, 80, 80, 100$

Row major ordering is default for C 2D arrays (and default for Numpy), column major is default for FORTRAN (since a lot of numerical methods are written in FORTRAN, also the standard for most numerical code)

Higher dimensional matrices

From a data storage standpoint, it is easy to generalize 1D vector and 2D matrices to higher dimensional ND storage

"Higher dimensional matrices" are called *tensors*

There is also an extension or linear algebra to tensors, but be aware: most tensor use cases you see are *not* really talking about true tensors in the linear algebra sense

Linear Equation

Matrices and vectors also provide a way to express and analyze systems of linear equations

Consider two linear equations, two unknowns

$$\begin{aligned} 4x_1 &- 5x_2 &&= -13 \\ -2x_1 &+ 3x_2 &&= 9 \end{aligned}$$

We can write this using matrix notation as

$$Ax = b$$

$$A = \begin{bmatrix} 4 & -5 \\ -2 & 3 \end{bmatrix}, \quad b = \begin{bmatrix} -13 \\ 9 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Matrix Operations

For $A, B \in \mathbb{R}^{m \times n}$, matrix addition/subtraction is just the elementwise addition or subtraction of entries

$$C \in \mathbb{R}^{m \times n} = A + B \iff C_{ij} = A_{ij} + B_{ij}$$

For $A \in \mathbb{R}^{m \times n}$, transpose is an operator that "flips" rows and columns

$$C \in \mathbb{R}^{n \times m} = A^T \iff C_{ji} = A_{ij}$$

For $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times p}$ matrix multiplication is defined as

$$C \in \mathbb{R}^{m \times p} = AB \iff C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

- Matrix multiplication is associative ($A(BC) = (AB)C$), distributive ($A(B+C) = AB + AC$), *not commutative* ($AB \neq BA$)

Matrix Inverse

The identity matrix $I \in \mathbb{R}^{n \times n}$ is a square matrix with ones on diagonal and zeros elsewhere, has property that for $A \in \mathbb{R}^{m \times n}$

$$AI = IA = A \text{ (for different sized } I)$$

For a square matrix $A \in \mathbb{R}^{n \times n}$, matrix inverse $A^{-1} \in \mathbb{R}^{n \times n}$ is the matrix such that

$$AA^{-1} = I = A^{-1}A$$

Recall our previous system of linear equations $Ax = b$, solution is easily written using the inverse

$$x = A^{-1}b$$

Miscellaneous definitions

Transpose of matrix multiplication, $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times p}$

$$(AB)^T = B^T A^T$$

Inverse of product, $A \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{n \times n}$ *both square and invertible*

$$(AB)^{-1} = B^{-1} A^{-1}$$

Inner product: for $x, y \in \mathbb{R}^n$, special case of matrix multiplication

$$x^T y \in \mathbb{R} = \sum_{i=1}^{n} x_i y_i$$

Vector norms: for $x \in \mathbb{R}^n$, we use $\|x\|_2$ to denote Euclidean norm

$$\|x\|_2 = (x^T x)^{\frac{1}{2}}$$

Numpy

In Python, the standard library for matrices, vectors, and linear algebra is Numpy

Numpy provides *both* a framework for storing tabular data as multidimensional arrays *and* linear algebra routines

Important note: numpy ndarrays are multi-dimensional arrays, *not* matrices and vectors (there are just routines that support them acting like matrices or vectors)

Importing Numpy

To import an external Python library such as Numpy, use Python's import function. To save yourself some typing later on, you can give the library you import an alias. Here, we are importing Numpy and giving it an alias of np.

```
import numpy as np
```

Creating Numpy Array

The np.array function is used to create an array

```
#creating a 1 dimensional array
x = np.array([1, 2, 3, 4, 5])
y = np.array([9, 10])
```

The shape property is usually used to get the current shape of an array

X.shape

```
# Creating a 2D arrays
z = np.array([[1, 2], [3, 4], [5, 6]])
```

Numpy Functions

Numpy has built-in functions for creating arrays and manipulating. These includes:

```
#arange is Used to create arrays with values in a specified range.
A = np.arange(25)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
```

```
#To change the shape of an array
B = A.reshape(25,1)
C = B.reshape(5,5)
```

Note: Before a reshape function will run successful, the multiplication of the two parameter supply to the function must be equal with multiplication of the shape of the original array you want to reshape.

For example: The shape of variable B is (25, 1) therefore 25 * 1 = 25

The two parameter supply to the reshape function is (5, 5), 5 * 5 = 25

Numpy Functions

```
#zeros is used to create an array filled with zeros.
#ones is used to create an array filled with ones
#random.randn is used to create an array filled with random normal
entries

#1D array
np.zeros (3)
np.ones (3)
np.random.randn (3)
#2D array
np.zeros((2,5))
np.ones ((2,5))
np.random.randn ((2,5))
```

Accessing elements of Numpy Array

To access an element in a two-dimensional array, you need to specify an index for both the row and the column.

```
D = np.array([[5, 7, 8],[3, 5, 9]])
array([[5, 7, 8],
       [3, 5, 9]])
# note that for array numbering in numpy, it starts from zero
#Row 1, column 0 gives a scalar value
D[1,0]
3
#Row 1, column 2
D[1,2]
9
```

Basic Numpy Array Math Operations

```
x = np.array([[1,2,3],[4,5,6]])
y = np.array([[2,2,2],[3,3,3]])
z = np.array([[1,2], [3,4]])

#Transpose a matrix
x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```

```
#Elementwise addittio n
np.add(x,y)
[[3  4  5]
 [7  8  9]]
#Elementwise Subtraction
np.subtract(x,y)
[[ -1   0   1]
 [ 1   2   3]]
[[ -1   0   1]
#Elementwise Multiplication
np.multiply(x,z)
[[ 1   4   9]
 [ 4 10 18]]
[[ 1   4   9]
 [ 4 10 18]]
```

The need for speed (optimization)

Numpy has been a gift to the Python community. It's allowed Data Scientists, Machine Learning Practitioners, and Statisticians to process huge amounts of data in matrix format in a way that's easy and efficient.

Even on its own, Numpy is already a significant step up from Python in terms of speed. Still, even with that speedup ,Numpy is only running on the CPU. With consumer CPUs typically having 8 cores or less, the amount of parallel processing, and therefore the amount of speedup that can be achieved, is limited.

Optimized computational framework

Whenever most data science and machine learning practitioners find their Python code running slow, especially if they see a lot of for-loops, it's always a good idea for them to move the data processing to **Numpy** and let its vectorization do the work at top speed!

However, there is a better alternative, a library that implements Numpy arrays on Nvidia GPUs and with that implementation, superior parallel speedup can be achieved due to the many CUDA cores GPUs have— **CuPY!**

CuPY

CuPy is an open-source library with NumPy syntax that increases speed by doing matrix operations on NVIDIA GPUs. It is accelerated with the CUDA platform from NVIDIA to make full use of the GPU architecture. CuPy's interface is highly compatible with NumPy; in most cases it can be used as a drop-in replacement. CuPy supports various methods, data types, indexing, broadcasting, and more.

Some basic exploration with CuPY in comparison with Numpy

```
import cupy as cp
import numpy as np

#arange is Used to create arrays with values in a specified range.
%time A = cp.arange(25)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
%time A = np.arange(25)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
```

Some basic exploration with CuPY in comparison with Numpy

```
#creating zeros and ones

%time x_np = np.zeros((1000,1000,1000))

%time x_cp = cp.zeros((1000,1000,1000))

%time x_np = np.ones((1000,1000,1000))

%time x_cp = cp.ones((1000,1000,1000))
```

References

- https://towardsdatascience.com/heres-how-to-use-cupy-to-make-numpy-700x-faster-4b920dda1f56
- https://cupy.dev/
- https://us.pycon.org/2018/schedule/presentation/119/
- https://developer.nvidia.com/blog/10-minutes-to-data-science-transitioning-between-rapids-cudf-and-cupy-libraries/
- https://github.com/cupy/cupy