13. Linear Algebra (NumPy) – Practical Focus

1. Topic Overview

What this topic is

Linear Algebra in NumPy means working with vectors and matrices.
It includes operations like dot product, matrix multiplication, inverse, and eigenvalues.

Why it exists

All Machine Learning models are math models.
Almost every model is written using linear algebra.

One real-world analogy

Think of matrices as machines.
You put numbers in.
The machine transforms them.
You get new numbers out.

2. Core Theory (Deep but Clear)

NumPy stores linear algebra objects as **ndarrays**.

Key internal ideas:

- **Shape** defines how math works
- **dtype** affects precision and speed
- **Memory layout** affects performance

We will cover these sub-topics:

1. Vectors and matrices
2. Dot product
3. Matrix multiplication
4. Transpose
5. Determinant
6. Inverse
7. Rank
8. Eigenvalues and eigenvectors
9. Solving linear equations

2.1 Vectors and Matrices

- Vector = 1D array
- Matrix = 2D array

Internally:

- Vector shape: `(n,)`
- Matrix shape: `(rows, cols)`

NumPy treats both as `ndarray`.

2.2 Dot Product

Dot product combines two vectors.
Result is a single number.

Internally:

- Shapes must match
- Computation uses optimized C loops

2.3 Matrix Multiplication

Matrix multiplication is not element-wise.
Rows of first matrix interact with columns of second.

Rule:

```
(A shape: m × n) · (B shape: n × p) → (m × p)
```

2.4 Transpose

Transpose swaps rows and columns.

Internally:

- No data copy
- Just a new view with changed strides

2.5 Determinant

Determinant tells:

- If matrix is invertible
- If data is linearly independent

Only for square matrices.

2.6 Inverse

Inverse undoes a matrix operation.

Condition:

- Determinant ≠ 0

Inverse is expensive and unstable for large matrices.

2.7 Rank

Rank = number of independent rows or columns.

Used to detect:

- Redundant features
- Multicollinearity

2.8 Eigenvalues and Eigenvectors

Eigenvectors do not change direction after transformation.
Eigenvalues tell how much they stretch.

Used in:

- PCA
- Dimensionality reduction

2.9 Solving Linear Equations

Equation form:

```
Ax = b
```

NumPy solves this without computing inverse directly.

3. Syntax & Examples

3.1 Vectors and Matrices

```
import numpy as np

v = np.array([1, 2, 3])
m = np.array([[1, 2],
              [3, 4]])
print(v)
print(m)
```

Output:

```
[1 2 3]
[[1 2]
 [3 4]]
```

Explanation:

- `v` is 1D vector
- `m` is 2D matrix

3.2 Dot Product

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

result = np.dot(a, b)
print(result)
```

Output:

```
32
```

Explanation:

```
1*4 + 2*5 + 3*6 = 32
```

3.3 Matrix Multiplication

```
A = np.array([[1, 2],
              [3, 4]])
B = np.array([[5, 6],
              [7, 8]])

C = A @ B
print(C)
```

Output:

```
[[19 22]
 [43 50]]
```

Explanation:

- `@` is matrix multiplication
- Shape logic is enforced

3.4 Transpose

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])

print(A.T)
```

Output:

```
[[1 4]
 [2 5]
 [3 6]]
```

Explanation:

- Rows become columns
- No data copy

3.5 Determinant

```
A = np.array([[1, 2],
              [3, 4]])

det = np.linalg.det(A)
print(det)
```

Output:

```
-2.0000000000000004
```

Explanation:

- Small float error is normal
- Determinant ≠ 0 → invertible

3.6 Inverse

```
A = np.array([[1, 2],
              [3, 4]])

inv_A = np.linalg.inv(A)
print(inv_A)
```

Output:

```
[[-2.   1. ]
 [ 1.5 -0.5]]
```

Explanation:

- Inverse exists because det ≠ 0

3.7 Rank

```
A = np.array([[1, 2],
              [2, 4]])

rank = np.linalg.matrix_rank(A)
print(rank)
```

Output:

```
1
```

Explanation:

- Second row is dependent
- Rank < number of rows

3.8 Eigenvalues and Eigenvectors

```
A = np.array([[2, 0],
              [0, 3]])

values, vectors = np.linalg.eig(A)
print(values)
print(vectors)
```

Output:

```
[2. 3.]
[[1. 0.]
 [0. 1.]]
```

Explanation:

- Diagonal matrix → simple eigenvectors

3.9 Solving Linear Equations

```
A = np.array([[3, 1],
              [1, 2]])
b = np.array([9, 8])

x = np.linalg.solve(A, b)
print(x)
```

Output:

```
[2. 3.]
```

Explanation:

- Solves `Ax = b`
- Faster and safer than inverse

4. Why This Matters in Data Science

Data Cleaning

- Detect redundant columns using rank

Feature Engineering

- Feature transformation uses matrix multiplication

Model Input Preparation

- Models expect `(n_samples, n_features)` matrices

ML / DL Pipelines

- Linear regression
- PCA
- Neural network layers

What breaks if you don’t know this

- Shape mismatch errors
- Wrong model outputs
- Slow and unstable training

5. Common Mistakes (VERY IMPORTANT)

1. Confusing `*` with `@`
    - `*` is element-wise
    - Use `@` for matrix math
2. Ignoring shapes
    - Wrong shape = runtime error
3. Using inverse everywhere
    - Causes numerical instability
4. Treating 1D arrays like matrices
    - `(n,)` is not `(n,1)`
5. Ignoring float precision errors
    - Determinant may not be exactly zero

6. Performance & Best Practices

- Fast when:
    - Using vectorized operations
    - Using `np.linalg.solve`
- Slow when:
    - Computing inverse repeatedly
    - Large dense matrices

Warnings:

- Always check shape
- Avoid inverse in ML pipelines
- Prefer float64 for stability

7. 20 Practice Problems

Easy (5)

1. Create a 3×3 matrix and print its transpose
2. Compute dot product of two vectors
3. Find determinant of a 2×2 matrix
4. Check rank of a matrix with duplicate rows
5. Multiply two compatible matrices

Medium (7)

6. Normalize feature matrix using matrix math
7. Detect redundant features using rank
8. Solve a linear system from real data
9. Compare `@` vs `*` on matrices
10. Convert 1D vector to column matrix
11. Compute covariance matrix
12. Validate matrix invertibility

Hard (5)

13. Implement linear regression using normal equation
14. Show instability using inverse
15. Perform PCA using eigen decomposition
16. Analyze multicollinearity
17. Optimize matrix operations for speed

Industry-Level Tasks (3)

18. Build PCA from scratch using NumPy

19. Detect feature redundancy in dataset
20. Prepare matrix pipeline for ML model

8. Mini Checklist

- Vector ≠ matrix
- Shapes control everything
- Use `@` for matrix multiplication
- Avoid inverse in ML
- Prefer `np.linalg.solve`
- Rank reveals redundancy
- Eigenvalues power PCA

13A. Advanced Linear Algebra (NumPy) – Mandatory Add-ons

1. Topic Overview

What this topic is

These are advanced linear algebra operations used behind real ML systems.
They handle numerical stability, optimization, and high-dimensional data.

Why it exists

Basic matrix math is not enough for real data.
Real data is noisy, large, and unstable.

One real-world analogy

Basic math is walking.
These topics are driving a truck with load.
You need control and stability.

2. Core Theory (Deep but Clear)

Additional sub-topics covered here:

1. Vector norms
2. Distance metrics
3. Orthogonality
4. Projection
5. Singular Value Decomposition (SVD)
6. Pseudo-inverse
7. Condition number
8. Numerical stability concepts

2.1 Vector Norms

Norm = length or size of a vector.

Common norms:

- L1 norm
- L2 norm
- Infinity norm

Internally:

- Computed from vector elements
- Uses float math
- Sensitive to scale

2.2 Distance Metrics

Distance measures similarity between data points.

Common distances:

- Euclidean
- Manhattan
- Cosine distance

Used heavily in ML.

2.3 Orthogonality

Two vectors are orthogonal if their dot product is zero.

Properties:

- No correlation
- Independent directions

Critical for PCA.

2.4 Projection

Projection puts one vector onto another.

Used to:

- Reduce dimensions
- Remove components

Math depends on dot product and norms.

2.5 Singular Value Decomposition (SVD)

SVD decomposes a matrix into three matrices:

```
A = U Σ Vᵀ
```

Works on:

- Any matrix
- Even non-square

Internally:

- Very expensive
- Very stable

2.6 Pseudo-Inverse

Used when inverse does not exist.

Defined as:

```
A⁺ = V Σ⁺ Uᵀ
```

Computed using SVD.

2.7 Condition Number

Measures numerical stability of a matrix.

High value means:

- Small errors cause large output changes

Important for optimization.

2.8 Numerical Stability Concepts

Floating point math is not exact.

Problems:

- Overflow
- Underflow
- Precision loss

Linear algebra must handle this carefully.

3. Syntax & Examples

3.1 Vector Norms

```
import numpy as np

v = np.array([3, 4])

print(np.linalg.norm(v, 1))
print(np.linalg.norm(v, 2))
print(np.linalg.norm(v, np.inf))
```

Output:

```
7.0
5.0
4.0
```

Explanation:

- L1 = |3| + |4|
- L2 = sqrt(3^2 + 4^2)
- Inf = max value

3.2 Distance Metrics

```
a = np.array([1, 2])
b = np.array([4, 6])

dist = np.linalg.norm(a - b)
print(dist)
```

Output:

```
5.0
```

Explanation:

- Euclidean distance
- Difference vector first

3.3 Orthogonality

```
a = np.array([1, 0])
b = np.array([0, 1])

print(np.dot(a, b))
```

Output:

```
0
```

Explanation:

- Dot product zero
- Vectors are orthogonal

3.4 Projection

```
a = np.array([2, 2])
b = np.array([1, 0])

proj = (np.dot(a, b) / np.dot(b, b)) * b
print(proj)
```

Output:

```
[2. 0.]
```

Explanation:

- Projection of `a` onto `b`

3.5 Singular Value Decomposition

```
A = np.array([[1, 2],
              [3, 4],
              [5, 6]])

U, S, Vt = np.linalg.svd(A)

print(U)
print(S)
print(Vt)
```

Explanation:

- `S` contains singular values
- Used for dimensionality reduction

3.6 Pseudo-Inverse

```
A = np.array([[1, 2],
              [2, 4]])

pinv = np.linalg.pinv(A)
print(pinv)
```

Explanation:

- Works even when inverse fails

3.7 Condition Number

```
A = np.array([[1, 2],
              [2, 4.0001]])

cond = np.linalg.cond(A)
print(cond)
```

Explanation:

- Large value → unstable matrix

3.8 Numerical Stability Example

```
A = np.array([[1e-10, 1],
              [1, 1]])

print(np.linalg.det(A))
```

Explanation:

- Determinant sensitive to float precision

4. Why This Matters in Data Science

Data Cleaning

- Norms for scaling
- Distance for outlier detection

Feature Engineering

- Projection for feature removal
- Orthogonality for decorrelation

Model Input Preparation

- SVD for dimensionality reduction
- Pseudo-inverse for regression

ML / DL Pipelines

- PCA
- Recommender systems
- Clustering
- Optimization algorithms

What breaks if you don't understand this

- Unstable models
- Wrong similarity measures
- Poor convergence
- Silent numerical bugs

5. Common Mistakes (VERY IMPORTANT)

1. Ignoring data scale before distance
2. Using Euclidean distance blindly
3. Thinking inverse always exists
4. Misinterpreting singular values

5. Ignoring condition number warnings

6. Performance & Best Practices

- Fast:
  - Norms
  - Dot products
- Slow:
  - SVD
  - Pseudo-inverse

Best practices:

- Normalize data before distance
- Use SVD for stability
- Avoid manual inverse logic
- Monitor condition number

7. 20 Practice Problems

Easy (5)

1. Compute L1 and L2 norm of a vector
2. Check orthogonality of two vectors
3. Calculate Euclidean distance
4. Project vector onto axis
5. Normalize a vector

Medium (7)

6. Implement cosine similarity
7. Compare distances before and after scaling
8. Detect unstable matrix using condition number
9. Reduce dimensions using SVD
10. Compute pseudo-inverse manually

11. Remove correlated features
12. Validate orthogonality in PCA output

Hard (5)

13. Build PCA using SVD only
14. Show instability of inverse vs pinv
15. Implement least squares using pinv
16. Analyze singular value decay
17. Optimize projection pipeline

Industry-Level Tasks (3)

18. Build similarity engine using cosine distance
19. Stabilize regression with pseudo-inverse
20. Diagnose ML model instability using condition number

8. Mini Checklist

- Norm controls scale
- Distance controls similarity
- Orthogonality means independence
- SVD is safest decomposition
- Pseudo-inverse beats inverse
- Condition number predicts failure
- Numerical stability matters