Brief Review of Linear Algebra (Part 3)

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
```

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 analagous to a 1 for matrices but more general
- A *square* matrix such that $Q^T Q = QQ^T = I$
- Or, equivalently $Q^{-1} = Q^T$
- Or, equivalently:
  - Every column (or row) is orthogonal to every other column (or row)
  - Every column (or row) has unit L^2 norm, i.e., $\|Q_{i,:}\|_2 = \|Q_{:,j}\|_2 = 1$

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

print('Reflection matrix')
Q = np.array([[1, 0], [0, -1]]) # Reflection
print(Q)
print(np.allclose(np.eye(2), np.dot(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.dot(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 [3]:
```python
A = np.arange(25).reshape(5, 5)+1
print('Symmetric')
B = np.random.RandomState(0).randint(50,size=(5, 5))
print(B + B.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
[[88 86 23  4 27]
 [86 18 25 59 53]
 [23 25 48 63 49]
 [ 4 59 63 46 71]
 [27 53 49 71 26]]
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

In [4]:
```
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.dot(A, D))
print('AD (via numpy * and broadcasting)')
print(A * diag_vec)
print('DA')
print(np.dot(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]]
```

Inverse of diagonal matrix is formed merely by taking inverse of diagonal elements

- Most operations on diagonal matrices are just the scalar versions of their entries

In [5]:
```
A = np.diag(np.arange(5)+1)
print(A)
diag_A = np.diag(A)
print('diag_A', diag_A)
diag_A_inv = 1 / diag_A
print('diag_A_inv', diag_A_inv)
Ainv = np.diag(diag_A_inv)
print(Ainv)
Ainv_full = np.linalg.inv(A)
print(Ainv_full)
```

```
[[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]]
diag_A [1 2 3 4 5]
diag_A_inv [1.         0.5        0.33333333 0.25       0.2       ]
[[1.         0.         0.         0.         0.        ]
 [0.         0.5        0.         0.         0.        ]
 [0.         0.         0.33333333 0.         0.        ]
 [0.         0.         0.         0.25       0.        ]
 [0.         0.         0.         0.         0.2       ]]
[[ 1.          0.          0.          0.          0.        ]
 [ 0.          0.5         0.          0.          0.        ]
 [ 0.          0.          0.33333333  0.          0.        ]
 [-0.         -0.         -0.          0.25       -0.        ]
 [ 0.          0.          0.          0.          0.2       ]]
```

Motivation: Matrix decompositions allow us to *understand* and *manipulate* matrices both theoretically and practically

- Analagous 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
- Analagous 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 Λ is a **diagonal** matrix.
- Often *in notation*, it is assumed that the diagonal of Λ, denoted λ is ordered by decreasing values, i.e., $\lambda_1 \geq \lambda_2, \geq \cdots \geq \lambda_d$.
- λ are known as the **eigenvalues** and Q is known as the **eigenvector matrix**

In [6]:
```
rng = np.random.RandomState(0)
B = rng.randn(4,4)
A = B + B.T # Make symmetric
lam, Q = np.linalg.eig(A)
print(np.diag(lam))
print(Q)
A_reconstructed = np.dot(np.dot(Q, np.diag(lam)), Q.T)
print('Are all entries equal up to machine precision?')
print('Yes' if np.allclose(A, A_reconstructed) else 'No')
```

```
[[ 6.54930093  0.          0.          0.        ]
 [ 0.         -3.728219    0.          0.        ]
 [ 0.          0.          0.45077461  0.        ]
 [ 0.          0.          0.         -0.7428718 ]]
[[ 0.77115168  0.36010163  0.51908231 -0.07877468]
 [ 0.25392564 -0.75129904  0.0518548  -0.60694531]
 [ 0.31251286  0.37021589 -0.78092889 -0.394241  ]
 [ 0.49313545 -0.41087317 -0.34353267  0.68555523]]
Are all entries equal up to machine precision?
Yes
```

Simple properties based on eigendecomposition

- A^{-1} is easy to compute (derive on board)
  - Easy to solve equation $A\mathbf{x} = \mathbf{b}$ (derive on board)
- Powers of matrix is easy to compute $A^3 = AAA$. (derive on board)
- The matrix is singular if and only if there is a zero in λ

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

- A is positive definite (PD) if and only if $\forall \mathbf{x}, \mathbf{x}^T A \mathbf{x} > 0$
- 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: $\mathbf{x}^T A \mathbf{x} + \mathbf{b}^T \mathbf{x} + c$
  - A is a generalization of a in the scalar equation
- If not positive definite, there may be saddle points.

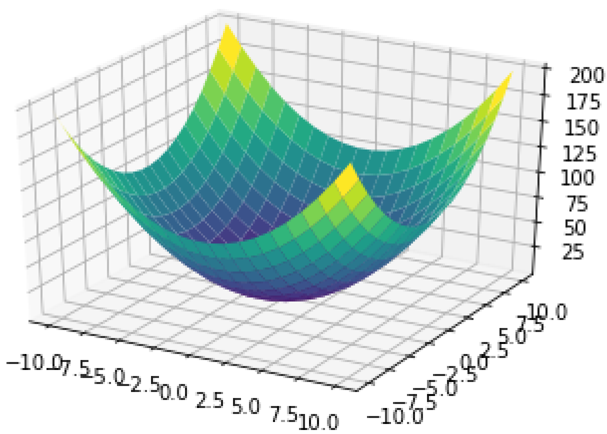
In [7]:
```
# Get random orthogonal matrix Q
rng = np.random.RandomState(0)
Q, _ = np.linalg.qr(rng.randn(2, 2))
# Create positive definite matrix
lam = np.array([1, 1])  # Positive definite
#lam = np.array([1, 1])  # Negative definite
#lam = np.array([-1, 1])  # Not positive or negative definite

# Construct a matrix from Q and lambda
A = np.dot(np.dot(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.dot(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[7]: <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x115b231d0>



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:
$$A = U \Sigma V^T$$
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 Σ, denoted σ is ordered by decreasing values, i.e., $\sigma_1 \geq \sigma_2, \geq \cdots \geq \sigma_d$.
- σ are known as the **singular values** and U and V are known as the **left singular vectors** and the **right singular vectors** respectively.

In [8]:
```
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] = np.diag(s)

print('U', U.shape)
print('Sigma', Sigma.shape)
print('Vt', Vt.shape)

A_reconstructed = np.dot(U, np.dot(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: $\mathbf{x}\mathbf{y}^T$ (show on board)
  - **Notice difference from inner product, denoted as $\mathbf{x}^T\mathbf{y}$**
  - $\mathbf{x}\mathbf{y}^T$ 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 [9]:
```
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.dot(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

$$A = U\Sigma V^T = \sum_{i=1}^{\text{rank}(A)} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

- SVD can be used to solve the following matrix approximation problem:

$$\min_B \|A - B\|_F \quad \text{s.t.} \quad \text{rank}(B) \le r$$

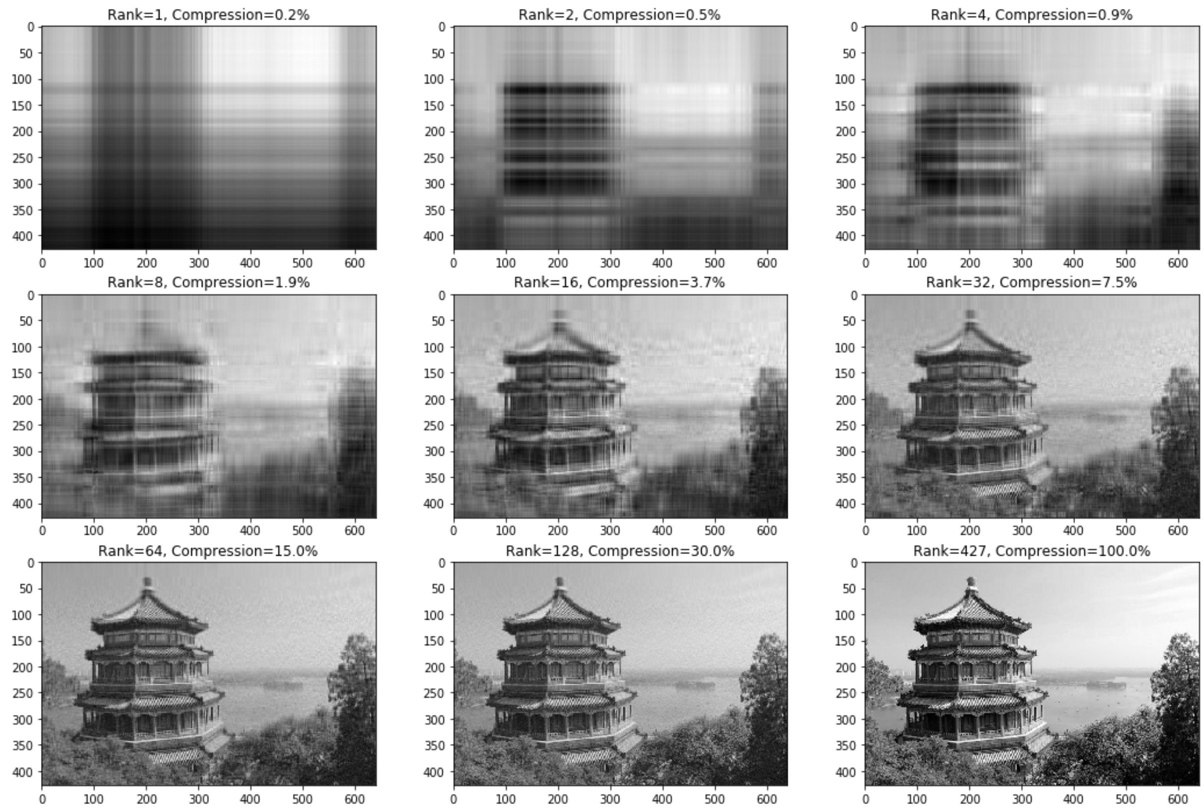
where $\|A\|_F$ is the Frobenius norm, or just like the L^2 norm but consider the matrix as a long vector.

In [10]:
```
from sklearn.datasets import load_sample_image
china = load_sample_image('china.jpg')
gray_china = china[:,:,0]/255.0
print('china matrix', gray_china.shape)
#print(gray_china)

U, s, Vt = np.linalg.svd(gray_china)
Sigma = np.zeros_like(gray_china, dtype=float)
Sigma[:427, :427] = np.diag(s)
```

```
china matrix (427, 640)
```

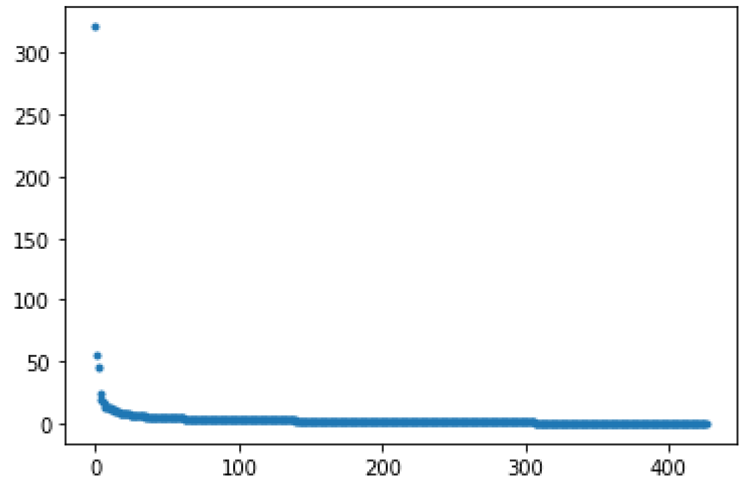
In [11]:
```
max_rank = np.min(gray_china.shape)
rank_arr = [1, 2, 4, 8, 16, 32, 64, 128, max_rank]
fig, axes = plt.subplots(3, 3, figsize=(len(rank_arr)*2, 3*4))
for r, ax in zip(rank_arr, axes.ravel()):
    china_approx = np.dot(U[:, :r], np.dot(Sigma[:r,:r], Vt[:r, :]))
    compression = r/max_rank
    ax.imshow(china_approx, cmap='gray')
    ax.set_title('Rank=%d, Compression=%.1f%%' % (r, compression*100))
```



Usually the most important information is in the first few singular values

In [12]:
```
# The most important components are
plt.plot(s,'.')
```

Out[12]: [<matplotlib.lines.Line2D at 0x1a1945d630>]



Determinant $\det(A)$ **(of square matrix) is the product of eigenvalues** λ

$$\det(A) = \prod_{i=1}^{d} \sigma_i$$

- 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 $\text{Tr}(A)$ **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)

$$\text{Tr}(AB) = \text{Tr}(BA)$$