`numpy` Tutorial and Review of Linear Algebra

Content and structure mainly from: http://www.deeplearningbook.org/contents/linear_algebra.html (http://www.deeplearningbook.org/contents/linear_algebra.html)

A lot of data science builds off of the concept of matrices in linear algebra. Matrices are effective ways of representing and manipulating data, and have useful properties when reasoning about data.

The best way to work with matrices and vectors in Python is through the `numpy` library. We will look at `numpy` in this tutorial.

In [47]:
```
import numpy as np
```

Scalars

- Single number
- Denoted as lowercase letter
- Examples
  - $x \in \mathrm{R}$ - Real number
  - $z \in \mathrm{Z}$ - Integer
  - $y \in \{0, 1, \ldots, C\}$ - Finite set
  - $u \in [0, 1]$ - Bounded set

In [48]:
```
x = 1.1343
print(x)
z = int(-5)
print(z)
```

```
1.1343
-5
```

Vectors

- In notation, we usually consider vectors to be "column vectors"
- Denoted as lowercase letter (often bolded)
- Dimension is often denoted by d, D, or p.
- Access elements via subscript, e.g., x_i is the i-th element
- Examples
  - $\mathbf{x} \in \mathrm{R}^d$ - Real vector

- $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$
- $\mathbf{x} = [x_1, x_2, \ldots, x_d]^T$

In Python, we use numpy arrays for vectors (and matrices). These are defined using the `.array` method in `numpy`.

In [49]:
```
x = np.array([1.1343, 6.2345, 35])
print(x)
z = 5 * np.ones(3, dtype=int)
print(z)
```

```
[ 1.1343  6.2345 35.    ]
[5 5 5]
```

Adding vectors in numpy

The operator `+` does different things on numpy arrays vs Python lists:

- For lists, Python concatenates the lists
- For numpy arrays, numpy performs an element-wise addition
- Similarly, for other binary operators such as `-`, `+`, `*`, and `/`

In [50]:
```
a_list = [1, 2]
b_list = [30, 40]
c_list = a_list + b_list
print(c_list)
a = np.array(a_list)  # Create numpy array from Python list
b = np.array(b_list)
c = a + b
print(c)
```

```
[1, 2, 30, 40]
[31 42]
```

We can also see this difference when we try to add a scalar to a vector. If the vector is a list, it doesn't work, but if the vector is a numpy array, then it does.

In [51]:
```
# Adding scalar to list doesn't work
try:
    a_list + 1
except Exception as e:
    print(f'Exception: {e}' )
```

```
Exception: can only concatenate list (not "int") to list
```

In [52]:
```
# Works with numpy arrays
a + 1
```

Out[52]:
```
array([2, 3])
```

Inner product

- **Inner product**, **dot product**, or **vector-vector** multiplication produces *scalar*:

$$\mathbf{x}^T\mathbf{y} = \sum_i x_i y_i$$

- Symmetric

$$\mathbf{x}^T\mathbf{y} = (\mathbf{x}^T\mathbf{y})^T = \mathbf{y}^T\mathbf{x}$$

- Can be executed in numpy via `np.dot`

In [53]:
```
# Inner product
a = np.arange(3)
print(f'a={a}')
b = np.array([11, 22, 33])
print(f'b={b}')
adotb = 0
for i in range(a.shape[0]):
    adotb += a[i] * b[i]
print(f'a^T b = {adotb}')
```

```
a=[0 1 2]
b=[11 22 33]
a^T b = 88
```

In [54]:
```
# The numpy way via np.dot
adotb = np.dot(a, b)
print(f'a^T b = {adotb}')
```

```
a^T b = 88
```

Matrices

- Denoted as uppercase letter
- Access elements by double subscript $X_{i,j}$ or $x_{i,j}$ is the i,j-th entry of the matrix

- Examples
  - $X \in \mathrm{R}^{n \times d}$
  - $X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

In [55]:
```
X = np.arange(12).reshape(3,4)
print(X)
Z = 5 * np.ones((3, 3), dtype=int)
print(Z)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[5 5 5]
 [5 5 5]
 [5 5 5]]
```

Matrix transpose

- Changes columns to rows and rows to columns
- Denoted as A^T
- For vectors $\mathbf{v}$, the transpose changes from a column vector to a row vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}, \qquad \mathbf{x}^T = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}^T = [x_1, x_2, \ldots, x_d]$$

In [56]:
```
A = np.arange(6).reshape(2,3)
print(A)
print(A.T)
```

```
[[0 1 2]
 [3 4 5]]
[[0 3]
 [1 4]
 [2 5]]
```

- NOTE: In numpy, there is only a "vector" (i.e., a 1D array), not really a row or column vector per se, unlike in MATLAB.

In [57]:
```
v = np.arange(5)
print(f'A numpy vector {v} with shape {v.shape}')
print(f'Transpose of numpy vector {v.T} with shape {v.T.shape}')
V = v.reshape(-1, 1)
print(f'A matrix with shape {V.shape}:\n{V}')
print(f'A transposed matrix with shape {V.T.shape}:\n{V.T}')
```

```
A numpy vector [0 1 2 3 4] with shape (5,)
Transpose of numpy vector [0 1 2 3 4] with shape (5,)
A matrix with shape (5, 1):
[[0]
 [1]
 [2]
 [3]
 [4]]
A transposed matrix with shape (1, 5):
[[0 1 2 3 4]]
```

Matrix product

- Let $\mathbf{X}^T \in \mathrm{R}^{m \times n}$, $\mathbf{Y} \in \mathrm{R}^{n \times p}$, then the **matrix product** $\mathbf{Z} = \mathbf{X}^T\mathbf{Y}$ is defined as:

$$\mathbf{Z} = \mathbf{X}^T\mathbf{Y} = \begin{bmatrix} \mathbf{x}_1 \, \mathbf{x}_2 \cdots \mathbf{x}_n \end{bmatrix}^T \begin{bmatrix} \mathbf{y}_1 \, \mathbf{y}_2 \cdots \mathbf{y}_n \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{bmatrix} \begin{bmatrix} \mathbf{y}_1 \, \mathbf{y}_2 \cdots \mathbf{y}_n \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^T\mathbf{y}_1 & \mathbf{x}_1^T\mathbf{y}_2 & \cdots & \mathbf{x}_1^T\mathbf{y}_n \\ \mathbf{x}_2^T\mathbf{y}_1 & \mathbf{x}_2^T\mathbf{y}_2 & \cdots & \mathbf{x}_2^T\mathbf{y}_n \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{x}_n^T\mathbf{y}_1 & \mathbf{x}_n^T\mathbf{y}_2 & \cdots & \mathbf{x}_n^T\mathbf{y}_n \end{bmatrix}$$

- Equivalently this can be written as:

$$z_{i,j} = \sum_{k \in \{1, 2, \ldots, n\}} x_{k,i} y_{k,j}$$

  where $\mathbf{Z} \in \mathrm{R}^{m \times p}$ (notice how inner dimension is collapsed!).

In [58]:
```
# Inner product version
X = np.arange(6).reshape(2, 3)
print(X.T)
Y = np.arange(6).reshape(2, 3)
print(Y)
Z = np.zeros((X.shape[1], Y.shape[1]))
for i in range(Z.shape[0]):
    for j in range(Z.shape[1]):
            Z[i, j] = np.dot(X[:, i], Y[:, j])
print(Z)
```

```
[[0 3]
 [1 4]
 [2 5]]
[[0 1 2]
 [3 4 5]]
[[ 9. 12. 15.]
 [12. 17. 22.]
 [15. 22. 29.]]
```

In [59]:
```
# Triple for loop
X = np.arange(6).reshape(2, 3) * 10
print(f'X with shape {X.shape}\n{X}')
Y = np.arange(6).reshape(2, 3)
print(f'Y with shape {X.shape}\n{X}')
Z = np.zeros((X.shape[1], Y.shape[1]))
for i in range(Z.shape[0]):
    for j in range(Z.shape[1]):
        for k in range(X.shape[0]):
            Z[i, j] += X[k, i] * Y[k, j]
print(f'Z = X^T Y =\n{Z}')
```

```
X with shape (2, 3)
[[ 0 10 20]
 [30 40 50]]
Y with shape (2, 3)
[[ 0 10 20]
 [30 40 50]]
Z = X^T Y =
[[ 90. 120. 150.]
 [120. 170. 220.]
 [150. 220. 290.]]
```

In [60]:
```
# Numpy matrix multiplication
print(np.matmul(X.T, Y))
print(X.T @ Y)
```

```
[[ 90 120 150]
 [120 170 220]
 [150 220 290]]
[[ 90 120 150]
 [120 170 220]
 [150 220 290]]
```

- The naive triple for loop has cubic complexity: $O(n^3)$

- The naive triple for loop has cubic complexity: $O(n^3)$
- `np.matmul` and `@` invoke special linear algebra algorithms in numpy which reduce this to $O(n^{2.803})$
- Takeaway: Use numpy `np.matmul` (or `@`)

Element-wise (Hadamard) product

- Normal matrix mutiplication $C = AB$ is very different from **element-wise** (or more formally **Hadamard**) multiplication, denoted $C = A \odot B$, which in numpy is just the star `*`

In [61]:
```
print(f'X with shape {X.shape}\n{X}')
print(f'Y with shape {Y.shape}\n{Y}')
try:
    Z = X.T * Y  # Fails since matrix shapes don't match and cannot broadca
except ValueError as e:
    print('Operation failed! Message below:')
    print(e)
```

```
X with shape (2, 3)
[[ 0 10 20]
 [30 40 50]]
Y with shape (2, 3)
[[0 1 2]
 [3 4 5]]
Operation failed! Message below:
operands could not be broadcast together with shapes (3,2) (2,3)
```

In [62]:
```
print(f'X with shape {X.shape}\n{X}')
print(f'Y with shape {Y.shape}\n{Y}')
Zelem = X * Y  # Elementwise / Hadamard product of two matrices
print(f'X elementwise product with Y\n{Zelem}')
```

```
X with shape (2, 3)
[[ 0 10 20]
 [30 40 50]]
Y with shape (2, 3)
[[0 1 2]
 [3 4 5]]
X elementwise product with Y
[[  0  10  40]
 [ 90 160 250]]
```

Properties of matrix product

- Distributive: $A(B + C) = AB + AC$
- Associative: $A(BC) = (AB)C$
- **NOT** commutative, i.e., $AB = BA$ does **NOT** always hold
- Transpose of multiplication (**switch order** and transpose of both):

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

In [63]:
```
A = X.T
B = Y
print('AB')
print(np.matmul(A, B))
print('BA')
print(np.matmul(B, A))
print('(AB)^T')
print(np.matmul(A, B).T)
print('B^T A^T')
print(np.matmul(B.T, A.T))
```

```
AB
[[ 90 120 150]
 [120 170 220]
 [150 220 290]]
BA
[[ 50 140]
 [140 500]]
(AB)^T
[[ 90 120 150]
 [120 170 220]
 [150 220 290]]
B^T A^T
[[ 90 120 150]
 [120 170 220]
 [150 220 290]]
```

Identity matrix

- Generalizes the concept of the scalar 1 to a matrix form
- Multiplying by the identity matrix does not change the vector/matrix
- Formally, $I_n \in \mathrm{R}^{n \times n}$, and $\forall \mathbf{X} \in \mathrm{R}^{n \times m}, I_n \mathbf{X} = \mathbf{X}$
- Structure is ones on the diagonal, zero everywhere else:

$$I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

- `np.eye` function to create identity

In [64]:
```
I3 = np.eye(3)
print(I3)
x = np.random.randn(3)
print(x)
print(np.dot(I3, x))
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[-0.43588323  1.27047516 -1.07495702]
[-0.43588323  1.27047516 -1.07495702]
```

- The inverse of *square* matrix $A \in \mathrm{R}^{n \times n}$ is denoted as A^{-1} and defined as:

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

- The "right" inverse is similar and is equal to the left inverse:

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

- Generalizes the concept of inverse x and $\frac{1}{x}$
- Does **NOT** always exist, similar to how the inverse of x only exists if $x \neq 0$

In [65]:
```
A = 100 * np.array([[1, 0.5], [0.2, 1]])
print(A)
Ainv = np.linalg.inv(A)
print(Ainv)
print('A^{-1} A = ')
print(np.dot(Ainv, A))
print('A A^{-1} = ')
print(np.dot(A, Ainv))
```

```
[[100.  50.]
 [ 20. 100.]]
[[ 0.01111111 -0.00555556]
 [-0.00222222  0.01111111]]
A^{-1} A =
[[1.00000000e+00 0.00000000e+00]
 [2.77555756e-17 1.00000000e+00]]
A A^{-1} =
[[1.00000000e+00 0.00000000e+00]
 [2.77555756e-17 1.00000000e+00]]
```

Summing or averaging along rows or columns in `numpy`

- Many times we want to compute the sum or mean along rows or columns of a matrix
- We can do this using `np.sum` (or `np.mean`) or directly call the method of a numpy array `A.sum` or `A.mean`
- NOTE: The `axis` argument is very important.
  - `axis=None` is full sum/mean of all entries in matrix/array

- `axis=0` is sum along the rows
- `axis=1` is sum along the columns

In [66]:
```
A = np.arange(6).reshape(2,3)
print(f'A\n{A}')
print(f'np.sum(A)\n{np.sum(A)}')
print(f'Row sum: np.sum(A, axis=0)\n{np.sum(A, axis=0)}')
print(f'Column sum: np.sum(A, axis=1)\n{np.sum(A, axis=1)}')
```

```
A
[[0 1 2]
 [3 4 5]]
np.sum(A)
15
Row sum: np.sum(A, axis=0)
[3 5 7]
Column sum: np.sum(A, axis=1)
[ 3 12]
```

In [67]:
```
A = np.arange(6).reshape(2,3)
print(f'A\n{A}')
print(f'np.mean(A)\n{np.mean(A)}')
print(f'Row mean: np.mean(A, axis=0)\n{np.mean(A, axis=0)}')
print(f'Column mean: np.mean(A, axis=1)\n{np.mean(A, axis=1)}')
```

```
A
[[0 1 2]
 [3 4 5]]
np.mean(A)
2.5
Row mean: np.mean(A, axis=0)
[1.5 2.5 3.5]
Column mean: np.mean(A, axis=1)
[1. 4.]
```

Singular matrices

- Informally, singular matrices are matrices that do not have an inverse (similar to the idea that 0 does not have an inverse)
- Consider the 1D equation $ax = b$
  - Usually we can solve for x by multiplying both sides by $1/a$
  - But what if $a = 0$?
  - What are the solutions to the equation?
- Called "singular" because a random matrix is unlikely to be singular, just like choosing a random number is unlikely to be 0.

In [68]:
```
from numpy.linalg import LinAlgError
def try_inv(A):
    print('A = ')
    print(np.array(A))
    try:
        np.linalg.inv(A)
    except LinAlgError as e:
        print(e)
    else:
        print('Not singular!')
    print()

try_inv([[0, 0], [0, 0]])
try_inv(np.eye(3))
try_inv([[1, 1], [1, 1]])
try_inv([[1, 10], [1, 10]])
try_inv([[2, 20], [4, 40]])
try_inv([[2, 20], [40, 4]])
```

```
A =
[[0 0]
 [0 0]]
Singular matrix

A =
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
Not singular!

A =
[[1 1]
 [1 1]]
Singular matrix

A =
[[ 1 10]
 [ 1 10]]
Singular matrix

A =
[[ 2 20]
 [ 4 40]]
Singular matrix

A =
[[ 2 20]
 [40  4]]
Not singular!
```

In [69]:
```
# Random matrix is very unlikely to be 0
for j in range(10):
    try_inv(np.random.randn(2, 2))
```

```
A =
[[-0.98488252  0.30120466]
 [-0.81247813  1.83108882]]
Not singular!

A =
[[-1.62972196 -1.29554373]
 [ 0.40803626 -0.17574461]]
Not singular!

A =
[[-0.29187122 -0.31441566]
 [-0.88843359 -0.46143531]]
Not singular!

A =
[[-0.42821856  0.73764165]
 [ 0.04247994 -0.61301516]]
Not singular!

A =
[[ 0.74635315  1.13370144]
 [-0.88846979 -0.74826847]]
Not singular!

A =
[[ 0.34401182  1.17651912]
 [-0.73487351 -0.3446669 ]]
Not singular!

A =
[[-1.138562    0.28076968]
 [-1.01990324  0.62218771]]
Not singular!

A =
[[-0.10438993 -1.92831209]
 [-0.6782082   1.25382139]]
Not singular!

A =
[[-0.20612421 -0.07080544]
 [-0.5240981  -0.471993  ]]
Not singular!

A =
[[ 1.25702837 -1.47350687]
 [ 1.34346636  2.02330837]]
Not singular!
```

System of equations in matrix form

- Example:

$$\begin{aligned} 2x &+ 3y &&= 6 \\ 4x &+ 9y &&= 15. \end{aligned}$$

  Solution is $x = \frac{3}{2}, y = 1$

- More general example:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 &= b_2 \\ a_{3,1}x_1 + a_{3,2}x_2 + a_{3,3}x_3 &= b_3 \end{aligned}$$

  is **equivalent** to:

$$A\mathbf{x} = \mathbf{b}$$

  where $A \in \mathrm{R}^{3,3}$, $\mathbf{x} \in \mathrm{R}^3$ and $\mathbf{b} \in \mathrm{R}^3$.