My Notes for Machine Learning

Philip Tracton

Contents

1 Math Review

1.1 Linear Algebra

1.1.1 Matrix

A matrix is a 2-dimensional array of numbers.

N is number of rows. **M** is number of columns.

Matrices are specified as NxM.

A_{ij} is how you specify a single location in a matrix. It is row I and column J.

$$\mathbf{A} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1m} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NM} \end{bmatrix} \tag{1.1}$$

A vector is a matrix with one column and many rows and specified as Nx1

v_i is the i^{th} element of the vector V.

$$\mathbf{v} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \tag{1.2}$$

Matrices are usually denoted by uppercase names while vectors are lowercase.

Scalar means that an object is a single value, not a vector or matrix.

$\mathbb{R}$ refers to the set of scalar real numbers.

$\mathbb{R}^n$ refers to the set of n-dimensional vectors of real numbers.

Matlab (or Octave) and Python are 2 very common languages for doing machine learning and math work in general. They are very different languages. Matlab is a number crunching specific language. This is pretty much all it does. Python is a general purpose language with a lot of built up libraries, Numpy in particular, to support doing this type of work.

This document will try to go over all concepts in both languages in order to better understand how the math works from 2 different perspectives.

One of the issues to keep track of is that Matlab starts counting at 1 and Python starts counting at 0! Notice in the code below that A(2,3) in Matlab will get you the same location as A[1][2] in Python for the same matrices

This is an example of creating a simple 3x3 matrix in both Matlab and Python. Notice that Matlab is a little bit simpler. There is no need to import a library for it. Python doesn't need one either technically, but numpy will be used a lot for machine learning work and we should just start with it.

Matlab

```
% The ; denotes we are going back to a new row.
A = [1, 2, 3; 4, 5, 6; 7, 8, 9; 10, 11, 12]

% Initialize a vector
v = [1;2;3]

% Get the dimension of the matrix A where m = rows and n = columns
[m,n] = size(A)

% You could also store it this way
dim_A = size(A)

% Get the dimension of the vector v
dim_v = size(v)

% Now let's index into the 2nd row 3rd column of matrix A
A_23 = A(2,3)
```

Python

```
#! /usr/bin/env python3

import numpy as np

A = np.array([[1, 2, 3], [3, 4, 5], [7, 8, 9]])
rows, cols = np.shape(A)
print("\nA= {}".format(A))
print("Rows = {} Cols = {}".format(rows, cols))
print("Location 2,3 = {}".format(A[1][2]))

V = np.array([[1], [2], [3], [4], [5], [6], [7], [8]])
rows, cols = np.shape(V)
print("\nV= {}".format(V))
print("Rows = {} Cols = {}".format(rows, cols))
print("Location 6,1 = {}".format(V[5][0]))
```

1.1.2 Matrix Operations

Addition and Subtraction

Addition and subtraction are element-wise, so you simply add or subtract each corresponding element:
To add or subtract two matrices, their dimensions must be the **same**.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a+w & b+x \\ c+y & d+z \end{bmatrix} \tag{1.3}$$

Matlab

```matlab
A = [1 2; 3 4]
B = [11 12; 13 14]
C = A + B
```

Python

```python
#! /usr/bin/env python3

import numpy as np

A = np.array(([1, 2], [3, 4]))
B = np.array(([11, 12], [13, 14]))
C = A + B
print("A = {}".format(A))
print("B = {}".format(B))
print("C = {}".format(C))
```

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} - \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a-w & b-x \\ c-y & d-z \end{bmatrix} \tag{1.4}$$

Matlab

```matlab
A = [1 2; 3 4]
B = [11 12; 13 14]
C = A - B
```

Python

```python
#! /usr/bin/env python3

import numpy as np

A = np.array(([1, 2], [3, 4]))
B = np.array(([11, 12], [13, 14]))
C = A - B
print("A = {}".format(A))
print("B = {}".format(B))
print("C = {}".format(C))
```

For matrix addition/subtraction there is not much of a difference. Python takes a little more set up in that you need to import numpy, but the actual operational step is identical.

Scalar Multiplication and Division

In scalar multiplication, we simply multiply every element by the scalar value:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * x = \begin{bmatrix} a*x & b*x \\ c*x & d*x \end{bmatrix} \tag{1.5}$$

In scalar division, we simply divide every element by the scalar value:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} / x = \begin{bmatrix} a/x & b/x \\ c/x & d/x \end{bmatrix} \tag{1.6}$$

Matlab

```matlab
A = [10 20; 30 40]
x = 10
C = A*x
D = A/x
```

Python

```python
#! /usr/bin/env python3

import numpy as np

A = np.array(([10, 20], [30, 40]))
x = 10
C = A * x
D = A/x
print("A = {}".format(A))
print("C = {}".format(C))
print("D = {}".format(D))
```

Matrix Vector Multiplication

The result is a **vector**. The number of **columns** of the matrix must equal the number of **rows** of the vector.

An **m x n matrix** multiplied by an **n x 1** vector results in an **m x 1** vector.

Some more Math Insights

Below is an example of a matrix-vector multiplication. Make sure you understand how the multiplication works.

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a*x+b*y \\ c*x+d*y \\ e*x+f*y \end{bmatrix} \tag{1.7}$$

Matrix Matrix Multiplication

This is also known as the dot product.

An **m x n** matrix multiplied by an **n x o** matrix results in an **m x o** matrix. In the example, a 3 x 2 matrix times a 2 x 2 matrix resulted in a 3 x 2 matrix.

To multiply two matrices, the number of columns of the first matrix must equal the number of rows of the second matrix. The process is to take each row of the first matrix and multiply it by each column of the second matrix. Iterate this through each row in the first matrix with each column in the second matrix.

You can **NOT** reverse the order. $\mathbf{A} * \mathbf{B}$ is not $\mathbf{B} * \mathbf{A}$
Multiplication is associative. $(\mathbf{A} * \mathbf{B}) * \mathbf{C} = \mathbf{A} * (\mathbf{B} * \mathbf{C})$

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a*w+b*y & a*x+b*z \\ c*w+d*y & c*x+d*z \\ e*w+f*y & e*x+f*z \end{bmatrix} \tag{1.8}$$

Matlab

```
A = [10 20; 30 40; 50 60]
B = [5; 10]
C = A* B
% The line below fails since A*B is
  not B*A
%D = B *A
```

Python

```
#! /usr/bin/env python3

import numpy as np

A = np.array(([10, 20], [30, 40],
  [50, 60]))
B = np.array(([5], [10]))
C = A.dot(B)
print("A = {}".format(A))
print("B = {}".format(B))
print("C = {}".format(C))
# The line below fails since A*B is
  not B*A
#D = B.dot(A)
```

Notice the syntax is starting to differ more. For Matlab, you can just multiply the vectors like any other variable. In python we need to use the dot method. Notice it is A that calls dot with B as a parameter.

Identity

The identity matrix is a square matrix (m=n) that has 1's along the diagnal and zeros everywhere else and is usually denoted by the letter **I**.
The identity matrix, when multiplied by any matrix of the same dimensions, results in the original matrix. It's just like multiplying numbers by 1. The identity matrix simply has 1's on the diagonal (upper left to lower right diagonal) and 0's elsewhere.

When multiplying the identity matrix after some matrix (AI), the square identity matrix's dimension should match the other matrix's **columns**. When multiplying the identity matrix before some other matrix (IA), the square identity matrix's dimension should match the other matrix's **rows**.

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \tag{1.9}$$

Matlab

```matlab
% Initialize random matrices A and B
A = [1,2;4,5]
B = [1,1;0,2]

% Initialize a 2 by 2 identity matrix
I = eye(2)

% The above notation is the same as I = [1,0;0,1]

% What happens when we multiply I*A ?
IA = I*A

% How about A*I ?
AI = A*I

% Compute A*B
AB = A*B

% Is it equal to B*A?
BA = B*A

% Note that IA = AI but AB != BA
```

Python

```python
#! /usr/bin/env python3

import numpy as np

# Initialize random matrices A and B
A = np.array(([1, 2],
              [4, 5]))
B = np.array(([1, 1],
              [0, 2]))

# Initialize a 2 by 2 identity matrix
I = np.eye(2)

# The above notation is the same as I = [1, 0
#                                        0, 1]

# What happens when we multiply I*A ?
IA = I.dot(A)

# How about A*I ?
AI = A.dot(I)

# Compute A*B
AB = A.dot(B)

# Is it equal to B*A?
BA = B.dot(A)

# Note that IA = AI but AB != BA
print("A = {}".format(A))
print("B = {}".format(B))
print("IA = {}".format(IA))
print("AI = {}".format(AI))
print("AB = {}".format(AB))
print("BA = {}".format(BA))
```

Again the syntax is still mostly the same. Matlab multiplies are calls to the dot method in Python. The eye() function in Matlab does the same thing as the numpy version of eye(). The difference is that in Matlab it is a built in function and in Python you need to call the method inside the Numpy library.

Transpose

The **transposition** of a matrix is like rotating the matrix 90 in clockwise direction and then reversing it. We can compute transposition of matrices in matlab with the transpose(A) function or A'

In other words: $A_{ij} = A^T_{ji}$

£

Matlab

```
% Initialize matrix A
A = [1,2,0;0,5,6;7,0,9]

% Transpose A
A_trans = A'
```

Python

```
#! /usr/bin/env python3

import numpy as np

A = np.array(([1, 2, 0], [0, 5, 6],
    [7, 0, 9]))
A_trans = A.transpose()
print("A = {}".format(A))
print("A_trans = {}".format(A_trans))
```

In Matlab the ' operator will transpose a matrix. There is a transpose function in Matlab but the ' notation is very common and easier.

In Python we must call the transpose method on our matrix.

Inverse

The inverse of a matrix A is denoted A^{-1}. Multiplying by the inverse results in the identity matrix.

$$I = A * A^{-1} \tag{1.10}$$

A non square matrix does not have an inverse matrix. We can compute inverses of matrices in Octave with the pinv(A) function and in Matlab with the inv(A) function. Matrices that don't have an inverse are singular or degenerate.

£

Matlab

```
% Initialize matrix A
A = [1,2,0;0,5,6;7,0,9]

% Take the inverse of A
A_inv = inv(A)

% What is A^(-1)*A?
A_invA = inv(A)*A
```

Python

```
#! /usr/bin/env python3

import numpy as np
import numpy.linalg

A = np.array(([1, 2, 0], [0, 5, 6],
 ↪ [7, 0, 9]))
A_inv = numpy.linalg.inv(A)
A_invA = A.dot(A_inv)
print("A = {}".format(A))
print("A_inv = {}".format(A_inv))
print("A_invA = {}".format(A_invA))
```

The Matlab code is pretty straight forward. You can call the inv() function on a matrix. You can multiply the original matrix by its inverse and get the identity.

In Python it is a little more complicated. We need to bring in the numpy Linear Algebra library. Once this library is brought in, we can use the inv() method in it on our matrix. From there we can do all the same things as in Matlab but with our Python syntax.

1.2 Calculus

1.3 Probability

1.4 Statistics

2 Introduction to Machine Learning

2.1 Linear Regression

2.2 Logistic Regression