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]:

```python
import numpy as np
import matplotlib.pyplot as plt
```

### Scalars

- Single number
- Denoted as lowercase letter
- Examples
  - \(x \in \mathbb{R}\) - Real number
  - \(y \in \{0, 1, \ldots, C\}\) - Finite set
  - \(u \in [0, 1]\) - Bounded set

In [2]:

```python
x = 1.1343
print(x)
z = int(-5)
print(z)
```

```
1.1343
-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
  - \(x \in \mathbb{R}^d\)
  - \(x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}\)
  - \(x = [x_1, x_2, \ldots, x_d]^T\)
  - \(z = [\sqrt{x_1}, \sqrt{x_2}, \ldots, \sqrt{x_d}]^T\)
  - \(y \in \{0, 1, \ldots, C\}^d\) - Finite set
  - \(u \in [0, 1]^d\) - Bounded set
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 [3]:
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]
```

```
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]
```

```
In [5]:
type(a_list)
```

```
Out[5]: list
```

```
In [6]:
type(a)
```

```
Out[6]: numpy.ndarray
```

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 `j`-th column
  - (Sometimes) `X_i`, `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
  - \( X \in \mathbb{R}^{nxd} \) - 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 [7]:
```python
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 [8]: 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

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

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

In [9]: 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.
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 [10]:
```
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]]

In [11]:
```
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.matmul(A, B))
```  

[[0 1]  
 [2 3]  
 [4 5]]  
[[0 1 2]  
 [3 4 5]]  
[[ 3.  4.  5.]  
 [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.matmul` or @ operator for matrix multiplication

(np.dot also works for matrix multiplication but is different in PyTorch and is less explicit so I suggest the two methods above for matrix multiplication)

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

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

```python
In [12]:
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)
```
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): $(AB)^T = B^T A^T$

Properties of inner product or vector-vector product

```
In [13]:
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]]
```

```
In [14]:
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
[[ 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]]
```

```
```
- Inner product or vector-vector multiplication produces scalar: \( x^T y = (x^T y)^T = y^T x \)

Also denoted as:

\( \langle x, y \rangle = x^T y \)

Can be executed via \( \text{np.dot} \) or \( \text{np.matmul} \)

In [15]:
```python
# 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[15]: 88

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 \( \forall x \in \mathbb{R}^n, I_n x = x \)
- Structure is ones on the diagonal, zero everywhere else:
- \( \text{np.eye} \) function to create identity

In [16]:
```python
I3 = np.eye(3)
print(I3)
x = np.random.randn(3)
print(x)
print(np.matmul(I3, x))
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[1.45901765 0.6176544 0.10913208]
[1.45901765 0.6176544 0.10913208]
```

Matrix inverse times the original matrix is the identity

- The inverse of square matrix \( A \in \mathbb{R}^{n \times n} \) 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 [17]:
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.matmul(Ainv, A))
print('A A^{-1} = ')
print(np.matmul(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]]

**Linear set of equations can be compactly represented as matrix equation**

- Example:
  
  \[
  \begin{align*}
  2x + 3y &= 6 \\
  4x + 9y &= 15.
  \end{align*}
  \]

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

- More general example:
  
  \[
  \begin{align*}
  a_{1,1} x_1 + a_{1,2} x_2 + a_{1,3} x_3 &= b_1 \\
  a_{2,1} x_1 + a_{2,2} x_2 + a_{2,3} x_3 &= b_2 \\
  a_{3,1} x_1 + a_{3,2} x_2 + a_{3,3} x_3 &= b_3
  \end{align*}
  \]

  is equivalent to:

  \[
  A x = b
  \]

  where \( A \in \mathbb{R}^{3 \times 3}, x \in \mathbb{R}^3 \) and \( b \in \mathbb{R}^3 \).

- If matrix inverse exists, then solution is

  \[
  x = A^{-1} b
  \]

**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 \( a x = b \)
  
  - 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 [18]: 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!
```
A =
\[
\begin{bmatrix}
0.62116151 & -1.01047326 \\
0.9207096 & 0.13609464
\end{bmatrix}
\]
Not singular!

A =
\[
\begin{bmatrix}
0.10241761 & 0.05638955 \\
0.6554859 & 0.81492455
\end{bmatrix}
\]
Not singular!

A =
\[
\begin{bmatrix}
-0.62152324 & 0.43003518 \\
-0.06451688 & -0.10078375
\end{bmatrix}
\]
Not singular!

A =
\[
\begin{bmatrix}
-0.06023321 & 1.72412948 \\
1.01745313 & 2.00707215
\end{bmatrix}
\]
Not singular!

A =
\[
\begin{bmatrix}
0.15428838 & 0.01666077 \\
-0.06106018 & 1.63095398
\end{bmatrix}
\]
Not singular!

A =
\[
\begin{bmatrix}
-0.65684713 & -0.16658363 \\
-0.55606557 & -0.00458845
\end{bmatrix}
\]
Not singular!

A =
\[
\begin{bmatrix}
-2.04915067 & -0.69560613 \\
0.02569157 & 0.6574612
\end{bmatrix}
\]
Not singular!

A =
\[
\begin{bmatrix}
0.13000679 & -1.43767639 \\
1.45339701 & 0.58621667
\end{bmatrix}
\]
Not singular!

A =
\[
\begin{bmatrix}
0.37263979 & 0.51563468 \\
-1.06825911 & -0.92117196
\end{bmatrix}
\]
Not singular!

A =
\[
\begin{bmatrix}
2.66511491 & -1.02085393 \\
1.40486011 & 0.9248407
\end{bmatrix}
\]
Not singular!

# Random matrix is very unlikely to be 0
for j in range(10):
    try_inv(np.random.randn(2, 2))
Norms: The "size" of a vector or matrix

- Informally, a generalization of the absolute value of a scalar
- Formally, a norm is a function \( f \) that has the following three properties:
  - \( f(\mathbf{x}) = 0 \Rightarrow \mathbf{x} = \mathbf{0} \) (zero point)
  - \( f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y}) \) (Triangle inequality)
  - \( \forall \alpha \in \mathbb{R}, f(\alpha \mathbf{x}) = |\alpha| f(\mathbf{x}) \) (absolutely homogenous)
- Examples
  - Absolute value of scalars
  - \( p \)-norm (also denoted \( \ell_p \)-norm)
    \[
    ||\mathbf{x}||_p = \left( \sum_{i=1}^{d} |x_i|^p \right)^{\frac{1}{p}}
    \]
    (Discussion) What does this represent when \( p = 2 \) (for simplicity you can assume \( d = 2 \))?  
    - When \( p = 2 \), we often merely denote as \( ||\mathbf{x}|| \).
    - What about when \( p = 1 \)?
    - What about when \( p = \infty \) (or more formally the limit as \( p \rightarrow \infty \))? 

In [20]:
```python
x = np.array([1, 1])
print(np.linalg.norm(x, ord=2))
print(np.linalg.norm(x, ord=1))
print(np.linalg.norm(x, ord=np.inf))
```

1.4142135623730951
2.0
1.0

Vectors that have the same norm form a "ball" that isn't necessarily circular

In [21]:
```python
rng = np.random.RandomState(0)
X = rng.randn(1000, 2)
p_vals = [1, 1.5, 2, 4, np.inf]
fig, axes = plt.subplots(1, len(p_vals), figsize=(len(p_vals)*4, 3))
for p, ax in zip(p_vals, axes):
    # Normalize them to have the unit norm
    Z = (X.T / np.linalg.norm(X, ord=p, axis=1)).T
    ax.scatter(Z[:, 0], Z[:, 1])
    ax.axis('equal')
    ax.set_title('Unit Norm Ball for $p$=%g' % p)
```

![Unit Norm Balls](image)
Squared $L_2$ norm is quite common since it simplifies to a simple summation

$$\|\mathbf{x}\|_2^2 = \left( \sum_{i=1}^{d} |x_i|^2 \right)^{\frac{1}{2}} = \sum_{i=1}^{d} |x_i|^2 = \sum_{i=1}^{d} x_i^2$$

- Additionally, this can be computed as $\|\mathbf{x}\|_2^2 = \mathbf{x}^T \mathbf{x}$
- Informally, this is analogous to taking the square of a scalar number

In [22]:
```python
x = np.arange(4)
print(np.linalg.norm(x, ord=2)**2)
print(np.dot(x, x))
```

```
14.0
14
```

Orthogonal vectors

- Orthogonal vectors are vectors such that $\mathbf{x}^T \mathbf{y} = 0$
- The dot product between vectors can be written in terms of norms and the cosine of the angle:
  $$\mathbf{x}^T \mathbf{y} = \|\mathbf{x}\|_2 \|\mathbf{y}\|_2 \cos \theta$$
- (Discussion) Suppose $\mathbf{x}$ and $\mathbf{y}$ are non-zero vectors, what must $\theta$ be if the vectors are orthogonal?

In [23]:
```python
print(np.matmul([[0, 1], [1, 0]]))
theta = np.pi/2
x = np.array([[np.cos(theta), -np.sin(theta)]]
            [np.sin(theta), np.cos(theta)])
print(x)
print(y)
print(np.dot(x, y))
```

```

0
[ 6.123234e-17 -1.000000e+00]
[1.000000e+00 6.123234e-17]
0.0
```

Special matrices: Orthogonal matrices

- Informally, an orthogonal matrix only rotates (or reflects) vectors around the origin (zero point), but does not change the size of the vectors.
- Informally, almost analogous to a 1 or -1 for matrices but more general
- A square matrix such that $\mathbf{Q}^T \mathbf{Q} = \mathbf{Q} \mathbf{Q}^T = \mathbf{I}$
- Or, equivalently $\mathbf{Q}^{-1} = \mathbf{Q}^T$
- Or, equivalently:
  - Every column (or row) is orthogonal to every other column (or row)
Every column (or row) has unit ℓ₂-norm, i.e., \( \|Q_{i,:}\|_2 = \|Q_{:,j}\|_2 = 1 \)

In [24]:
```python
print('Identity matrix')
Q = np.eye(2) # Identity
print(Q)
print(np.allclose(np.eye(2), np.matmul(Q.T, Q)))

print('Reflection matrix')
Q = np.array([[1, 0], [0, -1]]) # Reflection
print(Q)
print(np.allclose(np.eye(2), np.matmul(Q.T, Q)))

print('Rotation matrix')
theta = np.pi/3
Q = np.array([[
    [np.cos(theta), -np.sin(theta)],
    [np.sin(theta), np.cos(theta)]
])
print(Q)
print(np.allclose(np.eye(2), np.matmul(Q.T, Q)))
```

Identity matrix
[[1. 0.]
 [0. 1.]]
True
Reflection matrix
[[ 1 0]
 [0 -1]]
True
Rotation matrix
[[ 0.5 -0.8660254]
 [ 0.8660254 0.5]]
True

Other special matrices: Symmetric, Triangular, Diagonal

- Symmetric matrices are symmetric around the diagonal; formally, \( A = A^T \)
- Triangular matrices only have non-zeros in the upper or lower triangular part of the matrix
- Diagonal matrices only have non-zeros along the diagonal of a matrix
### In [25]:

```python
A = np.arange(25).reshape(5, 5)+1
print('Symmetric')
print(A + A.T)
print('Upper triangular')
print(np.triu(A))
print('Lower triangular')
print(np.tril(A))
print('Diagonal (both upper and lower triangular)')
print(np.diag(np.arange(5) + 1))
```

Symmetric

```
[[ 2  8 14 20 26]
 [ 8 14 20 26 32]
 [14 20 26 32 38]
 [20 26 32 38 44]
 [26 32 38 44 50]]
```

Upper triangular

```
[[ 1  2  3  4  5]
 [ 0  7  8  9 10]
 [ 0  0 13 14 15]
 [ 0  0  0 19 20]
 [ 0  0  0  0 25]]
```

Lower triangular

```
[[ 1  0  0  0  0]
 [ 6  7  0  0  0]
 [11 12 13  0  0]
 [16 17 18 19  0]
 [21 22 23 24 25]]
```

Diagonal (both upper and lower triangular)

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

---

**Multiplying a matrix by a diagonal matrix scales the columns or rows**

- Right multiplication scales rows
- Left multiplication scales columns
The inverse of a diagonal matrix is formed merely by taking the inverse of the diagonal elements. Most operations on diagonal matrices are just the scalar versions of their entries.

```python
A = np.arange(16).reshape(4, 4)
print(A)
D = np.diag(10**(np.arange(4)))
diag_vec = np.diag(D)
print(D)
print('AD')
print(np.matmul(A, D))
print('AD (via numpy * and broadcasting)')
print(A * diag_vec)
print('DA')
print(np.matmul(D, A))
print('DA (via numpy * and broadcasting)')
print((A.T * diag_vec).T)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
[12 13 14 15]]
[[  1  0  0  0]
 [  0 10  0  0]
 [  0  0 100  0]
 [  0  0  0 1000]]
AD
[[  0 10  200 3000]
 [ 4  50  600 7000]
 [ 8  90 1000 11000]
[12 130 1400 15000]]
AD (via numpy * and broadcasting)
[[  0 10  200 3000]
 [ 4  50  600 7000]
 [ 8  90 1000 11000]
[12 130 1400 15000]]
DA
[[  0  1  2  3]
 [ 40  50  60  70]
 [ 800 900 1000 1100]
[12000 13000 14000 15000]]
DA (via numpy * and broadcasting)
[[  0  1  2  3]
 [ 40  50  60  70]
 [ 800 900 1000 1100]
[12000 13000 14000 15000]]
```
Motivation: Matrix decompositions allow us to understand and manipulate matrices both theoretically and practically

- Anagalous to prime factorization of an integer, e.g., \( 12 = 2 \times 2 \times 3 \)
  - Allows us to determine whether things are divisible by other integers
- Analogous to representing a signal in the time versus frequency domain
  - Both domains represent the same object but are useful for different computations and derivations

Eigendecomposition

- For real symmetric matrices, the eigendecomposition is:
  \[
  A = Q \Lambda Q^T
  \]
  where \( Q \) is an orthogonal matrix and \( \Lambda \) is a diagonal matrix.
- Often in notation, it is assumed that the diagonal of \( \Lambda \), denoted \( \lambda \) is ordered by decreasing values, i.e., \( \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \).
- \( \lambda \) are known as the eigenvalues and \( Q \) is known as the eigenvector matrix.
Simple properties based on eigendecomposition

- $A^{-1}$ is easy to compute
  - Easy to solve equation $Ax = b$
- Powers of matrix is easy to compute $A^3 = AAA$.
- The matrix is singular if and only if there is a zero in $\lambda$

Positive definite (or semidefinite) matrices have positive (or possibly 0) eigenvalues

- $A$ is positive definite (PD) if and only if $\forall x, x^T A x > 0$
- Positive semi-definite (PSD) is where there could be zero eigenvalues.
- Informally, a PD matrix is like $a > 0$ in a quadratic formula, $ax^2$
  - Scalar quadratic: $ax^2 + bx + c$
  - Vector quadratic: $x^T A x + b^T x + c$
  - $A$ is a generalization of $a$ in the scalar equation
- If not positive definite, there may be saddle points.
Singular value decomposition of any matrix (The decomposition to end all decompositions)

- For any matrix $A \in \mathbb{R}^{m \times n}$ (even non-square), the singular value decomposition is:
  \[
  A = U \Sigma V^T
  \]

In [29]:
```python
# Get random orthogonal matrix Q
rng = np.random.RandomState(0)
Q, _ = np.linalg.qr(rng.randn(2, 2))

# Create positive definite matrix
lam = np.array([1, 1])  # Positive definite
#lam = np.array([1, 1])  # Negative definite
#lam = np.array([-1, 1])  # Not positive or negative definite

# Construct a matrix from Q and lambda
A = np.matmul(np.matmul(Q, np.diag(lam)), Q.T)

# Plot 3D
from mpl_toolkits.mplot3d import Axes3D
v = np.linspace(-10, 10, num=20)
xx, yy = np.meshgrid(v, v)
X = np.array([xx.ravel(), yy.ravel()]).T
f = np.sum(np.matmul(A, X.T) * X.T, axis=0)
ff = f.reshape(xx.shape)

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(xx, yy, ff, cmap='viridis')
```
Out[29]: <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x7fc77da1f1d0>
where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal matrices and $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal (though not necessarily square) matrix.

- Often in notation, it is assumed that the diagonal of $\Sigma$, denoted $\sigma$ is ordered by decreasing values, i.e., $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_d$.
- $\sigma$ are known as the singular values and $U$ and $V$ are known as the left singular vectors and the right singular vectors respectively.

$U \in \mathbb{R}^{m \times m}$ $V \in \mathbb{R}^{n \times n}$ $\Sigma \in \mathbb{R}^{m \times n}$ $\sigma \geq \sigma \geq \cdots \geq \sigma$

In [30]:

```python
rng = np.random.RandomState(0)
A = np.arange(6).reshape(2, 3)
print('A', A.shape)
print(A)

# Note returns V^T (i.e. transpose) rather than V
U, s, Vt = np.linalg.svd(A, full_matrices=True)

# Convert singular vector to matrix
Sigma = np.zeros_like(A, dtype=float)
Sigma[2:2, 2] = np.diag(s)
print('U', U.shape)
print('Sigma', Sigma.shape)
print('Vt', Vt.shape)

A_reconstructed = np.matmul(U, np.matmul(Sigma, Vt))
print('Are all entries equal up to machine precision?')
print('Yes' if np.allclose(A, A_reconstructed) else 'No')
```

A (2, 3)
[[0 1 2]
 [3 4 5]]
U (2, 2)
Sigma (2, 3)
Vt (3, 3)
Are all entries equal up to machine precision?
Yes

**Rank** $\text{rank}(A)$ is the number of linearly independent columns

- Consider an example of two equations with two unknowns (Is there a unique solution?):
  - $2x + 3y = 0$
  - $4x + 6y = 1$
- Similar to a matrix $A = \begin{bmatrix} 2 & 3 \\ 4 & 6 \end{bmatrix}$, notice "redundancy"
- SVD -> Rank = Number of non-zero singular values
- If $A \in \mathbb{R}^{d \times d}$, $A$ is not singular if and only if $\text{rank}(A) = d$.
- Simplest case is rank 1 matrix: $xy^T$ (show on board)
  - Notice difference from inner product, denoted as $x^T y$
  - $xy^T$ is also known as the outer product of two vectors
Matrix multiplication can be seen as a sum of rank 1 matrices

\[ AB = \sum_{i=1}^{d} A_{i,:} B_{i,:} \]  
where \( A_{i,:} \) is the \( i \)-th column of \( A \) and \( B_{i,:} \) is the \( i \)-th row of \( B \)

In [31]:

```python
A = np.arange(6).reshape(2, 3)
print(A)
B = -np.arange(6).reshape(3, 2)
print(B)

AB_sum = np.zeros((2, 2))
for acol, brow in zip(A.T, B):
    AB_sum += np.outer(acol, brow)

print('AB sum formula')
print(AB_sum)

print('AB standard')
AB = np.matmul(A, B)
print(AB)
```

```
[[ 0  1  2]
 [ 3  4  5]]
[[ 0 -1]
 [-2 -3]
[-4 -5]]
AB sum formula
```
```
[[ -10. -13.]
 [-28. -40.]]
AB standard
```
```
[[ -10 -13]
 [-28 -40]]
```

SVD provides powerful interpretation of matrix as sum of rank one matrices

\[ A = U\Sigma V^T = \sum_{i=1}^{\text{rank}(A)} \sigma_i u_i v_i^T \]

- SVD can be used to solve the following matrix approximation problem:
  \[
  \min_B \| A - B \|_F \quad \text{s.t.} \quad \text{rank}(B) \leq r
  \]
  
  where \( \| A \|_F \) is the Frobenius norm, or just like the \( \ell_2 \) norm but consider the matrix as a long vector.
  - Example:
    \[
    \| A \|_F = \left\| \begin{bmatrix} a & b \\ c & d \end{bmatrix} \right\|_F = \| [a, b, c, d] \|_2
    \]
Usually the most important information is in the first few singular values.
Determinant \( \det(A) \) (of square matrix) is the product of eigenvalues \( \lambda \)

\[
\det(A) = |A| = \prod_{i=1}^{d} \lambda_i
\]

- Absolute value of determinant roughly measures how much the matrix expands or contracts space
- Example: if determinant is 0, then compresses vectors onto a smaller subspace
- Example: if determinant is 1, then volume is preserved (how is this different than orthogonal matrix?)
Trace operation

- Trace is just the sum of the diagonal elements of a matrix
  \[ \text{Tr}(A) = \sum_{i=1}^{d} a_{i,i} \]
- Most useful property is rotational equivalence:
  \[ \text{Tr}(ABC) = \text{Tr}(CAB) = \text{Tr}(BCA) \]
- In particular, (even if different dimensions)
  \[ \text{Tr}(AB) = \text{Tr}(BA) \]