Brief 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)

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Scalars

- Single number
- Denoted as lowercase letter
- Examples
  - $x \in \mathbb{R}$ - Real number
  - $w \in \mathbb{C}$ - Complex number (i.e., with imaginary part)
  - $z \in \mathbb{Z}$ - Integer
  - $i \in \mathbb{Z}_+$ - Non-negative integers
  - $y \in \{0, 1, \dots, C\}$ - Finite set
  - $u \in [0, 1]$ - Bounded set

In [2]:
```
x = 1.1343
print(x)
w = 1.1343 + 2.1j
print(w)
z = int(-5)
print(z)
```

```
1.1343
(1.1343+2.1j)
-5
```

Vectors

- Array of numbers
- 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
  - $\mathbf{x} \in \mathbb{R}^d$ - Real number
  - $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$
  - $\mathbf{x} = [x_1, x_2, \dots, x_d]^T$
  - $\mathbf{z} = [\sqrt{x_1}, \sqrt{x_2}, \dots, \sqrt{x_d}]^T$
  - $\mathbf{y} \in \{0, 1, \dots, C\}^d$ - Finite set
  - $\mathbf{u} \in [0, 1]^d$ - Bounded set

In [3]:
```
x = np.array([1.1343, 6.2345, 35])
print(x)
w = np.array([1.1343 + 2.1j, 1j, 0.1 + 3.5j])
print(w)
z = 5 * np.ones(3, dtype=int)
print(z)
```

```
[ 1.1343  6.2345 35.    ]
[1.1343+2.1j 0.    +1.j  0.1   +3.5j]
[5 5 5]
```

Note: 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 [4]:
```
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]
```

Matrices

- 2D array of numbers
- Denoted as uppercase letter
- Number of samples often denoted by n or N.
- Access rows or columns via subscript or numpy notation:
  - $X_{i,:}$ is the i-th row, $X_{:,j}$ is the jth column
  - (Sometimes) X_i, $\mathbf{x}_i$ is the i-th row or column depending on context
- Access elements by double subscript $X_{i,j}$ or $x_{i,j}$ is the i,j-th entry of the matrix
- Examples
  - $X \in \mathbb{R}^{n \times d}$ - Real number
  - $X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ - Real number
  - $Y \in \{0, 1, \ldots, C\}^{k \times d}$ - Finite set
  - $U \in [0, 1]^{n \times d}$ - Bounded set

In [5]:
```
X = np.arange(12).reshape(3,4)
print(X)
W = np.array([
    [1.1343 + 2.1j, 1j, 0.1 + 3.5j],
    [3, 4, 5],
])
print(W)
Z = 5 * np.ones((3, 3), dtype=int)
print(Z)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[1.1343+2.1j 0.    +1.j  0.1   +3.5j]
 [3.    +0.j  4.    +0.j  5.    +0.j ]]
[[5 5 5]
 [5 5 5]
 [5 5 5]]
```

Tensors

- n-D arrays
- Examples
  - $X \in \mathbb{R}^{3\times m\times m}$, single color image in PyTorch
  - $X \in \mathbb{R}^{n\times 3\times m\times m}$, multiple color images in PyTorch
  - $X \in \mathbb{R}^{m\times m\times 3}$, single color image for matplotlib imshow

In [6]:
```
from sklearn.datasets import load_sample_image
china = load_sample_image('china.jpg')
print('Shape of image (height, width, channels):', china.shape)
ax = plt.axes(xticks=[], yticks=[])
ax.imshow(china);
```

```
Shape of image (height, width, channels): (427, 640, 3)
```

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

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

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

In [7]:
```
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.

In [8]:
```
v = np.arange(5)
print('A numpy vector', v)
print('Transpose of numpy vector', v.T)
print('A matrix with one column')
V = v.reshape(-1, 1)
print('V shape: ', V.shape)
print(V)
```

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

Matrix product

- Let $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$, then the **matrix product** $C = AB$ is defined as:

$$c_{i,j} = \sum_{k \in \{1,2,\ldots,n\}} a_{i,k} b_{k,j}$$

  where $C \in \mathbb{R}^{m \times p}$ (notice how inner dimension is collapsed.
- (Show on board visually)

In [9]:
```
A = np.arange(6).reshape(3, 2)
print(A)
B = np.arange(6).reshape(2, 3)
print(B)
C = np.zeros((A.shape[0], B.shape[1]))
for i in range(C.shape[0]):
    for j in range(C.shape[1]):
        for k in range(A.shape[1]):
            C[i, j] += A[i, k] * B[k, j]
print(C)
print(np.dot(A, B))
```

```
[[0 1]
 [2 3]
 [4 5]]
[[0 1 2]
 [3 4 5]]
[[ 3.  4.  5.]
 [ 9. 14. 19.]
 [15. 24. 33.]]
[[ 3  4  5]
 [ 9 14 19]
 [15 24 33]]
```

Notice triple loop, naively cubic complexity $O(n^3)$

However, special linear algebra algorithms can do it $O(n^2.803)$

Takeaway - Use numpy `np.dot` or `np.matmult`

Element-wise (Hadamard) product *NOT equal* to matrix multiplication

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

In [10]:
```
A = np.arange(6).reshape(3, 2)
print(A)
B = np.arange(6).reshape(2, 3)
print(B)
try:
    A * B  # Fails since matrix shapes don't match and cannot broadcast
except ValueError as e:
    print('Operation failed! Message below:')
    print(e)
```

```
[[0 1]
 [2 3]
 [4 5]]
[[0 1 2]
 [3 4 5]]
Operation failed! Message below:
operands could not be broadcast together with shapes (3,2) (2,3)
```

In [11]:
```
print(A)
D = 10*B.T
print(D)
F = A * D  # Element-wise / Hadamard product
print(F)
```

```
[[0 1]
 [2 3]
 [4 5]]
[[ 0 30]
 [10 40]
 [20 50]]
[[  0  30]
 [ 20 120]
 [ 80 250]]
```

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 [12]:
```
print('AB')
print(np.dot(A, B))
print('BA')
print(np.dot(B, A))
print('(AB)^T')
print(np.dot(A, B).T)
print('B^T A^T')
print(np.dot(B.T, A.T))
```

```
AB
[[ 3  4  5]
 [ 9 14 19]
 [15 24 33]]
BA
[[10 13]
 [28 40]]
(AB)^T
[[ 3  9 15]
 [ 4 14 24]
 [ 5 19 33]]
B^T A^T
[[ 3  9 15]
 [ 4 14 24]
 [ 5 19 33]]
```

Properties of inner product or vector-vector product

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

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

  Also denoted as:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T\mathbf{y}$$

  Can be executed via `np.dot`

In [13]:
```
# Inner product
a = np.arange(3)
print(a)
b = np.array([11, 22, 33])
print(b)
np.dot(a, b)
```

```
[0 1 2]
[11 22 33]
```

Out[13]:
```
88
```

Linear set of equations can be compactly represented as matrix equation

- 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 $A \in \mathbb{R}^{3,3}$, $\mathbf{x} \in \mathbb{R}^3$ and $\mathbf{b} \in \mathbb{R}^3$.

Identity matrix keeps vectors unchanged

- Multiplying by the identity does not change vector (generalizing the concept of the scalar 1)
- Formally, $I_n \in \mathbb{R}^{n \times n}$, and $\forall \mathbf{x} \in \mathbb{R}^n$, $I_n\mathbf{x} = \mathbf{x}$
- Structure is ones on the diagonal, zero everywhere else:
- `np.eye` function to create identity

In [14]:
```
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.]]
[-0.13829299  0.78492995 -0.26939102]
[-0.13829299  0.78492995 -0.26939102]
```

Matrix inverse times the original matrix is the identity

- The inverse of *square* matrix $A \in \mathbb{m} \times \mathbb{m}$ 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 [15]:
```
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]]
```

Singular matrices are similar to zeros

- 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 [16]:
```
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 [17]:
```
# Random matrix is very unlikely to be 0
for j in range(10):
    try_inv(np.random.randn(2, 2))
```

```
A =
[[-0.90550148 -0.02532468]
 [-0.89130022 -0.80571341]]
Not singular!

A =
[[ 0.4087488  -0.20136883]
 [ 2.23151464  0.87926746]]
Not singular!

A =
[[-1.02709192 -0.2750303 ]
 [-0.28645797 -2.02844463]]
Not singular!

A =
[[-0.02248636  0.72314   ]
 [ 1.60885039 -0.22551542]]
Not singular!

A =
[[ 1.53234123  1.24501593]
 [-1.84021119 -0.27807602]]
Not singular!

A =
[[-2.7548314   0.90938852]
 [ 1.11689209  0.72669141]]
Not singular!

A =
[[-0.67333293  0.83524065]
 [ 1.71028296 -0.3347097 ]]
Not singular!

A =
[[ 0.14837056  1.8609308 ]
 [-0.68470719  0.78144661]]
Not singular!

A =
[[-2.71647167  1.11170123]
 [-0.0056665   0.19595604]]
Not singular!

A =
[[ 0.68408856 -0.08419105]
 [-1.62402249  1.54859037]]
Not singular!
```

Next time: Norms, unitary matrices, eigendecomposition, singular value decomposition, determinant