Matrices-student

September 9, 2024

1 Linear Algebra and Python Lists

This notebook will review some important concepts in linear algebra while helping you practice working with lists and nested lists in Python. You will provide code cells as needed to complete the sections below. Testing code is provided for you to help check that your methods perform as expected.

[ ]:
```
import numpy as np
from matplotlib import pyplot as plt
import math
```

1.1 Dot product of two vectors

$\vec{a} \cdot \vec{b} = \sum_i a_i b_i$ is the dot product (or inner product) between $\vec{a}$ and $\vec{b}$. Write a python function `dot_product(a,b)` that returns a dot product of two input python lists (do not use numpy arrays yet). If the input types do not match, raise a ValueError.

[ ]:
```
## Your code goes here. Insert cells as needed.
```

The following cells will check your dot_product method for correctness and for catching errors.

[ ]:
```
assert(dot_product([1,3],[2,6])==20)
```

[ ]:
```
try:
    dot_product([1,2,3],[4,5])
    print("Shouldn't get here")
except ValueError:
    print("Size mismatched caught")
```

1.2 Cross Product of two vectors

Given two real 3-vectors **a** and **b**:

$$\mathbf{A} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

The cross product of **a** and **b** is a new vector **C** defined as:

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix}$$

This resulting vector $\mathbf{C}$ is orthogonal (perpendicular) to both $\mathbf{a}$ and $\mathbf{b}$, and its magnitude is equal to the area of the parallelogram formed by $\mathbf{a}$ and $\mathbf{b}$.

The direction of $\mathbf{C}$ is determined by the right-hand rule.

The cross product of $\mathbf{a}$ and $\mathbf{b}$ can be expressed as the determinant of a 3x3 matrix:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

Where $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ are the unit vectors along the x, y, and z axes, respectively.

To compute the determinant, expand along the first row (using cofactor expansion):

$$\mathbf{a} \times \mathbf{b} = \mathbf{i} \begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \end{vmatrix} - \mathbf{j} \begin{vmatrix} a_1 & a_3 \\ b_1 & b_3 \end{vmatrix} + \mathbf{k} \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix}$$

Each of these 2x2 determinants is computed as follows:

- For the $\mathbf{i}$ component:
$$\mathbf{i} \cdot (a_2 b_3 - a_3 b_2)$$
- For the $\mathbf{j}$ component (note the negative sign):
$$-\mathbf{j} \cdot (a_1 b_3 - a_3 b_1)$$
- For the $\mathbf{k}$ component:
$$\mathbf{k} \cdot (a_1 b_2 - a_2 b_1)$$

Thus, the cross product is:

$$\mathbf{a} \times \mathbf{b} = (a_2 b_3 - a_3 b_2)\,\mathbf{i} - (a_1 b_3 - a_3 b_1)\,\mathbf{j} + (a_1 b_2 - a_2 b_1)\,\mathbf{k}$$

Or, written in vector form:

$$\mathbf{a} \times \mathbf{b} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix}$$

This is the cross product of the vectors $\mathbf{a}$ and $\mathbf{b}$, derived from the determinant of a 3x3 matrix.

Write a python method which returns the cross product of 2 3D vectors, and throws an error in case of invalid input

```
## Your code goes here. Insert cells as needed.
```

These cells will check your method

```
assert(cross_product([1,3,2],[5,7,2])==[-8,8,-8])
```

```
try:
    cross_product([1,2,3],[4,5])
    print("Shouldn't get here")
except ValueError:
    print("Size mismatched caught")
```

1.3 Vector magnitude

The magnitude, or length, of a vector is defined as

$||\vec{x}|| = \sqrt{\vec{x} \cdot \vec{x}} = \sqrt{x_0^2 + x_1^2 + \cdots + x_{n-1}^2}$. Notice the exponent on each element is 2 and the radical exponent is $\frac{1}{2}$. This is an L_2 norm. If we compute the value $\sqrt[3]{x_0^3 + x_1^3 + \cdots + x_{n-1}^3}$, this is an L_3 norm. The sum of absolute values $|x_1| + |x_2| + \cdots + |x_n|$ is called the L_1 norm. All of these are p-norms, with the $p = 2$ being the most familiar, and the $p = 1$ norm being useful in several settings.

Write a python method `norm(a)` which returns the 2-norm (length) of a vector, by calling `dot_product`. Then write `p_norm(a,p)` which evaluates the p-norm for any integer $p \geq 1$ and throws an error if needed.

```
## Your code goes here. Insert cells as needed.
```

This cell will check your method

```
assert(norm([1,-1,1,-1])==2)
assert(norm([0])==0)
assert(norm([30,40,50])==np.sqrt(5000))
```

```
## Your code goes here. Insert cells as needed.
```

These cells will check your `p_norm`

```
assert(p_norm([1,-1,0],1)==2)
assert(p_norm([1,-1,1,-1],2)==2)
assert(p_norm([2,-2,4,-4],3)==0)
```

```
try:
    p_norm([1,2,3],0.5)
    print("Shouldn't get here")
except ValueError:
    print("Invalid p caught")
```

1.4 Angle between vectors

A very important result from linear algebra used in machine learning relates the angle between two vectors. You will derive this yourself. Given **a** and **b**, you can form a triangle where the vectors share the same tail. The vector forming the third side is the vector **a-b**. Find the length of this third side in terms of **a** and **b**. Hint: use the law of cosines. Hint: dot product distributes like 'times'

Using your result write a function `angle_between` that returns the smallest angle between two vectors **a** and **b**. Make sure your result is in the range $[0, \pi]$

[ ]:
```
## Your code goes here. Insert cells as needed.
```

[ ]:
```
assert(abs(angle_between([1,1],[1,0])-math.pi/4)<1e-9)
assert(abs(angle_between([1,0],[1,0]))<1e-9)
assert(abs(angle_between([1,0],[0,1])-math.pi/2)<1e-9)
assert(abs(angle_between([1,2,3,4],[-4,-3,-2,-1])-2.300523983021863)<1e-9)
```

[ ]:
```
try:
    angle_between([1,2,3],[4,5])
    print("Shouldn't get here")
except ValueError:
    print("Size mismatch caught")
```

1.5 Matrix Operations

Matrix addition and subtraction are componentwise and require matrices of the same size: $m_{ij} = a_{ij} + b_{ij}$ or $m_{ij} = a_{ij} - b_{ij}$

Matrix multiplication involves one entire row and one entire column of each matrix to determine an entry in the product. It can be written as

$$m_{ij} = a_{ik} b^{kj}$$

Where we are using *Einstein notation*. Each repeated index (here k) is an index of summation. Thus

$$a_{ik} b^{kj} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

Matrix multiplication requires an $(m_1 \times n_1)$ matrix multiplied on the right by an $(m_2 \times n_2)$ matrix where $n_1 = m_2$. The result is an $(m_1 \times n_2)$ matrix.

Write three methods: `mat_add`, `mat_sub`, `mat_mul`. You should throw an error if dimensions do not match. You should store all matrices as 2D lists in python, i.e. lists of lists. The i, j element is referenced by `M[i][j]`, both starting at 0.

[ ]:
```
## Your code goes here. Insert cells as needed.
```

[ ]:
```
A=[[1]]
B=[[2]]
assert(mat_add(A,B)==[[3]])

A=[[-4,3],[1,-10]]
B=[[2,-5],[-9,1]]
assert(mat_add(A,B)==[[-2,-2], [-8,-9]])
assert(mat_add(B,A)==[[-2,-2],[-8,-9]])

A = [[1,2,3],[5,3,-1],[6,5,2]]
B = [[4,-2,2],[2,4,3],[6,2,2]]
assert(mat_add(A,B)==[[5,0,5],[7,7,2],[12,7,4]])
```

[ ]:
```
try:
    mat_add([[1,2,3],[4,5,6]],[[1,2],[3,4]])
    print("Shouldn't get here")
except ValueError:
    print("Size mismatch caught")
```

[ ]:
```
## Your code goes here. Insert cells as needed.
```

[ ]:
```
A=[[1]]
B=[[2]]
assert(mat_transpose(mat_transpose(A))==A)
assert(mat_transpose(mat_transpose(B))==B)

A=[[-4,3],[1,-10]]
B=[[2,-5],[-9,1]]
assert(mat_transpose(mat_transpose(A))==A)
assert(mat_transpose(mat_transpose(B))==B)

A = [[1,2,3],[5,3,-1],[6,5,2]]
B = [[4,-2,2],[2,4,3],[6,2,2]]
assert(mat_transpose(mat_transpose(A))==A)
assert(mat_transpose(mat_transpose(B))==B)
```

[ ]:
```
## Your code goes here. Insert cells as needed.
```

[ ]:
```
A=[[1]]
B=[[2]]
assert(mat_mul(A,B)==[[2]])

A=[[-4,3],[1,-10]]
B=[[2,-5],[-9,1]]
assert(mat_mul(A,B)==[[-35, 23], [92, -15]])
assert(mat_mul(B,A)==[[-13, 56], [37, -37]])
```

```
A = [[1,2,3],[5,3,-1],[6,5,2],[-10,2,3]]
B = [[4,-2,3,2],[2,4,3,-1],[6,3,2,2]]
C = [[2,-1,-3,4],[3,0,0,-2],[2,-3,7,-9]]

assert(mat_mul(A,B)==[[26, 15, 15, 6], [20, -1, 22, 5], [46, 14, 37, 11], [-18,␣
 ↪37, -18, -16]])

assert(mat_transpose(mat_mul(A,B)) ==␣
 ↪mat_mul(mat_transpose(B),mat_transpose(A)))
assert(mat_add(mat_mul(A,B),mat_mul(A,C)) == mat_mul(A, mat_add(B,C)))
```

[ ]:
```
try:
    mat_mul(mat_transpose(C),A)
    print("Shouldn't get here")
except ValueError:
    print("Matrix incompatibility caught")
```