Machine Learning Primer

The Calculus, Algebra, and Intuition Behind Every Concept

From Mathematical Foundations to Modern Architectures

Binesh Sadanandan

October 2024

Contents

Part I

Mathematical Foundations

Chapter 1

Linear Algebra for Machine Learning

1.1 Introduction

Linear algebra is the mathematical language of machine learning. Nearly every ML algorithm—from simple linear regression to deep neural networks—can be described in terms of vectors, matrices, and the transformations they represent. If calculus tells us *how to optimize*, linear algebra tells us *what we are optimizing over*.

1.1.1 Why Linear Algebra?

Machine learning operates on data. A single data point with n features is naturally represented as a vector $\boldsymbol{x} \in \mathbb{R}^n$. A dataset of m samples becomes a matrix $X \in \mathbb{R}^{m \times n}$. Model parameters, gradients, and predictions are all vectors. The operations we perform—projections, rotations, scaling, decompositions—are linear algebra.

Consider a concrete example: a house described by 3 features (area, bedrooms, age) is a vector $\boldsymbol{x} = (1500, 3, 20)^\top \in \mathbb{R}^3$. A dataset of 1000 houses is a matrix $X \in \mathbb{R}^{1000 \times 3}$. A linear model predicts price as $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + b$—a dot product. Training means finding the optimal $\boldsymbol{w}$—solving a linear system.

The Big Picture: Every ML algorithm can be understood through three lenses:

1. **Geometric:** Data points live in a high-dimensional space. Models carve decision boundaries or fit surfaces.
2. **Algebraic:** Training reduces to solving systems of equations, matrix decompositions, or eigenproblems.
3. **Statistical:** Data comes from probability distributions. Models estimate distributional parameters.

Linear algebra provides the language for all three perspectives.

1.2 Vectors and Vector Spaces

A **vector** $\boldsymbol{x} \in \mathbb{R}^n$ is an ordered list of n real numbers:

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

We call n the **dimension** of the vector. Geometrically, a vector is an arrow from the origin

to a point in n-dimensional space.

A **vector space** V over $\mathbb{R}$ is a set of vectors closed under addition and scalar multiplication. The key properties are:

- **Closure:** If $\boldsymbol{x}, \boldsymbol{y} \in V$ and $c \in \mathbb{R}$, then $\boldsymbol{x} + \boldsymbol{y} \in V$ and $c\boldsymbol{x} \in V$.
- **Zero vector:** There exists $\mathbf{0} \in V$ such that $\boldsymbol{x} + \mathbf{0} = \boldsymbol{x}$.
- **Span:** The set of all linear combinations of vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ forms a **subspace**.

ML Intuition: Why do vector spaces matter? Because the set of all possible predictions of a linear model $\{X\boldsymbol{w} : \boldsymbol{w} \in \mathbb{R}^n\}$ is a *subspace* of $\mathbb{R}^m$—specifically, the **column space** of X. Training a linear model means finding the point in this subspace closest to the true targets $\boldsymbol{y}$.

1.2.1 Vector Operations

Addition. Two vectors of the same dimension are added component-wise:

$$\boldsymbol{x} + \boldsymbol{y} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{pmatrix}$$

Scalar Multiplication. Multiplying by a scalar c scales every component:

$$c\,\boldsymbol{x} = \begin{pmatrix} c\,x_1 \\ c\,x_2 \\ \vdots \\ c\,x_n \end{pmatrix}$$

Linear Combination. Given vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ and scalars $c_1, \ldots, c_k$:

$$c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + \cdots + c_k\boldsymbol{v}_k$$

The set of *all* linear combinations of $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ is their **span**. If no vector in the set can be written as a linear combination of the others, they are **linearly independent**.

ML Intuition: The columns of a data matrix X represent features. If some feature is a perfect linear combination of others (e.g., "total rooms = bedrooms + bathrooms + others"), those columns are linearly *dependent*. This makes $X^\top X$ singular and the normal equation unsolvable—one reason we need regularization.

1.2.2 The Dot Product

The **dot product** (inner product) of two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ is:

$$\boldsymbol{x} \cdot \boldsymbol{y} = \boldsymbol{x}^\top \boldsymbol{y} = \sum_{i=1}^{n} x_i \, y_i$$

Geometrically, $\boldsymbol{x} \cdot \boldsymbol{y} = \|\boldsymbol{x}\| \, \|\boldsymbol{y}\| \cos\theta$, where θ is the angle between them.

The geometric formula reveals three key cases:

- $\boldsymbol{x} \cdot \boldsymbol{y} > 0$: Vectors point in roughly the same direction ($\theta < 90°$).
- $\boldsymbol{x} \cdot \boldsymbol{y} = 0$: Vectors are **orthogonal** (perpendicular, $\theta = 90°$).
- $\boldsymbol{x} \cdot \boldsymbol{y} < 0$: Vectors point in roughly opposite directions ($\theta > 90°$).

ML Intuition: The dot product measures *similarity*. When $\boldsymbol{x} \cdot \boldsymbol{y}$ is large and positive, the vectors point in similar directions. This is exactly what a linear model does: $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + b$ computes a weighted similarity between the weight vector $\boldsymbol{w}$ and the input $\boldsymbol{x}$. In modern NLP, **cosine similarity**—$\cos\theta = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{\|\boldsymbol{x}\| \|\boldsymbol{y}\|}$—is used to compare word embeddings.

Example 1.1: Two word embedding vectors: $\boldsymbol{v}_{\text{king}} = (2, 1, 3)^\top$ and $\boldsymbol{v}_{\text{queen}} = (2, 1.5, 2.5)^\top$. Their dot product is $2 \cdot 2 + 1 \cdot 1.5 + 3 \cdot 2.5 = 4 + 1.5 + 7.5 = 13$. Their cosine similarity is $\frac{13}{\sqrt{14} \cdot \sqrt{10.5}} \approx 0.98$, indicating these words are very similar in embedding space.

1.2.3 Orthogonal Projection

The **projection** of $\boldsymbol{y}$ onto $\boldsymbol{x}$ is:

$$\text{proj}_{\boldsymbol{x}}(\boldsymbol{y}) = \frac{\boldsymbol{y} \cdot \boldsymbol{x}}{\boldsymbol{x} \cdot \boldsymbol{x}} \, \boldsymbol{x}$$

The projection onto a subspace spanned by columns of A is:

$$\text{proj}_A(\boldsymbol{y}) = A(A^\top A)^{-1} A^\top \boldsymbol{y}$$

ML Intuition: Linear regression *is* orthogonal projection. The prediction $\hat{\boldsymbol{y}} = X\boldsymbol{w}^*$ is the projection of the true labels $\boldsymbol{y}$ onto the column space of X. The residual $\boldsymbol{y} - \hat{\boldsymbol{y}}$ is orthogonal to every column of X. This geometric view makes it clear why the method is optimal: the projection is the closest point in the subspace.

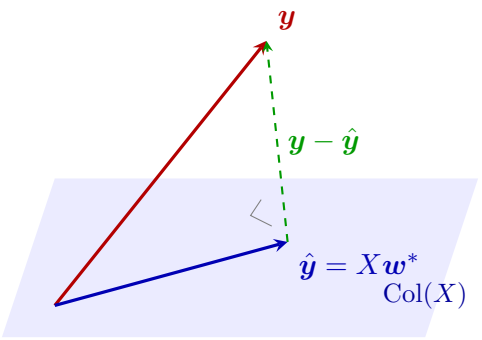


Figure 1.1: Linear regression as orthogonal projection. The prediction $\hat{\boldsymbol{y}}$ is the closest point to $\boldsymbol{y}$ in the column space of X. The residual is perpendicular to the subspace.

1.3 Norms

The L^p **norm** of a vector $\boldsymbol{x}$ is:

$$\|\boldsymbol{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}$$

Common cases:

- L^1 norm (Manhattan): $\|\boldsymbol{x}\|_1 = \sum |x_i|$ — sum of absolute values.
- L^2 norm (Euclidean): $\|\boldsymbol{x}\|_2 = \sqrt{\sum x_i^2}$ — straight-line distance.
- L^∞ norm (Max): $\|\boldsymbol{x}\|_\infty = \max_i |x_i|$ — largest component.

ML Intuition: Norms measure the "size" of a vector. In ML, we use norms for:

- **Measuring error:** $\|\boldsymbol{y} - \hat{\boldsymbol{y}}\|_2^2$ is the sum of squared errors (MSE loss).
- **Regularization:** L^1 regularization ($\|\boldsymbol{w}\|_1$) encourages **sparsity**—some weights become exactly zero. L^2 regularization ($\|\boldsymbol{w}\|_2^2$) encourages **small weights**—weights shrink but rarely hit zero.
- **Gradient clipping:** $\|\nabla \mathcal{L}\|_2$ is clipped to prevent exploding gradients in deep networks.

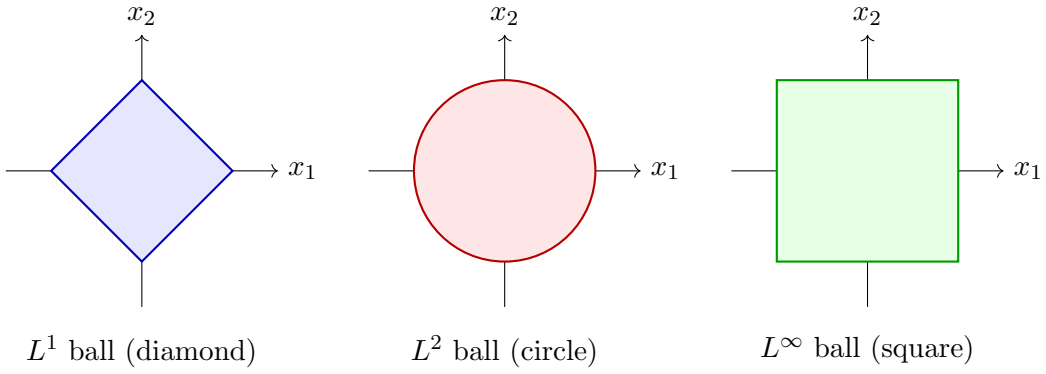


Figure 1.2: Unit balls for different norms. The shape of the constraint region determines how regularization affects weights. The L^1 diamond has corners on axes, which is why L^1 regularization

drives weights to zero.

Example 1.2: Compute the L^1, L^2, and L^∞ norms of $\boldsymbol{x} = (3, -4, 0)^\top$.

Solution:

$$\|\boldsymbol{x}\|_1 = |3| + |-4| + |0| = 7$$
$$\|\boldsymbol{x}\|_2 = \sqrt{3^2 + (-4)^2 + 0^2} = \sqrt{25} = 5$$
$$\|\boldsymbol{x}\|_\infty = \max(|3|, |-4|, |0|) = 4$$

Notice that $\|\boldsymbol{x}\|_\infty \leq \|\boldsymbol{x}\|_2 \leq \|\boldsymbol{x}\|_1$. This ordering holds for *all* vectors—each norm captures "size" differently.

1.4 Matrices and Transformations

A **matrix** $A \in \mathbb{R}^{m \times n}$ is a rectangular array of numbers with m rows and n columns. The entry in row i, column j is A_{ij}. The product $C = AB$ has entries:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Matrix multiplication is the most fundamental operation in ML computation.

Two views of matrix-vector multiplication $\boldsymbol{y} = A\boldsymbol{x}$:

1. **Row view:** Each entry y_i is the dot product of row i of A with $\boldsymbol{x}$. This is what a neural network layer does: each neuron computes a dot product with the input.
2. **Column view:** $\boldsymbol{y}$ is a linear combination of columns of A, weighted by entries of $\boldsymbol{x}$: $\boldsymbol{y} = x_1\boldsymbol{a}_1 + x_2\boldsymbol{a}_2 + \cdots + x_n\boldsymbol{a}_n$. This shows that the output lives in the column space of A.

Example 1.3: A neural network layer with 3 inputs and 2 neurons:

$$\boldsymbol{z} = W\boldsymbol{x} + \boldsymbol{b} = \begin{pmatrix} 0.5 & -0.3 & 0.8 \\ 0.1 & 0.7 & -0.2 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2.3 \\ 0.9 \end{pmatrix}$$

Verification: $z_1 = 0.5(1) + (-0.3)(2) + 0.8(3) = 0.5 - 0.6 + 2.4 = 2.3$. $z_2 = 0.1(1) + 0.7(2) + (-0.2)(3) = 0.1 + 1.4 - 0.6 = 0.9$. Each neuron computes a weighted sum of inputs—this is the dot product (row view) in action.

1.4.1 Special Matrices

Several matrix types appear constantly in ML:

- **Identity matrix** I: Has 1s on the diagonal, 0s elsewhere. Acts as the multiplicative identity: $AI = IA = A$. In regularization, λI is added to make matrices invertible.

- **Diagonal matrix** D: Only the diagonal entries are nonzero. Multiplication by D scales each row (or column) independently. Efficient to store and invert: $D_{ii}^{-1} = 1/D_{ii}$.
- **Symmetric matrix**: $A = A^\top$. All eigenvalues are real. Covariance matrices, Hessians, and kernel matrices are always symmetric.
- **Orthogonal matrix**: $Q^\top Q = QQ^\top = I$. Preserves lengths ($\|Q\boldsymbol{x}\| = \|\boldsymbol{x}\|$) and angles. Rotations and reflections are orthogonal transformations. Appears in SVD and PCA.

1.4.2 The Transpose and its Properties

The transpose $A^\top$ flips a matrix across its diagonal: $(A^\top)_{ij} = A_{ji}$. Key properties:

- $(AB)^\top = B^\top A^\top$ — the order reverses.
- $\boldsymbol{x}^\top \boldsymbol{y} = \boldsymbol{y}^\top \boldsymbol{x}$ — the dot product is commutative.
- $X^\top X$ is always symmetric and positive semi-definite.

ML Intuition: The quantity $X^\top X$ (called the **Gram matrix**) appears everywhere:
- **Normal equation:** $\boldsymbol{w}^* = (X^\top X)^{-1} X^\top \boldsymbol{y}$.
- **Covariance matrix:** $\Sigma = \frac{1}{m} X^\top X$ (after centering).
- **PCA:** Eigendecompose $X^\top X$ to find principal components.

Understanding $X^\top X$ unlocks half of classical machine learning.

1.5 Systems of Linear Equations

The system $A\boldsymbol{x} = \boldsymbol{b}$ has:
- A **unique solution** $\boldsymbol{x} = A^{-1}\boldsymbol{b}$ if A is square and invertible ($\det(A) \neq 0$).
- **No solution** if $\boldsymbol{b}$ is not in the column space of A (overdetermined, inconsistent).
- **Infinitely many solutions** if A has more columns than rank (underdetermined).

ML Intuition: In linear regression with $m > n$ (more data points than features), the system $X\boldsymbol{w} = \boldsymbol{y}$ is overdetermined and generally has no exact solution. The normal equation finds the **least-squares** solution—the $\boldsymbol{w}$ that minimizes $\|X\boldsymbol{w} - \boldsymbol{y}\|^2$. When $m < n$ (more features than samples), there are infinitely many solutions, and regularization picks the "best" one.

Example 1.4: Solve $\begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \end{pmatrix}$.

Solution: First compute $\det(A) = 2 \cdot 3 - 1 \cdot 1 = 5$. Since $\det \neq 0$, A is invertible.

$$A^{-1} = \frac{1}{5} \begin{pmatrix} 3 & -1 \\ -1 & 2 \end{pmatrix}, \quad \boldsymbol{x} = A^{-1}\boldsymbol{b} = \frac{1}{5} \begin{pmatrix} 3 \cdot 5 - 1 \cdot 7 \\ -1 \cdot 5 + 2 \cdot 7 \end{pmatrix} = \frac{1}{5} \begin{pmatrix} 8 \\ 9 \end{pmatrix} = \begin{pmatrix} 1.6 \\ 1.8 \end{pmatrix}$$

Verification: $2(1.6) + 1(1.8) = 5.0$. $1(1.6) + 3(1.8) = 7.0$. ✓

1.6 Eigenvalues and Eigenvectors

An **eigenvector** of a square matrix A is a nonzero vector $\boldsymbol{v}$ such that:

$$A\boldsymbol{v} = \lambda\boldsymbol{v}$$

The scalar λ is the corresponding **eigenvalue**. To find eigenvalues, solve the **characteristic equation**:

$$\det(A - \lambda I) = 0$$

What does this mean geometrically? When we multiply most vectors by A, they get both scaled and rotated. But eigenvectors are special: they only get **scaled**. The eigenvalue λ tells you the scaling factor. If $\lambda > 1$, the eigenvector gets stretched. If $0 < \lambda < 1$, it gets compressed. If $\lambda < 0$, it gets flipped.

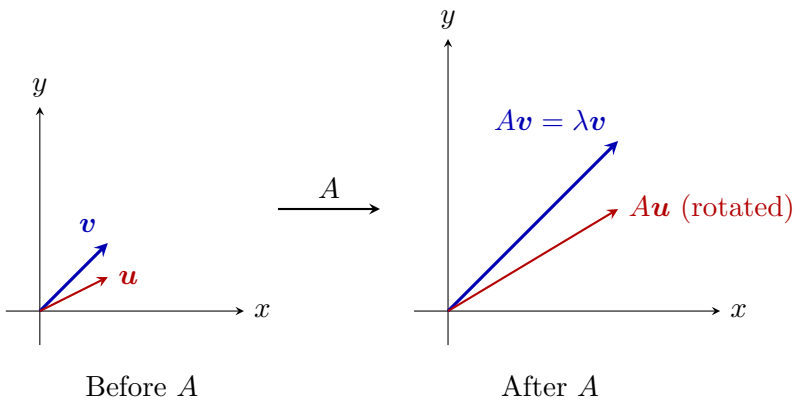


Figure 1.3: Eigenvectors (blue) only get scaled by A, so their direction stays unchanged. Non-eigenvectors (red) get scaled and rotated.

Example 1.5: Find the eigenvalues and eigenvectors of $A = \begin{pmatrix} 3 & 1 \\ 0 & 2 \end{pmatrix}$.

Solution: Characteristic equation:

$$\det(A - \lambda I) = (3 - \lambda)(2 - \lambda) = 0 \implies \lambda_1 = 3, \quad \lambda_2 = 2$$

For $\lambda_1 = 3$: $(A - 3I)\boldsymbol{v} = \boldsymbol{0}$ gives $\begin{pmatrix} 0 & 1 \\ 0 & -1 \end{pmatrix}\boldsymbol{v} = \boldsymbol{0}$, so $v_2 = 0$ and $\boldsymbol{v}_1 = (1, 0)^\top$.

For $\lambda_2 = 2$: $(A - 2I)\boldsymbol{v} = \boldsymbol{0}$ gives $\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}\boldsymbol{v} = \boldsymbol{0}$, so $v_1 = -v_2$ and $\boldsymbol{v}_2 = (-1, 1)^\top$.

ML Intuition: Eigenvectors of the covariance matrix $\Sigma = \frac{1}{m}X^\top X$ point in the directions of maximum variance. The eigenvalues tell you how much variance lies along each direction. This is the heart of **PCA**:

- The first eigenvector (largest eigenvalue) = direction of greatest data spread.
- The second eigenvector = direction of greatest spread *orthogonal to the first.*

- Keeping only the top k eigenvectors reduces dimensionality while preserving the most variance.

Listing 1.1: Computing eigenvalues in NumPy

```python
import numpy as np

A = np.array([[3, 1], [0, 2]])
eigenvalues, eigenvectors = np.linalg.eig(A)
print(f"Eigenvalues: {eigenvalues}")       # [3., 2.]
print(f"Eigenvectors:\n{eigenvectors}")    # columns are eigenvectors

# Verify: A @ v = lambda * v
for i in range(len(eigenvalues)):
    v = eigenvectors[:, i]
    lam = eigenvalues[i]
    print(f"A*v = {A @ v}, lambda*v = {lam * v}")  # Should be equal
```

1.7 Singular Value Decomposition (SVD)

Every matrix $A \in \mathbb{R}^{m \times n}$ (not necessarily square!) can be decomposed as:

$$A = U \Sigma V^\top$$

where:

- $U \in \mathbb{R}^{m \times m}$ is orthogonal (columns are **left singular vectors**).
- $\Sigma \in \mathbb{R}^{m \times n}$ is diagonal with **singular values** $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$.
- $V \in \mathbb{R}^{n \times n}$ is orthogonal (columns are **right singular vectors**).

Geometric Intuition: SVD says that *any* linear transformation can be decomposed into three steps:

1. **Rotate** (by $V^\top$): Align with the "natural axes" of the transformation.
2. **Scale** (by Σ): Stretch or compress along each axis.
3. **Rotate again** (by U): Align with the output coordinate system.

ML Applications:

- **Dimensionality reduction:** Truncated SVD keeps only the top k singular values, giving the best rank-k approximation (Eckart-Young theorem).
- **Pseudoinverse:** $A^+ = V \Sigma^+ U^\top$ gives the least-squares solution when A is not invertible.
- **Recommender systems:** Matrix factorization (Netflix Prize) is truncated SVD applied to user-item rating matrices.

Example 1.6: The relationship between SVD and eigendecomposition: if $A = U \Sigma V^\top$, then $A^\top A = V \Sigma^\top \Sigma V^\top$ (eigendecomposition of $A^\top A$) and $A A^\top = U \Sigma \Sigma^\top U^\top$ (eigendecomposition of $A A^\top$). The singular values of A are the square roots of the eigenvalues of $A^\top A$.

1.8 Positive Definite Matrices

A symmetric matrix A is:

- **Positive definite (PD):** $\boldsymbol{x}^\top A\boldsymbol{x} > 0$ for all nonzero $\boldsymbol{x}$. Equivalently, all eigenvalues > 0.
- **Positive semi-definite (PSD):** $\boldsymbol{x}^\top A\boldsymbol{x} \geq 0$ for all $\boldsymbol{x}$. Equivalently, all eigenvalues ≥ 0.

ML Intuition:

- If the **Hessian** H of a loss function is positive definite at a critical point, that point is a **local minimum**. If H has both positive and negative eigenvalues, the point is a **saddle point**.
- **Covariance matrices** are always PSD. The eigenvalues represent variances along principal directions.
- **Convexity:** A function f is convex if and only if its Hessian is PSD everywhere. Convex functions have no local minima that aren't global—this is why convex optimization is so desirable.

1.9 Worked Problems

Problem 1.1: Given $X = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$ and $\boldsymbol{y} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$, compute $X^\top X$, $X^\top \boldsymbol{y}$, and the least-squares solution $\boldsymbol{w}^*$.

Solution:

$$X^\top X = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} = \begin{pmatrix} 35 & 44 \\ 44 & 56 \end{pmatrix}$$

$$X^\top \boldsymbol{y} = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 22 \\ 28 \end{pmatrix}$$

Now solve $(X^\top X)\boldsymbol{w} = X^\top \boldsymbol{y}$: $\det(X^\top X) = 35 \cdot 56 - 44^2 = 1960 - 1936 = 24$.

$$\boldsymbol{w}^* = \frac{1}{24} \begin{pmatrix} 56 & -44 \\ -44 & 35 \end{pmatrix} \begin{pmatrix} 22 \\ 28 \end{pmatrix} = \frac{1}{24} \begin{pmatrix} 56 \cdot 22 - 44 \cdot 28 \\ -44 \cdot 22 + 35 \cdot 28 \end{pmatrix} = \frac{1}{24} \begin{pmatrix} 0 \\ 12 \end{pmatrix} = \begin{pmatrix} 0 \\ 0.5 \end{pmatrix}$$

Verification: $X\boldsymbol{w}^* = (1, 2, 3)^\top$. This matches $\boldsymbol{y}$ exactly, so the data lies on a perfect line.

Problem 1.2: Show that $A^\top A$ is always positive semi-definite.

Solution: For any $\boldsymbol{x}$:

$$\boldsymbol{x}^\top (A^\top A)\boldsymbol{x} = (A\boldsymbol{x})^\top (A\boldsymbol{x}) = \|A\boldsymbol{x}\|_2^2 \geq 0 \qquad \square$$

The quantity $\|A\boldsymbol{x}\|^2$ is a squared norm, which is always non-negative. It equals zero only if $A\boldsymbol{x} = \mathbf{0}$, which means $A^\top A$ is strictly positive definite if A has full column rank (no vector gets sent to zero).

Problem 1.3: In NumPy, verify the relationship between SVD and eigendecomposition.

Listing 1.2: SVD and eigendecomposition relationship

```python
import numpy as np

X = np.array([[1, 2], [3, 4], [5, 6]])
U, S, Vt = np.linalg.svd(X, full_matrices=False)
eigenvalues, eigenvectors = np.linalg.eigh(X.T @ X)

# Singular values are square roots of eigenvalues of X^T X
print(f"Singular values: {S}")
print(f"sqrt(eigenvalues): {np.sqrt(eigenvalues[::-1])}")  # Same!

# Reconstruct X from SVD
X_reconstructed = U @ np.diag(S) @ Vt
print(f"Reconstruction error: {np.linalg.norm(X - X_reconstructed):.2e}")
```

Practice Questions & Answers

Conceptual Questions

Q1.1: Why does $X^\top X$ appear so frequently in machine learning? Name at least three contexts where it shows up.

A1.1: $X^\top X$ (the Gram matrix) is central because it captures pairwise relationships between features. It appears in: (1) the **normal equation** $\boldsymbol{w}^* = (X^\top X)^{-1} X^\top \boldsymbol{y}$ for linear regression; (2) the **covariance matrix** $\Sigma = \frac{1}{m} X^\top X$ (after centering), whose eigenvectors define PCA; (3) **ridge regression** $(X^\top X + \lambda I)^{-1} X^\top \boldsymbol{y}$, where λI ensures invertibility; and (4) **kernel methods**, where $XX^\top$ is the kernel matrix measuring pairwise similarities between data points.

Q1.2: A colleague says "I don't need to worry about linear dependence in my features because I have thousands of data points." What's wrong with this reasoning?

A1.2: Linear dependence is a property of *features* (columns), not samples (rows). Having more data points doesn't fix the problem. If one feature is a perfect linear combination of others (e.g., "total area = living area + garage area"), $X^\top X$ becomes singular regardless of how many rows X has. The consequence: the normal equation has infinitely many solutions, making the model unstable. The fix is regularization (which makes $X^\top X + \lambda I$ invertible) or removing redundant features.

Q1.3: Explain geometrically why L^1 regularization produces sparse weights but L^2 does not.

A1.3: Picture the loss contours (ellipses centered at the unregularized optimum) and the constraint region (the norm ball). The solution is where they first touch. The L^1 ball is a diamond with sharp corners sitting on the coordinate axes; the L^2 ball is a smooth circle. Because ellipses are smooth curves, they're far more likely to make first contact with a corner of the diamond (where at least one coordinate equals zero) than with a smooth surface. The circle has no corners, so the tangent point almost always has all coordinates nonzero.

Multiple Choice

Q1.4: If A is a 5×3 matrix, what is the maximum possible rank of A?
(a) 5
(b) 3
(c) 15
(d) 8

Answer: (b) 3. The rank of a matrix cannot exceed $\min(m, n)$. Here $\min(5, 3) = 3$.

Q1.5: Which of the following matrices is always symmetric?
(a) X (the data matrix)
(b) $X^\top X$
(c) $XX^\top X$
(d) $X + X^\top$ only if X is square

Answer: (b) $X^\top X$. We can verify: $(X^\top X)^\top = X^\top (X^\top)^\top = X^\top X$. Option (d) is also always symmetric when X is square, but (b) works for any matrix X.

Q1.6: A matrix has eigenvalues $\{5, 3, -1\}$. Which statement is true?
(a) The matrix is positive definite
(b) The matrix is positive semi-definite
(c) The matrix is indefinite
(d) The matrix is singular

Answer: (c) Indefinite. Positive definite requires *all* eigenvalues > 0. One eigenvalue is $-1 < 0$, so it fails. Indefinite means the matrix has both positive and negative eigenvalues, which corresponds to a saddle point if this were a Hessian.

Computational Problems

Q1.7: Given $\boldsymbol{u} = (1, 2, -1)^\top$ and $\boldsymbol{v} = (3, 0, 4)^\top$, compute: (a) the dot product $\boldsymbol{u} \cdot \boldsymbol{v}$, (b) the angle between them, (c) the projection of $\boldsymbol{v}$ onto $\boldsymbol{u}$.

Solution:

$$\text{(a)} \quad \boldsymbol{u} \cdot \boldsymbol{v} = 1(3) + 2(0) + (-1)(4) = 3 + 0 - 4 = -1$$

$$\text{(b)} \quad \|\boldsymbol{u}\| = \sqrt{1+4+1} = \sqrt{6}, \quad \|\boldsymbol{v}\| = \sqrt{9+0+16} = 5$$

$$\cos\theta = \frac{-1}{\sqrt{6} \cdot 5} \approx -0.0816, \quad \theta \approx 94.7^\circ \quad \text{(nearly perpendicular)}$$

$$\text{(c)} \quad \text{proj}_{\boldsymbol{u}}(\boldsymbol{v}) = \frac{\boldsymbol{v} \cdot \boldsymbol{u}}{\boldsymbol{u} \cdot \boldsymbol{u}} \boldsymbol{u} = \frac{-1}{6}(1, 2, -1)^\top = (-\tfrac{1}{6}, -\tfrac{1}{3}, \tfrac{1}{6})^\top$$

Q1.8: Find the eigenvalues of $A = \begin{pmatrix} 4 & 2 \\ 1 & 3 \end{pmatrix}$ and determine if A is positive definite.

Solution: $\det(A - \lambda I) = (4-\lambda)(3-\lambda) - 2 = \lambda^2 - 7\lambda + 10 = (\lambda - 5)(\lambda - 2) = 0$. So $\lambda_1 = 5$, $\lambda_2 = 2$. Both are positive, but A is not symmetric ($A \neq A^\top$), so positive definiteness is only defined for symmetric matrices. However, $\frac{1}{2}(A + A^\top) = \begin{pmatrix} 4 & 1.5 \\ 1.5 & 3 \end{pmatrix}$ has eigenvalues 5.25 and 1.75, both positive, so the symmetrized form is PD.

Q1.9: Write NumPy code to verify that $\|A\boldsymbol{x}\|_2 \leq \|A\|_2 \cdot \|\boldsymbol{x}\|_2$ for a random matrix and vector.

Listing 1.3: Verifying the matrix norm inequality

```python
import numpy as np

A = np.random.randn(4, 3)
x = np.random.randn(3)

lhs = np.linalg.norm(A @ x)           # ||Ax||
rhs = np.linalg.norm(A, 2) * np.linalg.norm(x)  # ||A|| * ||x||
print(f"||Ax|| = {lhs:.4f}")
print(f"||A|| * ||x|| = {rhs:.4f}")
print(f"Inequality holds: {lhs <= rhs + 1e-10}")  # True
```

Interview-Style Questions

Q1.10: *"Explain eigenvectors to someone who has never taken a linear algebra course. Why should they care in the context of data science?"*

Model Answer: Imagine you have a dataset, like a cloud of data points in space. Some directions in this cloud are "important" because the data spreads out a lot along them, while other directions are less important because the data is tightly clustered. Eigenvectors point in these special directions, and eigenvalues tell you how "spread out" the data is along each one. In data science, this matters because you can reduce a dataset with 1000 features to maybe 50 features by keeping only the directions with the most spread (highest eigenvalues). This is called PCA. You lose very little information but gain massive speedups and often better model performance because you've removed the noise.

Q1.11: *"What happens to linear regression when two features are perfectly correlated? How would you detect and fix this?"*

Model Answer: When two features are perfectly correlated (e.g., temperature in Celsius and Fahrenheit), the columns of X are linearly dependent, making $X^\top X$ singular (determinant = 0). The normal equation $(X^\top X)^{-1}X^\top \boldsymbol{y}$ has no unique solution because there are infinitely many weight combinations that produce the same predictions. You can detect this by computing the condition number of $X^\top X$ (very large = nearly singular) or by looking at the Variance Inflation Factor (VIF). Fixes include: (1) dropping one of the correlated features, (2) adding L^2 regularization, which converts $(X^\top X)^{-1}$ to $(X^\top X + \lambda I)^{-1}$, always invertible, or (3) using PCA to decorrelate features before fitting.

Q1.12: *"When would you use SVD instead of eigendecomposition?"*

Model Answer: Eigendecomposition only applies to square matrices, but most data matrices are rectangular (m samples $\times$ n features, with $m \neq n$). SVD works on any matrix. Use SVD for: (1) dimensionality reduction on the data matrix directly (truncated SVD); (2) computing the pseudoinverse when $X^\top X$ is singular; (3) recommender systems (matrix factorization of the user-item matrix); (4) latent semantic analysis in NLP. Use eigendecomposition when you already have a square symmetric matrix like the covariance matrix $X^\top X$ or the Hessian. In practice, many libraries internally use SVD even when you call an eigenvalue solver on a symmetric matrix, because SVD is numerically more stable.

Key Takeaways

- Vectors represent data points; matrices represent datasets and transformations. ML is written in the language of linear algebra.
- The dot product measures similarity—it's the core operation of linear models: $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + b$.
- Norms measure vector size; L^1 and L^2 norms drive regularization (sparsity vs. shrinkage).
- Linear regression is orthogonal projection of $\boldsymbol{y}$ onto the column space of X.
- Eigenvalues/eigenvectors reveal the principal directions of data (PCA) and curvature of loss surfaces (Hessian).
- SVD generalizes eigendecomposition to any matrix and powers dimensionality reduction and recommender systems.
- Positive definiteness guarantees convexity—the key to well-behaved optimization with unique global minima.

Chapter 2

Multivariable Calculus for Machine Learning

2.1 Introduction

Machine learning is fundamentally an **optimization problem**: find the parameters that minimize a loss function. Calculus—specifically *multivariable* calculus—gives us the tools to navigate loss landscapes, compute gradients, and understand why optimization algorithms work.

Think of it this way: a loss function $\mathcal{L}(\boldsymbol{\theta})$ is a surface in high-dimensional parameter space. Each point on this surface represents a particular set of model parameters and the corresponding loss. Training means finding the lowest point. Calculus gives us the slope at every point (the gradient), and we use that slope to walk downhill.

2.2 Derivatives and Partial Derivatives

2.2.1 Review: Single-Variable Derivative

The derivative $f'(x) = \frac{df}{dx}$ measures the rate of change of f with respect to x. Key rules:

- Power rule: $\frac{d}{dx}x^n = nx^{n-1}$
- Chain rule: $\frac{d}{dx}f(g(x)) = f'(g(x)) \cdot g'(x)$
- Product rule: $(fg)' = f'g + fg'$
- Exponential: $\frac{d}{dx}e^x = e^x$; Logarithm: $\frac{d}{dx}\ln x = \frac{1}{x}$

2.2.2 Partial Derivatives

Given $f(x_1, x_2, \ldots, x_n)$, the **partial derivative** with respect to x_i is:

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(\ldots, x_i + h, \ldots) - f(\ldots, x_i, \ldots)}{h}$$

We differentiate with respect to x_i while **holding all other variables constant**.

Example 2.1: Let $f(x, y) = 3x^2y + 2y^3 - 5x$.
$\frac{\partial f}{\partial x}$: Treat y as a constant. $\frac{\partial}{\partial x}(3x^2y) = 6xy$, $\frac{\partial}{\partial x}(2y^3) = 0$, $\frac{\partial}{\partial x}(-5x) = -5$.
Result: $\frac{\partial f}{\partial x} = 6xy - 5$.

$\frac{\partial f}{\partial y}$: Treat x as a constant. $\frac{\partial}{\partial y}(3x^2y) = 3x^2$, $\frac{\partial}{\partial y}(2y^3) = 6y^2$, $\frac{\partial}{\partial y}(-5x) = 0$.
Result: $\frac{\partial f}{\partial y} = 3x^2 + 6y^2$.

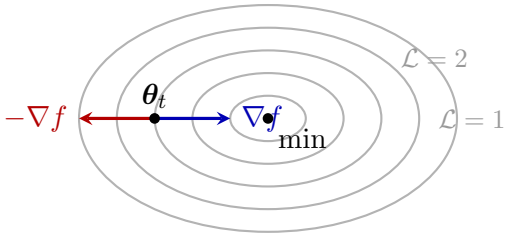
ML Intuition: If f is a loss function and x, y are model parameters, then $\frac{\partial f}{\partial x}$ tells you: "If I nudge parameter x by a tiny amount ϵ, the loss changes by approximately $\frac{\partial f}{\partial x} \cdot \epsilon$." A large partial derivative means the loss is very sensitive to that parameter—small changes have big effects. A partial derivative near zero means the parameter has little effect on the loss at that point.

2.3 The Gradient

The **gradient** of $f : \mathbb{R}^n \to \mathbb{R}$ is the vector of all partial derivatives:

$$\nabla f = \begin{pmatrix} \partial f/\partial x_1 \\ \partial f/\partial x_2 \\ \vdots \\ \partial f/\partial x_n \end{pmatrix}$$

Key property: The gradient points in the direction of **steepest ascent**. Its magnitude $\|\nabla f\|$ is the rate of steepest ascent. The negative gradient $-\nabla f$ points **downhill**—the direction that decreases f most rapidly.



Gradient descent follows $-\nabla f$ toward the minimum

Figure 2.1: Contour plot of a loss function (each ellipse = constant loss). The gradient ∇f is perpendicular to contours and points uphill. Gradient descent moves opposite to it.

The Gradient Descent Update Rule:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t)$$

where η is the **learning rate** (step size). Since $\nabla\mathcal{L}$ points uphill, $-\nabla\mathcal{L}$ points downhill toward lower loss. The learning rate controls how far we step in each iteration.

Example 2.2: Compute the gradient of MSE for a single data point: $\mathcal{L}(w, b) = \frac{1}{2}(wx+b-y)^2$.

Solution: Let $e = wx + b - y$ (the prediction error). Then $\mathcal{L} = \frac{1}{2}e^2$.

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial}{\partial w}\left[\frac{1}{2}(wx + b - y)^2\right] = (wx + b - y) \cdot x = e \cdot x$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial}{\partial b}\left[\frac{1}{2}(wx + b - y)^2\right] = (wx + b - y) \cdot 1 = e$$

$$\nabla \mathcal{L} = \begin{pmatrix} ex \\ e \end{pmatrix} = (wx + b - y)\begin{pmatrix} x \\ 1 \end{pmatrix}$$

Interpretation: The gradient is the prediction error e scaled by the input features. When the error is large, the gradient is large and the parameters change a lot. When the error is small, the gradient is small and the parameters change little. This is exactly what we want—big corrections when we're far off, small adjustments when we're close.

Example 2.3 (Numerical): $w = 0.5$, $b = 0.1$, $x = 2$, $y = 3$. Then $\hat{y} = 0.5 \cdot 2 + 0.1 = 1.1$, $e = 1.1 - 3 = -1.9$.
$\nabla \mathcal{L} = (-1.9)(2, 1)^\top = (-3.8, -1.9)^\top$.
With $\eta = 0.1$: $w_{\text{new}} = 0.5 - 0.1(-3.8) = 0.88$, $b_{\text{new}} = 0.1 - 0.1(-1.9) = 0.29$.
New prediction: $\hat{y}_{\text{new}} = 0.88(2) + 0.29 = 2.05$. Closer to 3—the gradient step improved the prediction!

2.4 The Chain Rule in Multiple Variables

If f depends on $\boldsymbol{u} = (u_1, \ldots, u_m)$, which in turn depends on $\boldsymbol{x} = (x_1, \ldots, x_n)$, the **multivariable chain rule** states:

$$\frac{\partial f}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial f}{\partial u_j} \cdot \frac{\partial u_j}{\partial x_i}$$

In matrix form: $\frac{\partial f}{\partial \boldsymbol{x}} = J^\top \frac{\partial f}{\partial \boldsymbol{u}}$, where J is the Jacobian of $\boldsymbol{u}$ with respect to $\boldsymbol{x}$.

ML Intuition: The multivariable chain rule is the mathematical foundation of **backpropagation**. A neural network is a composition of functions: $f = f_L \circ f_{L-1} \circ \cdots \circ f_1$. Each layer f_ℓ depends on the output of the previous layer.
The chain rule lets us compute how the final loss depends on parameters in *any* layer:

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(L)}} \cdot \frac{\partial \boldsymbol{a}^{(L)}}{\partial \boldsymbol{a}^{(L-1)}} \cdots \frac{\partial \boldsymbol{a}^{(2)}}{\partial \boldsymbol{a}^{(1)}} \cdot \frac{\partial \boldsymbol{a}^{(1)}}{\partial W^{(1)}}$$

Each factor is a "local" derivative that only requires information from one layer.

Example 2.4: Compute $\frac{d\mathcal{L}}{dw}$ for the chain $z = wx + b$, $a = \text{ReLU}(z)$, $\mathcal{L} = (a - y)^2$.

Solution (step by step):

$$\frac{d\mathcal{L}}{da} = 2(a-y) \qquad \text{(derivative of squared error)}$$

$$\frac{da}{dz} = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z < 0 \end{cases} \qquad \text{(ReLU derivative)}$$

$$\frac{dz}{dw} = x \qquad \text{(linear layer derivative)}$$

Chain rule: $\frac{d\mathcal{L}}{dw} = \frac{d\mathcal{L}}{da} \cdot \frac{da}{dz} \cdot \frac{dz}{dw} = 2(a-y) \cdot \mathbb{1}[z > 0] \cdot x$.

Interpretation: The gradient flows backward through the chain. Each link multiplies by its local derivative. If $z \leq 0$, the ReLU "kills" the gradient—the weight gets no update. This is the "dying ReLU" problem.

2.5 The Jacobian and Hessian

The **Jacobian** of a vector-valued function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$ is the $m \times n$ matrix of all first-order partial derivatives:

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

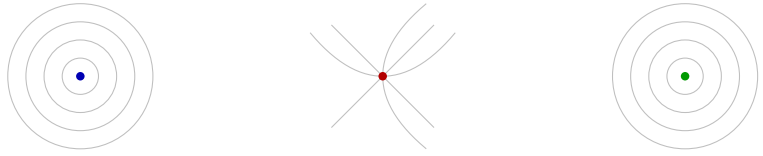
The **Hessian** of a scalar-valued function $f : \mathbb{R}^n \to \mathbb{R}$ is the $n \times n$ matrix of all second-order partial derivatives:

$$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

The Hessian is always symmetric (assuming continuous second derivatives).

ML Intuition: The Hessian captures **curvature** of the loss surface—how fast the gradient itself is changing:

- H **positive definite** at a critical point $\implies$ **local minimum**. The surface curves upward in all directions—like the bottom of a bowl.
- H has **mixed signs** $\implies$ **saddle point**. The surface curves up in some directions and down in others. Saddle points are extremely common in high-dimensional optimization.
- **Condition number** $\kappa(H) = \frac{\lambda_{\max}}{\lambda_{\min}}$ determines how "elongated" the loss landscape is. A large κ means gradient descent zigzags. A κ near 1 means smooth, fast convergence.



Local minimum ($H \succ 0$) | Saddle point (H indefinite) | Local maximum ($H \prec 0$)

Figure 2.2: Types of critical points classified by the Hessian. In high-dimensional neural networks, saddle points vastly outnumber local minima.

2.6 Matrix Calculus

Matrix calculus provides compact formulas for derivatives involving vectors and matrices. These are essential for deriving ML algorithms.

Key Matrix Derivatives:

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{a}^\top \boldsymbol{x}) = \boldsymbol{a}$$
$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{x}^\top A\boldsymbol{x}) = (A + A^\top)\boldsymbol{x} \quad \text{(if } A \text{ symmetric: } 2A\boldsymbol{x})$$
$$\frac{\partial}{\partial \boldsymbol{x}}\|\boldsymbol{x}\|_2^2 = 2\boldsymbol{x}$$
$$\frac{\partial}{\partial \boldsymbol{x}}\|A\boldsymbol{x} - \boldsymbol{b}\|_2^2 = 2A^\top(A\boldsymbol{x} - \boldsymbol{b})$$

Example 2.5 (Deriving the Normal Equation): Minimize $\mathcal{L}(\boldsymbol{w}) = \frac{1}{2}\|X\boldsymbol{w} - \boldsymbol{y}\|_2^2$.

Solution: Expand: $\mathcal{L} = \frac{1}{2}(X\boldsymbol{w} - \boldsymbol{y})^\top(X\boldsymbol{w} - \boldsymbol{y}) = \frac{1}{2}(\boldsymbol{w}^\top X^\top X\boldsymbol{w} - 2\boldsymbol{y}^\top X\boldsymbol{w} + \boldsymbol{y}^\top\boldsymbol{y})$.

Take the gradient and set to zero:

$$\nabla_{\boldsymbol{w}}\mathcal{L} = X^\top X\boldsymbol{w} - X^\top\boldsymbol{y} = \boldsymbol{0}$$

$$\boxed{\boldsymbol{w}^* = (X^\top X)^{-1}X^\top\boldsymbol{y}}$$

This is the **normal equation**—the closed-form solution for linear regression. The name comes from the fact that the residual $\boldsymbol{y} - X\boldsymbol{w}^*$ is "normal" (perpendicular) to the column space of X.

Example 2.6 (Ridge Regression): Minimize $\mathcal{L}(\boldsymbol{w}) = \frac{1}{2m}\|X\boldsymbol{w} - \boldsymbol{y}\|_2^2 + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2$.

Solution:

$$\nabla_{\boldsymbol{w}}\mathcal{L} = \frac{1}{m}X^\top(X\boldsymbol{w} - \boldsymbol{y}) + \lambda\boldsymbol{w} = \boldsymbol{0}$$
$$\left(\frac{1}{m}X^\top X + \lambda I\right)\boldsymbol{w} = \frac{1}{m}X^\top\boldsymbol{y}$$
$$\boxed{\boldsymbol{w}^* = (X^\top X + m\lambda I)^{-1}X^\top\boldsymbol{y}}$$

Key insight: The regularization term λI gets added to $X^\top X$, which does two things:

1. **Guarantees invertibility:** Even if $X^\top X$ is singular, $X^\top X + m\lambda I$ is always invertible (all eigenvalues shift by $m\lambda > 0$).
2. **Shrinks weights:** The solution $\boldsymbol{w}^*$ has smaller norm than the unregularized solution, reducing overfitting.

2.7 Taylor Expansion and Local Approximation

The second-order Taylor expansion of f around $\boldsymbol{x}_0$ is:

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}_0) + \nabla f(\boldsymbol{x}_0)^\top (\boldsymbol{x} - \boldsymbol{x}_0) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_0)^\top H(\boldsymbol{x}_0)(\boldsymbol{x} - \boldsymbol{x}_0)$$

ML Intuition: The Taylor expansion explains why gradient descent works and what its limitations are:

- **First-order methods** (GD, SGD, Adam) only use ∇f—they approximate the loss as a linear function. This is cheap but ignores curvature.
- **Second-order methods** (Newton's method) also use H—they approximate the loss as a quadratic. This is more accurate but requires computing and inverting the Hessian ($O(n^3)$), which is impractical for neural networks with millions of parameters.
- **Quasi-Newton methods** (L-BFGS) approximate the Hessian cheaply.

2.8 Worked Problems

Problem 2.1: Compute the gradient and Hessian of $f(x, y) = x^2 + 4y^2 + xy$. Classify the critical point.

Solution: Gradient: $\nabla f = (2x + y,\ 8y + x)^\top$. Setting to zero: $2x + y = 0$ and $x + 8y = 0$, which gives $x = y = 0$.

Hessian: $H = \begin{pmatrix} 2 & 1 \\ 1 & 8 \end{pmatrix}$. Eigenvalues: $\lambda = 5 \pm \sqrt{10}$, both positive. So $H \succ 0$ and $(0, 0)$ is a **local (and global) minimum**. Condition number: $\kappa \approx 4.4$.

Problem 2.2: Show that the optimal learning rate for gradient descent on $f(\theta) = \frac{1}{2}\theta^\top H\theta$ is $\eta^* = \frac{2}{\lambda_{\max}+\lambda_{\min}}$.

Solution: The GD update is $\theta_{t+1} = (I - \eta H)\theta_t$. Convergence requires $\|I - \eta H\| < 1$, meaning $|1 - \eta\lambda_i| < 1$ for all eigenvalues. This gives $0 < \eta < 2/\lambda_i$ for each i. The optimal η balances the slowest and fastest modes: $\eta^* = \frac{2}{\lambda_{\max}+\lambda_{\min}}$.

The convergence rate is $\rho = \frac{\kappa-1}{\kappa+1}$ per step, where $\kappa = \lambda_{\max}/\lambda_{\min}$. High condition number = slow convergence.

Practice Questions & Answers

Conceptual Questions

Q2.1: Why does the gradient point in the direction of steepest ascent, not steepest descent? And why does this matter for ML?

A2.1: By definition, the directional derivative of f in direction $\boldsymbol{u}$ is $\nabla f \cdot \boldsymbol{u} = \|\nabla f\| \cos\theta$. This is maximized when $\theta = 0$ (i.e., $\boldsymbol{u}$ is parallel to ∇f). So ∇f points where f increases fastest. For ML, we want to *minimize* the loss, so we move in the opposite direction: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla \mathcal{L}$. The negative sign is what turns "steepest ascent" into "steepest descent."

Q2.2: What's the difference between the Jacobian and the Hessian? When does each one matter in ML?

A2.2: The **Jacobian** is the matrix of first derivatives of a vector-valued function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$. It shows up in backpropagation, where each layer's transformation has a Jacobian that gets multiplied in the chain rule. The **Hessian** is the matrix of second derivatives of a scalar function $f : \mathbb{R}^n \to \mathbb{R}$. It captures curvature. The Hessian matters for: (1) classifying critical points (PD = minimum, indefinite = saddle), (2) understanding convergence speed ($\kappa = \lambda_{\max}/\lambda_{\min}$), and (3) second-order optimization methods like Newton's method.

Q2.3: Explain why the chain rule is the single most important calculus concept for deep learning.

A2.3: A neural network is a composition: $f = f_L \circ f_{L-1} \circ \cdots \circ f_1$. To train it, we need $\frac{\partial \mathcal{L}}{\partial W^{(\ell)}}$ for every layer ℓ. Without the chain rule, we'd need to compute each derivative from scratch, requiring a separate forward pass per parameter. The chain rule decomposes this into a product of "local" derivatives: $\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(L)}} \cdot \frac{\partial \boldsymbol{a}^{(L)}}{\partial \boldsymbol{a}^{(L-1)}} \cdots \frac{\partial \boldsymbol{a}^{(\ell)}}{\partial W^{(\ell)}}$. Each factor only requires local information from one layer, making the computation efficient. This is backpropagation.

Multiple Choice

Q2.4: If $f(x, y) = x^2 y + e^y$, what is $\frac{\partial f}{\partial y}$?
(a) $2xy + e^y$
(b) $x^2 + e^y$
(c) $2xy + ye^{y-1}$
(d) $x^2 y + e^y$

Answer: (b) $x^2 + e^y$. Treat x as constant: $\frac{\partial}{\partial y}(x^2 y) = x^2$ and $\frac{\partial}{\partial y}(e^y) = e^y$.

Q2.5: At a critical point, the Hessian has eigenvalues $\{3, 0, -2\}$. The point is:
(a) A local minimum
(b) A local maximum
(c) A saddle point
(d) Cannot be determined

Answer: (c) A saddle point. The Hessian has both positive (3) and negative (-2) eigenvalues, meaning the surface curves up in one direction and down in another.

Q2.6: The condition number of a Hessian is 1000. This means:
(a) Gradient descent will converge quickly
(b) The loss surface is nearly spherical
(c) The loss surface is highly elongated, causing zigzagging
(d) The function has no minimum

Answer: (c). A condition number of 1000 means $\lambda_{\max}/\lambda_{\min} = 1000$. The loss surface is 1000x steeper in one direction than another, causing GD to zigzag along the narrow dimension.

Computational Problems

Q2.7: Compute the gradient and Hessian of $f(x, y) = 3x^2 + xy - y^2 + 5x$. Find all critical points and classify them.

Solution:

$$\nabla f = (6x + y + 5,\ x - 2y)^\top$$
$$H = \begin{pmatrix} 6 & 1 \\ 1 & -2 \end{pmatrix}$$

Setting $\nabla f = \mathbf{0}$: From $x - 2y = 0$, we get $x = 2y$. Substituting: $12y + y + 5 = 0 \implies y = -5/13$, $x = -10/13$.

Eigenvalues of H: $\det(H - \lambda I) = (6 - \lambda)(-2 - \lambda) - 1 = \lambda^2 - 4\lambda - 13 = 0$. $\lambda = 2 \pm \sqrt{17}$, giving $\lambda_1 \approx 6.12 > 0$ and $\lambda_2 \approx -2.12 < 0$. Mixed signs $\implies$ **saddle point**.

Q2.8: Perform one step of gradient descent on $\mathcal{L}(w, b) = \frac{1}{2}(2w + b - 5)^2$ starting from $(w_0, b_0) = (0, 0)$ with $\eta = 0.1$.

Solution: Let $e = 2w + b - 5$. At $(0, 0)$: $e = -5$, $\mathcal{L} = 12.5$.

$$\frac{\partial \mathcal{L}}{\partial w} = e \cdot 2 = -10, \quad \frac{\partial \mathcal{L}}{\partial b} = e \cdot 1 = -5$$

$$w_1 = 0 - 0.1(-10) = 1.0, \quad b_1 = 0 - 0.1(-5) = 0.5$$

New loss: $e = 2(1) + 0.5 - 5 = -2.5$, $\mathcal{L} = 3.125$. Loss dropped from 12.5 to 3.125.

Interview-Style Questions

Q2.9: *"Why do we need the chain rule for training neural networks? Can't we just compute the derivative directly?"*

Model Answer: We could compute each derivative directly using the limit definition, but that would require a separate forward pass for each parameter (perturb one parameter, see how the output changes). A network with 10 million parameters would need 10 million forward passes per gradient computation. The chain rule lets us decompose the derivative into a product of local derivatives, each computed from one layer's information. Backpropagation exploits this by computing all 10 million derivatives in just one forward pass plus one backward pass. That's the difference between training taking hours vs. centuries.

Q2.10: *"What is the relationship between gradient descent and the Taylor expansion?"*

Model Answer: Gradient descent is justified by the first-order Taylor approximation. Near $\boldsymbol{\theta}_t$, we approximate $\mathcal{L}(\boldsymbol{\theta}) \approx \mathcal{L}(\boldsymbol{\theta}_t) + \nabla\mathcal{L}^\top(\boldsymbol{\theta} - \boldsymbol{\theta}_t)$. To decrease $\mathcal{L}$, we move in the direction that makes $\nabla\mathcal{L}^\top(\boldsymbol{\theta} - \boldsymbol{\theta}_t)$ most negative, which is $-\nabla\mathcal{L}$. This approximation is only valid locally, which is why we need a small learning rate. Second-order methods (Newton's method) use the quadratic Taylor approximation with the Hessian, giving better steps but at the cost of computing and inverting an $n \times n$ matrix.

Listing 2.1: Visualizing gradient descent on a 2D loss surface with matplotlib

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

# Loss function: f(x,y) = x^2 + 4*y^2
def loss(x, y): return x**2 + 4*y**2
def grad(x, y): return np.array([2*x, 8*y])

# Gradient descent trajectory
path = [(3.0, 2.0)]
lr = 0.08
for _ in range(30):
    x, y = path[-1]
    g = grad(x, y)
    path.append((x - lr*g[0], y - lr*g[1]))

# Plot
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
X, Y = np.meshgrid(np.linspace(-4, 4, 100), np.linspace(-3, 3, 100))
ax.contour(X, Y, loss(X, Y), levels=20, cmap='viridis', alpha=0.6)
px, py = zip(*path)
ax.plot(px, py, 'r.-', markersize=8, label='GD path')
ax.set_xlabel('x'); ax.set_ylabel('y')
ax.set_title('Gradient Descent on f(x,y) = x^2 + 4y^2')
ax.legend(); plt.tight_layout(); plt.savefig('gd_trajectory.png', dpi=150)
```

Key Takeaways

- Partial derivatives measure sensitivity of the loss to each parameter—the basis of gradient descent.
- The gradient ∇f points in the steepest ascent direction; we descend by moving opposite to it: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla \mathcal{L}$.
- The chain rule makes backpropagation possible by chaining local derivatives through network layers.
- The Hessian captures curvature: its eigenvalues determine convergence speed and whether critical points are minima or saddle points.
- Matrix calculus identities let us derive closed-form solutions (normal equation) and gradient expressions compactly.
- The condition number $\kappa = \lambda_{\max}/\lambda_{\min}$ of the Hessian governs how fast gradient descent converges. Feature scaling reduces κ.

Chapter 3

Probability and Statistics for Machine Learning

3.1 Introduction

Machine learning is applied statistics. Every ML model makes assumptions about the probability distribution that generated the data, and every training algorithm uses statistical principles to fit parameters.

The central idea is simple: we observe a finite sample from an unknown distribution, and we want to learn something about that distribution—its parameters, its structure, or how to make predictions from it. Probability theory tells us how to reason about uncertainty, and statistics tells us how to make inferences from data.

3.2 Probability Basics

A **probability distribution** assigns probabilities to outcomes. For a discrete random variable X:

- **PMF:** $P(X = x) = p(x)$, where $\sum_x p(x) = 1$.
- **Expected value:** $\mathbb{E}[X] = \sum_x x \cdot p(x)$ (discrete) or $\mathbb{E}[X] = \int x\, p(x)\, dx$ (continuous).
- **Variance:** $\mathrm{Var}(X) = \mathbb{E}[(X - \mu)^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$.
- **Standard deviation:** $\sigma = \sqrt{\mathrm{Var}(X)}$.

ML Intuition: The expected loss over the *true data distribution* is the **risk**—what we truly want to minimize:

$$R(\theta) = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\text{data}}}[\mathcal{L}(f_\theta(\boldsymbol{x}), y)]$$

But we don't know p_{data}! In practice, we approximate with the **empirical risk**: the average loss over our training set:

$$\hat{R}(\theta) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(f_\theta(\boldsymbol{x}_i), y_i)$$

The gap between R and $\hat{R}$ is the **generalization gap**. The entire field of statistical learning theory is about bounding this gap.

3.2.1 Conditional Probability and Independence

Conditional probability: $P(A \mid B) = \frac{P(A \cap B)}{P(B)}$
Two events are **independent** if $P(A \cap B) = P(A)P(B)$, equivalently $P(A \mid B) = P(A)$.
The **law of total probability:** $P(A) = \sum_i P(A \mid B_i)P(B_i)$.

ML Intuition: Independence is a strong assumption. The *naive* Bayes classifier assumes all features are conditionally independent given the class:

$$P(\boldsymbol{x} \mid y) = \prod_{j=1}^{n} P(x_j \mid y)$$

This is almost always wrong (e.g., "machine" and "learning" are not independent words in a document). Yet naive Bayes works surprisingly well in practice—a phenomenon that has been well-studied. The lesson: sometimes simple, "wrong" models outperform complex, "correct" ones.

3.3 Key Distributions

Bernoulli: $P(X = k) = p^k(1-p)^{1-k}$, $k \in \{0, 1\}$. Models binary outcomes (spam/not spam, click/no click).
$\mathbb{E}[X] = p$, $\mathrm{Var}(X) = p(1-p)$.
Gaussian (Normal): $p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$. The single most important distribution in ML.
$\mathbb{E}[X] = \mu$, $\mathrm{Var}(X) = \sigma^2$.
Multivariate Gaussian: $p(\boldsymbol{x}) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right)$
where $\boldsymbol{\mu}$ is the mean vector and Σ is the covariance matrix.

Why is the Gaussian so central to ML?

1. **Central Limit Theorem:** The sum (or average) of many independent random variables tends toward a Gaussian, regardless of the original distribution. Since many real-world measurements are sums of many small effects, Gaussian distributions arise naturally.
2. **Linear regression:** If we assume the noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$, maximizing the likelihood leads to minimizing the **sum of squared errors**—i.e., least-squares loss.
3. **Regularization:** A Gaussian prior on weights, $\boldsymbol{w} \sim \mathcal{N}(\mathbf{0}, \frac{1}{\lambda}I)$, gives L^2 regularization (ridge regression).
4. **Maximum entropy:** Among all distributions with fixed mean and variance, the Gaussian has the **maximum entropy**—it's the "least informative" choice, making it a natural default.

Example 3.1: Show that Gaussian noise leads to least-squares loss.
Setup: Assume $y = \boldsymbol{w}^\top \boldsymbol{x} + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Then $p(y \mid \boldsymbol{x}, \boldsymbol{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y-\boldsymbol{w}^\top \boldsymbol{x})^2}{2\sigma^2}\right)$.

Solution: For m i.i.d. data points, the log-likelihood is:

$$\begin{aligned}\ell(\boldsymbol{w}) &= \sum_{i=1}^{m} \log p(y_i \mid \boldsymbol{x}_i, \boldsymbol{w}) \\ &= -\frac{m}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{m}(y_i - \boldsymbol{w}^\top \boldsymbol{x}_i)^2\end{aligned}$$

Maximizing $\ell(\boldsymbol{w})$ is equivalent to **minimizing** $\sum_i (y_i - \boldsymbol{w}^\top \boldsymbol{x}_i)^2$—the sum of squared errors! The probabilistic and geometric views coincide.

3.4 Bayes' Theorem

$$P(\theta \mid \mathcal{D}) = \frac{P(\mathcal{D} \mid \theta)\, P(\theta)}{P(\mathcal{D})}$$

- $P(\theta)$ — **prior**: our belief about θ *before* seeing data.
- $P(\mathcal{D} \mid \theta)$ — **likelihood**: how probable is the observed data given θ.
- $P(\theta \mid \mathcal{D})$ — **posterior**: our updated belief about θ *after* seeing data.
- $P(\mathcal{D})$ — **evidence**: a normalizing constant (often intractable).

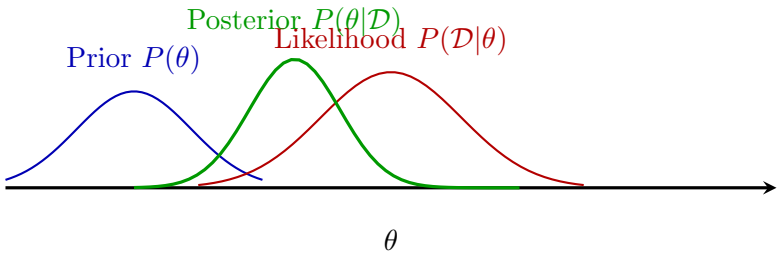


Figure 3.1: Bayes' theorem in action. The posterior (green) is a compromise between the prior belief (blue) and the evidence from data (red). With more data, the likelihood dominates and the posterior concentrates around the true parameter value.

ML Intuition: Bayes' theorem is the foundation of all learning from data. Different ML methods correspond to different ways of using the posterior:

- **Maximum Likelihood Estimation (MLE):** Ignore the prior entirely, maximize $P(\mathcal{D} \mid \theta)$. Simple and widely used, but can overfit.
- **Maximum A Posteriori (MAP):** Maximize $P(\mathcal{D} \mid \theta) \cdot P(\theta)$. This is MLE with regularization—the prior penalizes extreme parameter values.
- **Full Bayesian:** Compute the entire posterior $P(\theta \mid \mathcal{D})$, integrating over all possible θ for predictions. Most principled, but often computationally intractable.

The beautiful connection: **regularization = prior belief**.

- Gaussian prior $\boldsymbol{w} \sim \mathcal{N}(0, \frac{1}{\lambda} I) \longrightarrow L^2$ regularization (ridge regression).
- Laplace prior $p(w_i) \propto e^{-\lambda |w_i|} \longrightarrow L^1$ regularization (lasso).

3.5 Maximum Likelihood Estimation

Given i.i.d. data $\mathcal{D} = \{x_1, \ldots, x_m\}$, the **likelihood** is:

$$L(\theta) = \prod_{i=1}^{m} p(x_i \mid \theta)$$

The **log-likelihood** (easier to work with) is:

$$\ell(\theta) = \sum_{i=1}^{m} \log p(x_i \mid \theta)$$

The MLE is $\hat{\theta}_{\text{MLE}} = \arg\max_\theta \ell(\theta)$.

Example 3.2: MLE for Gaussian mean. Given $x_1, \ldots, x_m \sim \mathcal{N}(\mu, \sigma^2)$ with known σ^2:

$$\ell(\mu) = -\frac{m}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^{m} (x_i - \mu)^2$$

Solution: $\frac{d\ell}{d\mu} = \frac{1}{\sigma^2} \sum_{i=1}^{m} (x_i - \mu) = 0$, giving $\hat{\mu} = \frac{1}{m} \sum_{i=1}^{m} x_i$ (the sample mean). The MLE for the Gaussian mean is simply the average of the data—exactly what our intuition would suggest!

Example 3.3: MLE for Bernoulli parameter. Flip a coin m times, observe k heads. $\ell(p) = k \log p + (m - k) \log(1 - p)$.

Solution: $\frac{d\ell}{dp} = \frac{k}{p} - \frac{m-k}{1-p} = 0 \implies \hat{p} = \frac{k}{m}$.

With 7 heads in 10 flips: $\hat{p} = 0.7$. Again, the MLE matches our intuition.

3.6 Information Theory

Entropy: $H(p) = -\sum_x p(x) \log p(x)$. Measures the "uncertainty" or "surprise" of a distribution. Maximum entropy = uniform distribution (maximum uncertainty).
Cross-entropy: $H(p, q) = -\sum_x p(x) \log q(x)$. Measures the average surprise when using distribution q to encode data from distribution p.
KL divergence: $D_{\text{KL}}(p \| q) = H(p, q) - H(p) = \sum_x p(x) \log \frac{p(x)}{q(x)} \geq 0$. Measures how different q is from p. Equals zero if and only if $p = q$.

ML Intuition: Why is cross-entropy the standard classification loss?
When we train a classifier, the true labels define a distribution p (one-hot vectors), and the model outputs a predicted distribution q (softmax probabilities). Minimizing cross-entropy $H(p, q)$ is equivalent to:

1. **Minimizing KL divergence** $D_{\mathrm{KL}}(p\|q)$ between true and predicted distributions (since $H(p)$ is constant).
2. **Maximizing the log-likelihood** of the correct labels under the model.

All three perspectives—cross-entropy loss, KL divergence minimization, and maximum likelihood—are mathematically identical for classification!

Example 3.4: A 3-class problem with true label $p = (1, 0, 0)$ and model prediction $q = (0.7, 0.2, 0.1)$.
Cross-entropy: $H(p, q) = -(1 \cdot \log 0.7 + 0 \cdot \log 0.2 + 0 \cdot \log 0.1) = -\log 0.7 \approx 0.357$.
If the model were more confident: $q' = (0.95, 0.03, 0.02)$, then $H(p, q') = -\log 0.95 \approx 0.051$.
Lower cross-entropy = better prediction!

3.7 Bias-Variance Decomposition

For any estimator $\hat{\theta}$:

$$\mathbb{E}[(\hat{\theta} - \theta)^2] = \mathrm{Bias}(\hat{\theta})^2 + \mathrm{Var}(\hat{\theta})$$

where $\mathrm{Bias}(\hat{\theta}) = \mathbb{E}[\hat{\theta}] - \theta$ and $\mathrm{Var}(\hat{\theta}) = \mathbb{E}[(\hat{\theta} - \mathbb{E}[\hat{\theta}])^2]$.

ML Intuition: Imagine throwing darts at a target:

- **Low bias, low variance:** Darts clustered around the bullseye. This is what we want!
- **High bias, low variance:** Darts clustered tightly, but off-center. The model is consistently wrong (underfitting).
- **Low bias, high variance:** Darts scattered around the bullseye. The model is right on average but unreliable (overfitting).
- **High bias, high variance:** Scattered and off-center. The worst case.

The bias-variance trade-off is one of the most important concepts in ML. Simple models have high bias and low variance. Complex models have low bias and high variance. The art of ML is finding the sweet spot.

3.8 Worked Problems

Problem 3.1: Show that a Gaussian prior on weights gives ridge regression.

Solution: Assume the model $y = \boldsymbol{w}^\top \boldsymbol{x} + \epsilon$ with $\epsilon \sim \mathcal{N}(0, \sigma^2)$ and prior $\boldsymbol{w} \sim \mathcal{N}(\mathbf{0}, \frac{1}{\lambda} I)$.

The MAP estimate maximizes:

$$\begin{aligned}\log P(\boldsymbol{w} \mid \mathcal{D}) &\propto \log P(\mathcal{D} \mid \boldsymbol{w}) + \log P(\boldsymbol{w}) \\ &= -\frac{1}{2\sigma^2}\sum_i (y_i - \boldsymbol{w}^\top \boldsymbol{x}_i)^2 - \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2 + \text{const}\end{aligned}$$

Maximizing this is equivalent to minimizing:

$$\frac{1}{2\sigma^2}\|X\boldsymbol{w} - \boldsymbol{y}\|^2 + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

This is exactly ridge regression! The regularization strength λ is the ratio of the noise variance to the prior variance.

Problem 3.2: Similarly, show that a Laplace prior gives lasso regression.

Solution: The Laplace prior is $p(w_i) = \frac{\lambda}{2}e^{-\lambda|w_i|}$. The log-prior is $\log P(\boldsymbol{w}) = \text{const} - \lambda\sum_i |w_i| = \text{const} - \lambda\|\boldsymbol{w}\|_1$. Adding this to the log-likelihood gives the lasso objective:

$$\frac{1}{2\sigma^2}\|X\boldsymbol{w} - \boldsymbol{y}\|^2 + \lambda\|\boldsymbol{w}\|_1$$

Practice Questions & Answers

Conceptual Questions

Q3.1: What's the difference between MLE and MAP? When does MAP reduce to MLE, and when does it reduce to the prior?

A3.1: MLE maximizes $P(\mathcal{D}|\theta)$ (likelihood only), ignoring prior beliefs. MAP maximizes $P(\mathcal{D}|\theta) \cdot P(\theta)$ (likelihood $\times$ prior). MAP reduces to MLE when the prior is uniform (flat), because a constant prior doesn't change the argmax. MAP approaches the prior when data is very scarce (likelihood is flat and the prior dominates). With infinite data, both MLE and MAP converge to the true parameter, because the likelihood overwhelms any prior.

Q3.2: Why is cross-entropy, not mean squared error, the standard loss for classification?

A3.2: Three reasons: (1) Cross-entropy is the negative log-likelihood of the Bernoulli/categorical model, so minimizing it is equivalent to maximum likelihood estimation. (2) With MSE and sigmoid, the gradient can be very small when the model is confidently wrong (sigmoid saturation). Cross-entropy has a gradient proportional to the error, so confident wrong predictions get large corrections. (3) Cross-entropy is convex in the logits for logistic regression, guaranteeing a unique global minimum. MSE with sigmoid is non-convex and has multiple local minima.

Q3.3: In your own words, explain the connection between regularization and Bayesian priors.

A3.3: Regularization adds a penalty term $\lambda\Omega(\boldsymbol{w})$ to the loss. In the Bayesian view, this penalty is the negative log-prior $-\log P(\boldsymbol{w})$. A Gaussian prior $\boldsymbol{w} \sim \mathcal{N}(0, \frac{1}{\lambda}I)$ gives $-\log P(\boldsymbol{w}) \propto \lambda\|\boldsymbol{w}\|^2$ (L2 regularization). A Laplace prior gives $-\log P(\boldsymbol{w}) \propto \lambda\|\boldsymbol{w}\|_1$ (L1 regularization). So every regularization scheme is secretly encoding a belief about what "reasonable" weights look like. L2 says "I believe weights should be small." L1 says "I believe most weights should be zero."

Multiple Choice

Q3.4: The KL divergence $D_{\mathrm{KL}}(p\|q)$ is always:
(a) Symmetric: $D_{\mathrm{KL}}(p\|q) = D_{\mathrm{KL}}(q\|p)$
(b) Non-negative: $D_{\mathrm{KL}}(p\|q) \geq 0$
(c) A valid distance metric
(d) Bounded above by 1

Answer: (b). KL divergence is always ≥ 0 (Gibbs' inequality), with equality iff $p = q$. It is *not* symmetric ($D_{\mathrm{KL}}(p\|q) \neq D_{\mathrm{KL}}(q\|p)$ in general) and not a true distance metric (violates symmetry and triangle inequality). It is also unbounded.

Q3.5: If you observe 100 coin flips with 60 heads, the MLE for p is 0.6. A Bayesian with a strong prior $p \sim \mathrm{Beta}(100, 100)$ (centered at 0.5) would estimate p to be approximately:
(a) 0.60 (same as MLE)
(b) 0.50 (ignoring the data)
(c) 0.53 (between prior and MLE)
(d) 0.70 (above MLE)

Answer: (c). The Beta-Binomial conjugate posterior is $\mathrm{Beta}(100+60, 100+40) = \mathrm{Beta}(160, 140)$, giving MAP estimate $\hat{p} \approx 160/300 \approx 0.533$. The strong prior "pulls" the estimate toward 0.5 from the MLE of 0.6.

Q3.6: The Central Limit Theorem states that:
(a) All data follows a Gaussian distribution
(b) The mean of any random variable is Gaussian
(c) The sum of many independent random variables tends toward Gaussian
(d) Gaussian noise is the only valid noise model

Answer: (c). The CLT says that the sum (or average) of many independent random variables, each with finite mean and variance, converges in distribution to a Gaussian, regardless of the original distribution. This is why Gaussian distributions appear naturally wherever many small independent effects combine.

Computational Problems

Q3.7: Given data $\{1, 3, 5, 7, 9\}$, compute the MLE for the mean μ and variance σ^2 of a Gaussian distribution.

Solution:

$$\hat{\mu}_{\text{MLE}} = \frac{1}{5}(1+3+5+7+9) = \frac{25}{5} = 5$$

$$\hat{\sigma}^2_{\text{MLE}} = \frac{1}{5}\sum_{i=1}^{5}(x_i - 5)^2 = \frac{1}{5}(16+4+0+4+16) = \frac{40}{5} = 8$$

Note: the MLE for variance uses $\frac{1}{m}$, not $\frac{1}{m-1}$. The unbiased estimator uses $m-1$: $s^2 = \frac{40}{4} = 10$. For large m, the difference is negligible.

Q3.8: Compute the cross-entropy between $p = (0.7, 0.2, 0.1)$ and $q = (0.5, 0.3, 0.2)$.

Solution:

$$H(p, q) = -(0.7 \log 0.5 + 0.2 \log 0.3 + 0.1 \log 0.2)$$

Using natural log: $= -(0.7(-0.693) + 0.2(-1.204) + 0.1(-1.609)) = 0.485 + 0.241 + 0.161 = 0.887$.

Compare with entropy of p: $H(p) = -(0.7 \log 0.7 + 0.2 \log 0.2 + 0.1 \log 0.1) = 0.802$. So $D_{\text{KL}}(p \| q) = H(p, q) - H(p) = 0.887 - 0.802 = 0.085$.

Interview-Style Questions

Q3.9: *"Explain Bayes' theorem as if you were talking to a product manager who doesn't know statistics."*

Model Answer: Say you're building a spam filter. Before seeing any email, you know about 30% of all emails are spam. That's your "prior belief." Now a new email arrives with the word "winner" in it. You know that 80% of spam emails contain "winner," but only 5% of legitimate emails do. Bayes' theorem lets you combine your prior belief (30% spam) with the new evidence ("winner" appeared) to get an updated belief: the probability this specific email is spam, given it contains "winner." The math works out to about 87%. Bayes' theorem is just a principled way to update your beliefs when you get new evidence. Every ML model that learns from data is doing some version of this.

Q3.10: *"What's the bias-variance trade-off and how does it affect model selection?"*

Model Answer: Every model's prediction error has three parts: bias (systematic error from wrong assumptions), variance (sensitivity to the specific training data), and irreducible noise. Simple models like linear regression have high bias (they can't capture complex patterns) but low variance (they give consistent results across different training sets). Complex models like deep neural networks have low bias but high variance (they can memorize noise in the

training set). The trade-off means you can't minimize both simultaneously. The goal is finding the sweet spot. Practical tools: use cross-validation to estimate test error, regularization to control complexity, and learning curves (plot training vs. validation error as you add data) to diagnose whether you're in the high-bias or high-variance regime.

Listing 3.1: Visualizing the prior-to-posterior update with matplotlib

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import beta

x = np.linspace(0, 1, 500)
fig, axes = plt.subplots(1, 3, figsize=(14, 4))

# Prior: Beta(2, 5) -- believe coin is biased toward tails
prior = beta(2, 5)
axes[0].plot(x, prior.pdf(x), 'b-', lw=2)
axes[0].set_title('Prior: Beta(2,5)'); axes[0].set_xlabel('p')

# After 10 flips, 7 heads: posterior = Beta(2+7, 5+3) = Beta(9, 8)
posterior_10 = beta(9, 8)
axes[1].plot(x, prior.pdf(x), 'b--', alpha=0.5, label='Prior')
axes[1].plot(x, posterior_10.pdf(x), 'r-', lw=2, label='Posterior')
axes[1].set_title('After 10 flips (7H)'); axes[1].legend()

# After 100 flips, 70 heads: posterior = Beta(72, 35)
posterior_100 = beta(72, 35)
axes[2].plot(x, prior.pdf(x), 'b--', alpha=0.5, label='Prior')
axes[2].plot(x, posterior_100.pdf(x), 'r-', lw=2, label='Posterior')
axes[2].axvline(0.7, color='k', ls=':', label='True p=0.7')
axes[2].set_title('After 100 flips (70H)'); axes[2].legend()

plt.tight_layout(); plt.savefig('bayesian_update.png', dpi=150)
```

Key Takeaways

- ML minimizes expected loss (risk); in practice we use empirical risk (average over training data). The gap is the generalization error.
- The Gaussian distribution connects to least-squares (MLE) and L^2 regularization (MAP with Gaussian prior).
- Bayes' theorem unifies MLE, MAP, and full Bayesian inference. Regularization is simply a prior belief about parameters.
- Cross-entropy loss arises from maximizing the likelihood of a categorical model and is equivalent to minimizing KL divergence.
- The bias-variance trade-off governs model selection: simple models underfit (high bias), complex models overfit (high variance).
- Information theory (entropy, cross-entropy, KL divergence) provides the mathematical foundation for classification losses.

Part II

Core Machine Learning

Chapter 4

Linear Regression

4.1 Introduction

Linear regression is the "Hello World" of machine learning. It predicts a continuous output $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + b$ by fitting a hyperplane through data. Despite its simplicity, the ideas behind it, loss functions, optimization, regularization, and the bias-variance trade-off, appear in nearly every other ML method.

Why start with linear regression? Because it is:

- **Interpretable:** Each weight w_j tells you the effect of feature j on the prediction.
- **Analytically solvable:** We can derive the exact optimal parameters (no iterative optimization needed).
- **Foundational:** Logistic regression, neural networks, and even transformers build on the same linear core: $\boldsymbol{z} = W\boldsymbol{x} + \boldsymbol{b}$.

4.2 The Model

Linear regression models the relationship between features $\boldsymbol{x} \in \mathbb{R}^n$ and a continuous target $y \in \mathbb{R}$:

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + b = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$$

In matrix form for m data points: $\hat{\boldsymbol{y}} = X\boldsymbol{w}$ (absorbing b into $\boldsymbol{w}$ by adding a column of ones to X).

Example 4.1 (House Prices): Predicting price from area (x_1) and bedrooms (x_2):

$$\hat{y} = 150 \cdot x_1 + 20000 \cdot x_2 + 50000$$

A 1500 sq ft, 3-bedroom house: $\hat{y} = 150(1500) + 20000(3) + 50000 = \$335{,}000$.
Interpretation: Each additional square foot adds \$150. Each additional bedroom adds \$20,000. The base price is \$50,000.

4.3 The Loss Function

The **Mean Squared Error** (MSE):

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 = \frac{1}{m}\|X\boldsymbol{w} - \boldsymbol{y}\|_2^2$$

Why squared error? Three perspectives:

1. **Calculus:** Differentiable everywhere, making optimization smooth.
2. **Statistics:** Arises from MLE under Gaussian noise assumption.
3. **Geometry:** Minimizing MSE projects $\boldsymbol{y}$ onto the column space of X.

4.4 The Normal Equation

The closed-form solution: $\boldsymbol{w}^* = (X^\top X)^{-1}X^\top \boldsymbol{y}$

Geometric Intuition: The residual $\boldsymbol{y} - X\boldsymbol{w}^*$ is orthogonal to every column of X. The prediction $X\boldsymbol{w}^*$ is the **orthogonal projection** of $\boldsymbol{y}$ onto the column space of X, the closest possible prediction.

Example 4.2: Fit a line to data $(1,2),(2,4),(3,5),(4,4),(5,5)$.

Solution: With a bias column,

$$X = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{pmatrix}, \qquad \boldsymbol{y} = \begin{pmatrix} 2 \\ 4 \\ 5 \\ 4 \\ 5 \end{pmatrix}.$$

$$X^\top X = \begin{pmatrix} 5 & 15 \\ 15 & 55 \end{pmatrix}, \qquad X^\top \boldsymbol{y} = \begin{pmatrix} 20 \\ 66 \end{pmatrix}.$$

$$\boldsymbol{w}^* = (X^\top X)^{-1}X^\top \boldsymbol{y} = \frac{1}{50}\begin{pmatrix} 55 & -15 \\ -15 & 5 \end{pmatrix}\begin{pmatrix} 20 \\ 66 \end{pmatrix} = \begin{pmatrix} 2.2 \\ 0.6 \end{pmatrix}.$$

So the best-fit line is $\hat{y} = 0.6x + 2.2$.

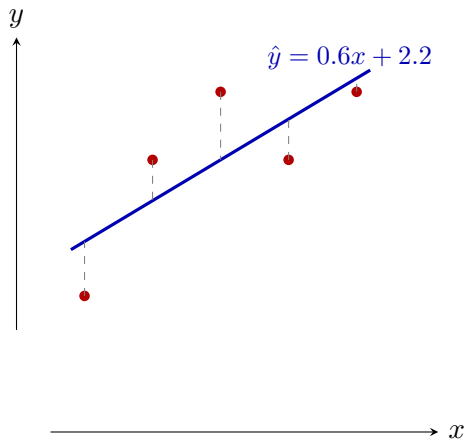


Figure 4.1: The fitted line balances all five points. The dashed segments are residuals, and least squares minimizes the sum of their squares.

Quick Check 4.1. Why does the fitted line not pass through every point in this example?

Answer. A single straight line has only two degrees of freedom, slope and intercept. These five points are not perfectly collinear, so no line can hit all of them exactly. Least squares instead picks the line that keeps the total squared residual as small as possible.

Listing 4.1: Linear regression with NumPy

```python
import numpy as np

x = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 5, 4, 5])
X = np.column_stack([np.ones(len(x)), x])

# Normal equation
w = np.linalg.solve(X.T @ X, X.T @ y)
print(f"y = {w[1]:.1f}x + {w[0]:.1f}")  # y = 0.6x + 2.2
```

4.5 Gradient Descent for Linear Regression

The gradient of MSE: $\nabla_{\boldsymbol{w}} \mathcal{L} = \frac{1}{m} X^\top (X\boldsymbol{w} - \boldsymbol{y})$.
Update: $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{m} X^\top (X\boldsymbol{w}_t - \boldsymbol{y})$.

Interpretation: Each term $(\hat{y}_i - y_i)\boldsymbol{x}_i$ says: the prediction was off by $(\hat{y}_i - y_i)$, so adjust weights in the direction of $\boldsymbol{x}_i$ proportionally.

4.6 Regularization

Ridge (L^2): $\mathcal{L}_{\text{ridge}} = \frac{1}{m}\|X\boldsymbol{w} - \boldsymbol{y}\|^2 + \lambda\|\boldsymbol{w}\|_2^2$. Solution: $(X^\top X + m\lambda I)^{-1}X^\top \boldsymbol{y}$.
Lasso (L^1): $\mathcal{L}_{\text{lasso}} = \frac{1}{m}\|X\boldsymbol{w} - \boldsymbol{y}\|^2 + \lambda\|\boldsymbol{w}\|_1$. Produces **sparse** weights (feature selection).

Method	**Penalty**	**Typical effect**
Ridge (L^2)	$\lambda\|\boldsymbol{w}\|_2^2$	Shrinks all weights toward zero and stabilizes the fit when features are correlated.
Lasso (L^1)	$\lambda\|\boldsymbol{w}\|_1$	Pushes some weights to zero, so it can act as a feature selector.

Why L^1 produces sparsity: The L^1 constraint region is a diamond with corners on axes. The tangent point with the loss contours often lands on a corner, setting some weights to exactly zero.

4.7 The Bias-Variance Trade-Off

$\mathbb{E}[(y - \hat{f})^2] = \text{Bias}^2 + \text{Variance} + \text{Irreducible noise}$

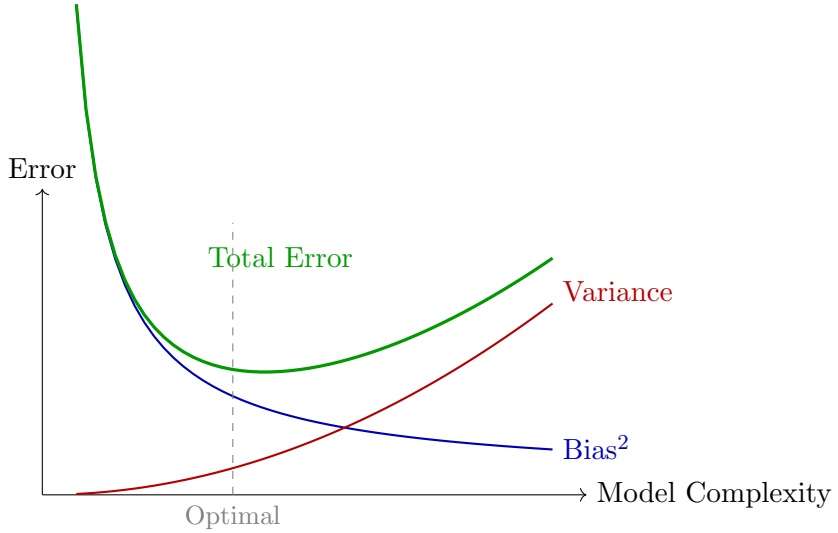


Figure 4.2: The bias-variance trade-off. Increasing complexity reduces bias but increases variance. Regularization helps choose a useful middle ground.

Practice Questions & Answers

Conceptual Questions

Q4.1. Derive the normal equation $\boldsymbol{w}^* = (X^\top X)^{-1} X^\top \boldsymbol{y}$ starting from the MSE cost function. Under what conditions does this solution fail to exist?

Answer. We minimise $J(\boldsymbol{w}) = \frac{1}{2m}\|X\boldsymbol{w} - \boldsymbol{y}\|^2$. Expanding and differentiating:

$$\nabla_{\boldsymbol{w}} J = \frac{1}{m} X^\top (X\boldsymbol{w} - \boldsymbol{y}) = \boldsymbol{0} \quad \Longrightarrow \quad X^\top X\, \boldsymbol{w} = X^\top \boldsymbol{y} \quad \Longrightarrow \quad \boldsymbol{w}^* = (X^\top X)^{-1} X^\top \boldsymbol{y}.$$

This requires $X^\top X$ to be invertible, i.e. X must have full column rank ($m \geq d$ and no perfectly collinear features). When $X^\top X$ is singular (multicollinearity or $d > m$), the solution is non-unique. Adding L^2 regularisation replaces $X^\top X$ with $(X^\top X + \lambda I)$, which is always invertible for $\lambda > 0$.

Q4.2. Explain the bias-variance decomposition of the expected test MSE. How does model complexity affect each term, and how does regularisation help?

Answer. For a fixed test point $\boldsymbol{x}_0$ the expected MSE decomposes as:

$$\mathbb{E}\big[(y_0 - \hat{f}(\boldsymbol{x}_0))^2\big] = \underbrace{\big(\mathbb{E}[\hat{f}] - f\big)^2}_{\text{Bias}^2} + \underbrace{\mathbb{E}\big[(\hat{f} - \mathbb{E}[\hat{f}])^2\big]}_{\text{Variance}} + \sigma_\epsilon^2.$$

Bias measures systematic error from overly simple models (underfitting). **Variance** measures sensitivity to training data from overly complex models (overfitting). σ_ϵ^2 is irreducible noise. Regularisation (Ridge or Lasso) intentionally increases bias by shrinking weights, but reduces variance substantially, often lowering total error.

Q4.3. Compare Ridge (L^2) and Lasso (L^1) regression. When would you prefer one over the other?

Answer.

- **Ridge** adds $\lambda\|\boldsymbol{w}\|_2^2$ to the loss. It shrinks all weights toward zero but never exactly to zero. The solution is $(X^\top X + \lambda I)^{-1} X^\top \boldsymbol{y}$. Prefer Ridge when all features are potentially relevant and multicollinearity is present.
- **Lasso** adds $\lambda\|\boldsymbol{w}\|_1$. It produces *sparse* solutions, driving some weights exactly to zero, thus performing automatic feature selection. Prefer Lasso when you suspect many features are irrelevant.
- **Elastic Net** combines both: $\lambda_1\|\boldsymbol{w}\|_1 + \lambda_2\|\boldsymbol{w}\|_2^2$. Useful when features are correlated and you still want sparsity.

Geometrically, the L^1 constraint region has corners on the axes where weights become exactly zero; the L^2 region is a smooth ball that merely shrinks weights.

Multiple Choice

Q4.4. What is the computational complexity of solving the normal equation for m data points and d features?
(a) $O(md)$
(b) $O(md^2 + d^3)$
(c) $O(m^2 d)$
(d) $O(m^3)$

Answer: (b). Computing $X^\top X$ is $O(md^2)$ and inverting the $d \times d$ matrix is $O(d^3)$. This makes the normal equation impractical when d is very large, but efficient when $d \ll m$.

Q4.5. A model has high training error and high test error. This is most likely due to:
(a) High variance (overfitting)
(b) High bias (underfitting)
(c) Irreducible noise only
(d) Label leakage

Answer: (b). High error on *both* training and test sets indicates underfitting, so the model is too simple to capture the underlying pattern (high bias). High variance would show low training error but high test error.

Q4.6. Adding polynomial features of degree p to a linear regression model:
(a) Increases bias, decreases variance
(b) Decreases bias, increases variance
(c) Decreases both bias and variance
(d) Has no effect on the bias-variance trade-off

Answer: (b). Polynomial features increase model capacity. The richer hypothesis class can fit the training data more closely (lower bias), but becomes more sensitive to the specific training set (higher variance). Regularisation is typically needed to control the variance increase.

Computational Problems

Q4.7. Given the design matrix $X = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{pmatrix}$ and targets $\boldsymbol{y} = \begin{pmatrix} 2 \\ 4 \\ 5 \end{pmatrix}$, compute the optimal weights using the normal equation.

Solution.

$$X^\top X = \begin{pmatrix} 3 & 6 \\ 6 & 14 \end{pmatrix}, \qquad X^\top \boldsymbol{y} = \begin{pmatrix} 11 \\ 25 \end{pmatrix}.$$

$$(X^\top X)^{-1} = \frac{1}{3 \cdot 14 - 6 \cdot 6} \begin{pmatrix} 14 & -6 \\ -6 & 3 \end{pmatrix} = \frac{1}{6} \begin{pmatrix} 14 & -6 \\ -6 & 3 \end{pmatrix}.$$

$$\boldsymbol{w}^* = \frac{1}{6} \begin{pmatrix} 14 & -6 \\ -6 & 3 \end{pmatrix} \begin{pmatrix} 11 \\ 25 \end{pmatrix} = \frac{1}{6} \begin{pmatrix} 154 - 150 \\ -66 + 75 \end{pmatrix} = \frac{1}{6} \begin{pmatrix} 4 \\ 9 \end{pmatrix} = \begin{pmatrix} 2/3 \\ 3/2 \end{pmatrix}.$$

So the best-fit line is $\hat{y} = \frac{2}{3} + \frac{3}{2}x \approx 0.67 + 1.5x$.

Q4.8. Compute the Ridge regression solution with $\lambda = 2$ for the same data as Q4.7.

Solution.

$$X^\top X + \lambda I = \begin{pmatrix} 3 & 6 \\ 6 & 14 \end{pmatrix} + \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 5 & 6 \\ 6 & 16 \end{pmatrix}.$$

$$(X^\top X + \lambda I)^{-1} = \frac{1}{5 \cdot 16 - 36} \begin{pmatrix} 16 & -6 \\ -6 & 5 \end{pmatrix} = \frac{1}{44} \begin{pmatrix} 16 & -6 \\ -6 & 5 \end{pmatrix}.$$

$$\boldsymbol{w}^*_{\text{ridge}} = \frac{1}{44} \begin{pmatrix} 16 & -6 \\ -6 & 5 \end{pmatrix} \begin{pmatrix} 11 \\ 25 \end{pmatrix} = \frac{1}{44} \begin{pmatrix} 176 - 150 \\ -66 + 125 \end{pmatrix} = \frac{1}{44} \begin{pmatrix} 26 \\ 59 \end{pmatrix} \approx \begin{pmatrix} 0.591 \\ 1.341 \end{pmatrix}.$$

Compared to the unregularised solution, both weights move toward zero: the intercept drops from 0.67 to 0.59, and the slope drops from 1.5 to 1.34.

Q4.9. A dataset has $m = 50$ points and $d = 200$ features. Explain why the normal equation fails and describe two remedies.

Solution. With $d > m$, the matrix $X^\top X \in \mathbb{R}^{200 \times 200}$ has rank at most 50, so it is singular and cannot be inverted. Two remedies:

1. **Ridge regularisation:** Replace $(X^\top X)^{-1}$ with $(X^\top X + \lambda I)^{-1}$, which is always invertible for $\lambda > 0$.
2. **Lasso / feature selection:** Use L^1 penalty to select at most m relevant features, reducing the effective dimensionality.

Alternatively, one could use gradient descent, which does not require explicit matrix inversion, or apply PCA to reduce dimensionality before fitting.

Interview-Style Questions

Q4.10. *Your linear regression model achieves near-zero training MSE but very high test MSE. Walk me through how you would diagnose and fix this problem.*

Model Answer. This is a classic overfitting scenario (high variance). My diagnostic and remediation steps:

1. **Learning curves:** Plot training and validation error vs. training set size. A large gap confirms overfitting.
2. **Feature audit:** Check if d is too large relative to m. Remove irrelevant or highly correlated features.
3. **Regularisation:** Add Ridge (L^2) or Lasso (L^1) penalty. Tune λ via cross-validation.
4. **Reduce complexity:** If using polynomial features, lower the degree.
5. **More data:** If feasible, collect more training examples to reduce variance.
6. **Cross-validation:** Use k-fold CV rather than a single train/test split for more reliable evaluation.

Q4.11. *Explain why feature scaling matters for regularised linear regression but not for ordinary least squares.*

Model Answer. In ordinary least squares, rescaling a feature x_j by factor c simply rescales its weight w_j by $1/c$, so the predictions $\hat{y} = X\boldsymbol{w}$ are invariant. However, regularisation penalises $\|\boldsymbol{w}\|$: if features are on different scales, the penalty disproportionately shrinks weights of large-scale features while under-penalising small-scale features. Standardising all features to zero mean and unit variance ensures the penalty treats all features equally, leading to a fairer and more effective regularisation.

Python Visualisation

Listing 4.2: Bias-variance trade-off: polynomial regression with varying degree

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

np.random.seed(42)
x = np.linspace(0, 1, 30)
y_true = np.sin(2 * np.pi * x)
y = y_true + 0.3 * np.random.randn(len(x))

degrees = [1, 4, 15]
fig, axes = plt.subplots(1, 3, figsize=(14, 4))
x_plot = np.linspace(0, 1, 200)

for ax, d in zip(axes, degrees):
    coeffs = np.polyfit(x, y, d)
    y_pred = np.polyval(coeffs, x_plot)
    ax.scatter(x, y, s=20, alpha=0.7, label='Data')
    ax.plot(x_plot, np.sin(2*np.pi*x_plot), 'g--',
            label='True function')
    ax.plot(x_plot, y_pred, 'r-', lw=2,
            label=f'Degree {d} fit')
    ax.set_title(f'Degree {d}')
    ax.set_ylim(-2, 2)
    ax.legend(fontsize=8)

fig.suptitle('Bias-Variance Trade-off: Polynomial Regression',
             fontsize=13)
plt.tight_layout()
plt.savefig('ch4_bias_variance.pdf')
plt.show()
```

Key Takeaways

- Linear regression minimizes MSE, which arises from Gaussian noise (MLE).
- The normal equation: $\boldsymbol{w}^* = (X^\top X)^{-1} X^\top \boldsymbol{y}$ (projection onto column space of X).
- Gradient descent is an iterative alternative for large n.
- Ridge (L^2) shrinks weights; Lasso (L^1) zeros out weights (feature selection).
- The bias-variance trade-off: too simple = underfit, too complex = overfit.

Chapter 5

Gradient Descent and Optimization

5.1 Introduction

Training an ML model means finding parameters $\boldsymbol{\theta}$ that minimize a loss function $\mathcal{L}(\boldsymbol{\theta})$. Gradient descent is the workhorse of modern ML optimization.

The Core Idea: Imagine standing on a hilly landscape in thick fog. You can feel the slope beneath your feet. Gradient descent says: always step downhill in the steepest direction. The learning rate controls your step size.

5.2 Batch Gradient Descent

$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla \mathcal{L}(\boldsymbol{\theta}_t)$
For quadratic loss with Hessian H: converges iff $0 < \eta < 2/\lambda_{\max}(H)$. High condition number $\kappa = \lambda_{\max}/\lambda_{\min}$ causes zigzagging.

Example 5.1: 3 steps of GD on $f(\theta) = (\theta - 3)^2$ from $\theta_0 = 0$, $\eta = 0.1$:
$\theta_1 = 0 - 0.1 \cdot 2(0 - 3) = 0.6$, $\theta_2 = 0.6 + 0.48 = 1.08$, $\theta_3 = 1.08 + 0.384 = 1.464$.

5.3 Stochastic Gradient Descent

SGD uses a single random sample: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla \mathcal{L}_i(\boldsymbol{\theta}_t)$.
Mini-batch GD uses a random subset of size B: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{B} \sum_{i \in \mathcal{B}} \nabla \mathcal{L}_i$.

Why SGD works: $\mathbb{E}[\nabla \mathcal{L}_i] = \nabla \mathcal{L}$ (unbiased). The noise helps escape shallow minima and saddle points. Variance $\propto 1/B$—larger batches are smoother.

Listing 5.1: Mini-batch SGD implementation

```
import numpy as np

def sgd(X, y, lr=0.01, epochs=100, batch_size=32):
    m, n = X.shape
    w = np.zeros(n)
    for epoch in range(epochs):
        indices = np.random.permutation(m)
```

```
        for start in range(0, m, batch_size):
            batch = indices[start:start+batch_size]
            grad = X[batch].T @ (X[batch] @ w - y[batch]) / len(batch)
            w -= lr * grad
    return w
```

5.4 Momentum

SGD with Momentum:

$$\boldsymbol{v}_{t+1} = \beta\, \boldsymbol{v}_t + \nabla \mathcal{L}(\boldsymbol{\theta}_t)$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta\, \boldsymbol{v}_{t+1}$$

Typical $\beta = 0.9$.

Intuition: Like a ball rolling downhill with inertia. Consistent gradients build velocity (acceleration); oscillating gradients cancel (damping). Reduces zigzagging on elongated loss surfaces.

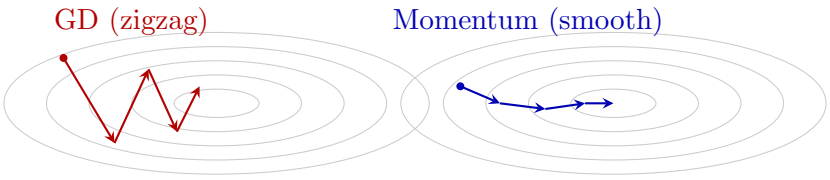


Figure 5.1: GD zigzags on elongated surfaces; momentum smooths the path.

5.5 Adam Optimizer

Adam combines momentum and adaptive learning rates:

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\nabla\mathcal{L} \qquad \text{(1st moment: momentum)}$$
$$\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)(\nabla\mathcal{L})^2 \qquad \text{(2nd moment: RMSProp)}$$
$$\hat{\boldsymbol{m}}_t = \boldsymbol{m}_t/(1-\beta_1^t),\ \hat{\boldsymbol{v}}_t = \boldsymbol{v}_t/(1-\beta_2^t) \qquad \text{(bias correction)}$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon}\hat{\boldsymbol{m}}_t \qquad \text{(update)}$$

Defaults: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\eta = 0.001$.

Why Adam is popular: Works out-of-the-box, handles sparse gradients, fast convergence. Parameters with large gradients get smaller effective learning rates; parameters with small gradients get larger ones.

5.6 Learning Rate Schedules

- **Step decay:** Multiply η by 0.1 every k epochs.
- **Cosine annealing:** $\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})(1 + \cos(\pi t/T))$.
- **Linear warmup + decay:** Start small, ramp up, then decay. Essential for Transformers.

Intuition: "Explore early, exploit late." Large η early explores the landscape broadly. Small η late fine-tunes into the best minimum.

Practice Questions & Answers

Conceptual Questions

Q5.1. Compare Batch Gradient Descent, Stochastic Gradient Descent (SGD), and Mini-Batch Gradient Descent in terms of convergence behaviour, computational cost per update, and noise properties.

Answer.

- **Batch GD:** Uses the full dataset for each gradient computation. Produces the exact gradient, converges smoothly, but each step costs $O(md)$. Impractical for large m.
- **SGD ($B = 1$):** Uses a single random sample per update. Very noisy gradient estimates but extremely fast per step. The noise can help escape shallow local minima but makes convergence oscillatory. Requires decreasing learning rate for convergence guarantees.
- **Mini-Batch ($1 < B < m$):** Compromise—uses B samples per step. Reduces variance by factor $\sim 1/B$ compared to SGD while remaining efficient. Exploits hardware parallelism (GPU vectorisation). The standard choice in practice with $B \in [32, 512]$.

Key trade-off: larger batches $\rightarrow$ more accurate gradient $\rightarrow$ smoother convergence, but each step is slower and may converge to sharper (less generalisable) minima.

Q5.2. Why is momentum useful in gradient descent? Explain the analogy with physics and write the update equations.

Answer. Momentum accumulates an exponentially weighted moving average of past gradients, acting like a "velocity" term:

$$\boldsymbol{v}_t = \beta\, \boldsymbol{v}_{t-1} + (1 - \beta)\, \nabla J(\boldsymbol{w}_t), \qquad \boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta\, \boldsymbol{v}_t.$$

(Some formulations absorb the $(1 - \beta)$ differently.)

Physics analogy: Imagine a ball rolling down a hilly loss landscape. Without momentum, it changes direction instantly with the local slope. With momentum ($\beta \approx 0.9$), it builds up speed in consistent downhill directions and dampens oscillations in directions where the gradient alternates sign. This is particularly helpful for elongated loss surfaces (high condition number), where vanilla GD oscillates across the narrow valley while making slow progress along

the long axis. Momentum accelerates along the long axis and smooths out the oscillations.

Q5.3. Explain the Adam optimiser. What problem does it solve over plain SGD with momentum, and what are its two key ideas?

Answer. Adam (Adaptive Moment Estimation) combines two ideas:

1. **First moment (momentum):** $\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t$ tracks the mean gradient direction.
2. **Second moment (RMSProp):** $\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2$ tracks the mean squared gradient per parameter.

Bias-corrected estimates: $\hat{\boldsymbol{m}}_t = \boldsymbol{m}_t/(1-\beta_1^t)$, $\hat{\boldsymbol{v}}_t = \boldsymbol{v}_t/(1-\beta_2^t)$.
Update: $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta\, \hat{\boldsymbol{m}}_t/(\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon)$.
The *adaptive* per-parameter learning rate $\eta/\sqrt{\hat{v}_{t,j}}$ automatically scales down updates for parameters with large gradients and scales up updates for parameters with small gradients. This eliminates much of the need for manual learning rate tuning. Default hyperparameters ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$) work well in most settings.

Multiple Choice

Q5.4. If the learning rate η is set too high in gradient descent, what is the most likely outcome?

(a) The algorithm converges to a better minimum
(b) The loss oscillates or diverges
(c) The algorithm converges faster to the global minimum
(d) The gradient becomes exactly zero

Answer: (b). An excessively large learning rate causes the updates to overshoot the minimum, leading to oscillation or divergence. For a quadratic loss with curvature (eigenvalue) $\lambda_{\max}$, stability requires $\eta < 2/\lambda_{\max}$.

Q5.5. Which of the following is *not* an advantage of mini-batch SGD over full-batch gradient descent?

(a) Faster wall-clock time per epoch for large datasets
(b) Better generalisation due to gradient noise
(c) Guaranteed convergence to the global minimum
(d) Better utilisation of GPU parallelism

Answer: (c). Mini-batch SGD does *not* guarantee convergence to the global minimum (the loss landscape for neural networks is typically non-convex). Options (a), (b), and (d) are genuine advantages: mini-batch is faster per epoch, the stochastic noise acts as implicit regularisation aiding generalisation, and batch operations exploit GPU parallelism efficiently.

Q5.6. In the Adam optimiser, the bias correction terms $\hat{m}_t = m_t/(1-\beta_1^t)$ exist because:

(a) The gradients may contain NaN values

(b) The moving averages are initialised at zero, creating a bias toward zero in early steps
(c) The learning rate needs to decay over time
(d) The second moment estimate can become negative

Answer: (b). Since $\boldsymbol{m}_0 = \mathbf{0}$ and $\boldsymbol{v}_0 = \mathbf{0}$, the exponential moving averages are biased toward zero during the first few iterations. The correction factor $1/(1-\beta^t)$ compensates for this initialisation bias. As $t \to \infty$, $\beta^t \to 0$ and the correction vanishes.

Computational Problems

Q5.7. Consider the 1D loss function $J(w) = w^2 + 4$. Starting at $w_0 = 10$ with learning rate $\eta = 0.3$, compute three steps of gradient descent and the loss at each step.

Solution. The gradient is $J'(w) = 2w$. Update rule: $w_{t+1} = w_t - 0.3 \cdot 2w_t = w_t(1 - 0.6) = 0.4\, w_t$.

$$\begin{aligned} w_0 &= 10, & J(w_0) &= 104 \\ w_1 &= 0.4 \times 10 = 4, & J(w_1) &= 20 \\ w_2 &= 0.4 \times 4 = 1.6, & J(w_2) &= 6.56 \\ w_3 &= 0.4 \times 1.6 = 0.64, & J(w_3) &= 4.41 \end{aligned}$$

The loss decreases monotonically toward the minimum $J^* = 4$ at $w^* = 0$. In general, $w_t = 0.4^t \cdot 10$ converges geometrically.

Q5.8. For the loss $J(w) = w^2$, determine the maximum learning rate η for which gradient descent converges. What happens at exactly this critical value?

Solution. The update is $w_{t+1} = w_t - \eta \cdot 2w_t = (1 - 2\eta)w_t$. Convergence requires $|1 - 2\eta| < 1$, giving $0 < \eta < 1$.

At $\eta = 1$: $w_{t+1} = -w_t$, so the iterates oscillate between $+w_0$ and $-w_0$ without converging. At $\eta > 1$: $|1 - 2\eta| > 1$, so $|w_t| \to \infty$ (divergence). The optimal rate is $\eta^* = 0.5$, giving $w_{t+1} = 0$ (convergence in one step), since $1 - 2(0.5) = 0$.

Q5.9. Compute one step of the Adam update for a single parameter w given: $g_1 = 3.0$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\eta = 0.01$, $\epsilon = 10^{-8}$, and initial $m_0 = 0$, $v_0 = 0$.

Solution. At $t = 1$:

$$\begin{aligned} m_1 &= 0.9 \cdot 0 + 0.1 \cdot 3.0 = 0.3 \\ v_1 &= 0.999 \cdot 0 + 0.001 \cdot 9.0 = 0.009 \\ \hat{m}_1 &= \frac{0.3}{1 - 0.9^1} = \frac{0.3}{0.1} = 3.0 \\ \hat{v}_1 &= \frac{0.009}{1 - 0.999^1} = \frac{0.009}{0.001} = 9.0 \\ \Delta w &= \frac{0.01 \times 3.0}{\sqrt{9.0} + 10^{-8}} = \frac{0.03}{3.0} = 0.01 \end{aligned}$$

So $w_1 = w_0 - 0.01$. Note that bias correction was crucial: without it, $m_1 = 0.3$ and $\sqrt{v_1} = 0.095$ would give a very different (and incorrect) step size.

Interview-Style Questions

Q5.10. *You are training a deep neural network and the training loss plateaus after a few epochs. What learning rate strategies would you try, and why?*

Model Answer. A plateau often means the learning rate is either too small (stuck in a flat region) or needs scheduling. My approach:

1. **Learning rate warmup:** Start with a small η, linearly increase over k steps. Prevents instability in early training when gradients are large and noisy.
2. **Cosine annealing:** After warmup, decay η following $\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})(1 + \cos(\pi t/T))$. Smoothly reduces the rate, allowing fine-grained convergence.
3. **Reduce-on-plateau:** Monitor validation loss; if it stagnates for p epochs, reduce η by a factor (e.g., $\times 0.1$).
4. **Cyclical learning rates:** Oscillate η between bounds. Can help escape saddle points.
5. **Switch optimiser:** If using plain SGD, try Adam which adapts per-parameter rates automatically.

I would also verify that the plateau is not caused by a bug (e.g., frozen layers, zero gradients due to dead ReLUs).

Q5.11. *Why might SGD with momentum generalise better than Adam in some cases, despite Adam converging faster?*

Model Answer. Empirically, SGD with momentum often finds "flatter" minima in the loss landscape, which tend to generalise better (the loss doesn't change much when weights are perturbed). Adam's adaptive per-parameter rates can converge to sharp minima that have low training loss but poor test performance. The reasons include:

- Adam's effective learning rate can become very small for frequently updated parameters, causing it to "lock in" to the nearest sharp minimum.
- SGD's uniform learning rate and noise act as implicit regularisation, encouraging flatter solutions.
- The noise from small batch sizes in SGD provides a "temperature" that helps escape sharp minima.

In practice, many vision tasks (e.g., ImageNet) still use SGD+momentum with a cosine schedule. NLP and Transformer models typically use Adam with warmup, partly because the loss landscape is more complex and Adam's adaptivity is more beneficial there.

Python Visualisation

Listing 5.2: Comparing SGD, Momentum, and Adam on a 2D loss surface

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

def rosenbrock(x, y, a=1, b=5):
    return (a - x)**2 + b * (y - x**2)**2

def grad_rosenbrock(x, y, a=1, b=5):
```

```
    dx = -2*(a - x) + b * 2*(y - x**2)*(-2*x)
    dy = b * 2*(y - x**2)
    return np.array([dx, dy])

def run_sgd(lr=0.002, steps=300):
    w = np.array([-1.0, 1.5])
    path = [w.copy()]
    for _ in range(steps):
        g = grad_rosenbrock(w[0], w[1])
        w -= lr * g
        path.append(w.copy())
    return np.array(path)

def run_momentum(lr=0.002, beta=0.9, steps=300):
    w = np.array([-1.0, 1.5])
    v = np.zeros(2)
    path = [w.copy()]
    for _ in range(steps):
        g = grad_rosenbrock(w[0], w[1])
        v = beta * v + g
        w -= lr * v
        path.append(w.copy())
    return np.array(path)

def run_adam(lr=0.05, steps=300):
    w = np.array([-1.0, 1.5])
    m, v = np.zeros(2), np.zeros(2)
    path = [w.copy()]
    for t in range(1, steps + 1):
        g = grad_rosenbrock(w[0], w[1])
        m = 0.9*m + 0.1*g
        v = 0.999*v + 0.001*g**2
        mh, vh = m/(1-0.9**t), v/(1-0.999**t)
        w -= lr * mh / (np.sqrt(vh) + 1e-8)
        path.append(w.copy())
    return np.array(path)

x = np.linspace(-2, 2, 200)
y = np.linspace(-1, 3, 200)
X, Y = np.meshgrid(x, y)
Z = rosenbrock(X, Y)

fig, ax = plt.subplots(figsize=(8, 6))
ax.contour(X, Y, Z, levels=np.logspace(-1, 3, 30),
           cmap='viridis', alpha=0.6)

for runner, label, color in [
    (run_sgd, 'SGD', 'red'),
    (run_momentum, 'Momentum', 'blue'),
    (run_adam, 'Adam', 'orange')]:
    path = runner()
    ax.plot(path[:,0], path[:,1], '-', color=color,
            lw=1.5, label=label, alpha=0.8)
    ax.plot(path[0,0], path[0,1], 'o', color=color, ms=6)
```

```
62
63 ax.plot(1, 1, 'k*', ms=15, label='Minimum (1,1)')
64 ax.set_xlabel('$w_1$'); ax.set_ylabel('$w_2$')
65 ax.set_title('Optimiser Trajectories on Rosenbrock Surface')
66 ax.legend(); plt.tight_layout()
67 plt.savefig('ch5_optimisers.pdf')
68 plt.show()
```

Key Takeaways

- Batch GD: exact gradient, slow. SGD: noisy but fast. Mini-batch: practical middle ground.
- Learning rate is critical. Feature scaling reduces the condition number and helps convergence.
- Momentum builds velocity in consistent gradient directions and damps oscillations.
- Adam = momentum + adaptive per-parameter learning rates. Works well out-of-the-box.
- Learning rate schedules (warmup + cosine decay) implement "explore early, exploit late."

Chapter 6

Logistic Regression and Classification

6.1 Introduction

While linear regression predicts continuous values, many problems require predicting **categories**. Logistic regression adapts the linear model to output probabilities by passing the linear score through a sigmoid function.

6.2 The Sigmoid Function

$\sigma(z) = \frac{1}{1+e^{-z}}$. Maps any real number to $(0, 1)$.
Properties: $\sigma(0) = 0.5$, $\sigma(-z) = 1 - \sigma(z)$, $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, max derivative $= 0.25$.

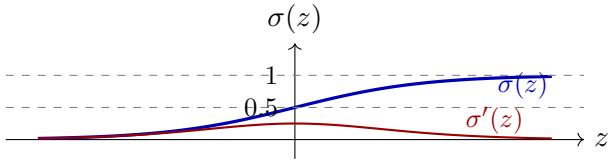


Figure 6.1: The sigmoid (blue) squashes inputs to $(0, 1)$. Its derivative (red) peaks at $z = 0$ and vanishes for $|z| \gg 0$ (the vanishing gradient problem).

6.3 The Logistic Regression Model

$P(y = 1 \mid \boldsymbol{x}) = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$. Predict class 1 if $P \geq 0.5$ (i.e., $\boldsymbol{w}^\top \boldsymbol{x} + b \geq 0$). The decision boundary is the hyperplane $\boldsymbol{w}^\top \boldsymbol{x} + b = 0$.

Log-Odds Interpretation: $\log \frac{P(y=1)}{P(y=0)} = \boldsymbol{w}^\top \boldsymbol{x} + b$. Each weight w_j tells you: increasing x_j by 1 increases the log-odds of class 1 by w_j.

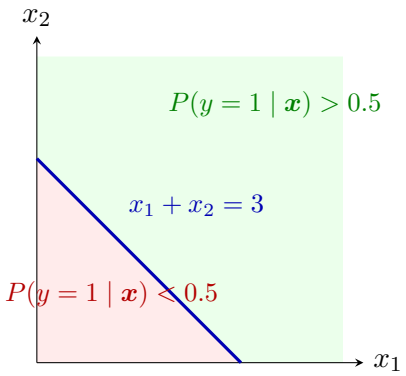


Figure 6.2: Thresholding the sigmoid at 0.5 is the same as thresholding the linear score at 0. For the simple score $z = -3 + x_1 + x_2$, the decision boundary is the line $x_1 + x_2 = 3$.

Quick Check 6.1. Why does the probability threshold 0.5 lead to a linear decision boundary?

Answer. The sigmoid is monotone increasing, and $\sigma(0) = 0.5$. So $P(y = 1 \mid \boldsymbol{x}) \geq 0.5$ holds exactly when the logit $\boldsymbol{w}^\top \boldsymbol{x} + b$ is at least 0. That logit is a linear function of the features, which is why the boundary is a hyperplane.

6.4 The Cross-Entropy Loss

$\mathcal{L} = -\frac{1}{m}\sum_i [y_i \log \hat{p}_i + (1 - y_i)\log(1 - \hat{p}_i)]$
Gradient: $\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = \frac{1}{m}\sum_i (\hat{p}_i - y_i)\boldsymbol{x}_i$, which is the prediction error times the input.

Why cross-entropy, not MSE? The two losses behave differently once a sigmoid is part of the model:

Loss	**When the model is confidently wrong**	**Practical consequence**
MSE with sigmoid	The gradient includes $\sigma'(z)$, which becomes very small in the saturated tails.	Learning can slow down exactly when the model most needs a strong correction.
Cross-entropy	The loss grows sharply as $\hat{p}$ approaches 0 for a positive example or 1 for a negative one.	Updates stay informative for large mistakes, and the objective remains convex in $(\boldsymbol{w}, b)$, though redundant features can still lead to multiple minimizers.

Cross-entropy is also the negative log-likelihood of the Bernoulli model, so it matches the probabilistic story of logistic regression.

Example 6.1: $\boldsymbol{x} = (1, 2)^\top$, $y = 1$, $\boldsymbol{w} = (0.5, -0.3)^\top$, $b = 0$.
$z = -0.1$, $\hat{p} = \sigma(-0.1) \approx 0.475$.
$\nabla_{\boldsymbol{w}} \mathcal{L} = (0.475 - 1)(1, 2)^\top = (-0.525, -1.05)^\top$. The gradient pushes weights to increase z, correcting the misclassification.

6.5 Softmax and Multiclass Classification

$\text{softmax}(\boldsymbol{z})_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$. Converts scores into a probability distribution over K classes.
Multiclass cross-entropy: $\mathcal{L} = -\frac{1}{m} \sum_i \sum_k y_{ik} \log \hat{p}_{ik}$.
Gradient: $\nabla_{\boldsymbol{z}} \mathcal{L} = \hat{\boldsymbol{p}} - \boldsymbol{y}$ (predicted − true distribution).

Temperature scaling: $\text{softmax}(\boldsymbol{z}/T)$ with $T \to 0$ approaches hard argmax; $T \to \infty$ approaches uniform. Used in knowledge distillation and language model sampling.

6.6 Evaluation Metrics for Classification

Predicted probabilities matter, but evaluation also depends on the mistakes the classifier makes after we choose a threshold. For binary classification, those mistakes are summarized by the **confusion matrix**.

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Metric	Formula	What it tells you
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$	Overall fraction of correct predictions. It can look strong even when the model misses many positives in an imbalanced dataset.
Precision	$\frac{TP}{TP+FP}$	Of the predicted positives, how many were truly positive? Use it when false positives are costly.
Recall / True Positive Rate (TPR)	$\frac{TP}{TP+FN}$	Of the true positives, how many did we find? Use it when false negatives are costly.
Specificity / True Negative Rate (TNR)	$\frac{TN}{TN+FP}$	Of the true negatives, how many did we reject correctly?
F1 Score	$\frac{2TP}{2TP+FP+FN}$	A single number that balances precision and recall at one threshold.
Balanced Accuracy	$\frac{1}{2}\left(\frac{TP}{TP+FN}+\frac{TN}{TN+FP}\right)$	Average of recall and specificity, useful when one class is much rarer than the other.

Threshold choice matters. Most thresholded metrics change when we move the cutoff away from 0.5. Lowering the threshold usually raises recall and the false positive rate. Raising it usually raises precision and lowers recall.

Threshold move	Typical effect
Lower the threshold	More examples become positive. Recall usually rises, precision often falls, and the false positive rate rises.
Raise the threshold	Fewer examples become positive. Precision often rises, recall usually falls, and the false positive rate falls.

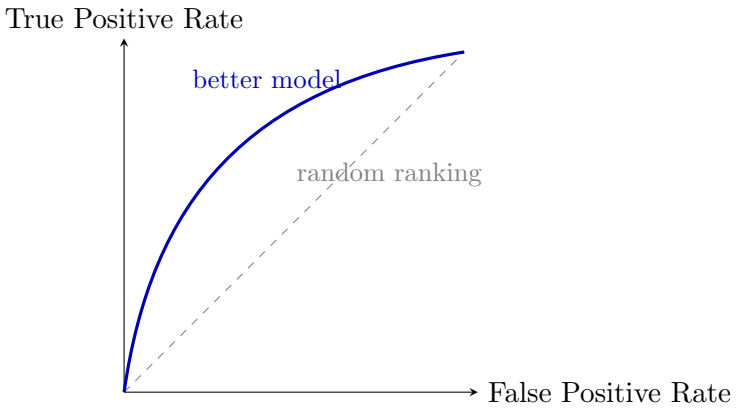


Figure 6.3: The **Receiver Operating Characteristic (ROC)** curve plots the true positive rate against the false positive rate, where $\text{FPR} = \frac{FP}{FP+TN}$, as we sweep the threshold. The **area under the curve (AUC)** summarizes this plot in one number: larger is better, and 0.5 matches random ranking.

Curve or Metric	**What it summarizes**	**When it is especially useful**
ROC and ROC-AUC	Trade-off between recall and false positive rate across all thresholds.	Useful when ranking quality matters and class frequencies are not extremely skewed.
Precision-Recall Curve and PR-AUC	Trade-off between precision and recall across all thresholds.	Often more informative when the positive class is rare.
Log Loss	Uses the full predicted probability, not just the hard label. Overconfident wrong predictions are penalized heavily.	Useful when probability calibration matters, not just ranking or hard classification.

Quick Check 6.2. A medical screening model should miss as few sick patients as possible. Which metric deserves the most attention first?

Answer. Recall should come first, because recall measures how many truly positive cases the model finds. In this setting, a false negative can be much more costly than a false positive, so we usually accept some loss in precision to keep recall high.

Practice Questions & Answers

Conceptual Questions

Q6.1. Derive the gradient of the binary cross-entropy loss with respect to the weights $\boldsymbol{w}$ for logistic regression. Why is the result $(\hat{p} - y)\boldsymbol{x}$ considered "elegant"?

Answer. The binary cross-entropy loss for one sample is:

$$\ell = -\big[y \log \sigma(z) + (1-y)\log(1-\sigma(z))\big], \quad z = \boldsymbol{w}^\top \boldsymbol{x}.$$

Using $\sigma'(z) = \sigma(z)(1-\sigma(z))$ and letting $\hat{p} = \sigma(z)$:

$$\frac{\partial \ell}{\partial z} = -\Big[\frac{y}{\hat{p}} - \frac{1-y}{1-\hat{p}}\Big]\hat{p}(1-\hat{p}) = -\big[y(1-\hat{p}) - (1-y)\hat{p}\big] = \hat{p} - y.$$

By the chain rule: $\nabla_{\boldsymbol{w}} \ell = (\hat{p} - y)\boldsymbol{x}$.
This is "elegant" because it has the *same form* as the gradient for linear regression with MSE loss: (prediction − target) × input. This arises because both are members of the **exponential family**, where MLE gradients share this canonical form.

Q6.2. What is the decision boundary of logistic regression? Why can logistic regression never learn the XOR function?

Answer. Logistic regression predicts class 1 when $\sigma(\boldsymbol{w}^\top \boldsymbol{x}) \geq 0.5$, i.e. when $\boldsymbol{w}^\top \boldsymbol{x} \geq 0$. The decision boundary is the hyperplane $\boldsymbol{w}^\top \boldsymbol{x} = 0$, which is linear in the input space.
XOR is not linearly separable: the four points $(0,0) \to 0$, $(0,1) \to 1$, $(1,0) \to 1$, $(1,1) \to 0$ cannot be separated by a single line. Since logistic regression can only produce linear boundaries, it fundamentally cannot learn XOR. This was a key motivator for multi-layer neural networks, which compose multiple linear boundaries with nonlinear activations to carve out arbitrary decision regions.

Q6.3. Explain the relationship between sigmoid and softmax. How does binary logistic regression generalise to the multiclass case?

Answer. The sigmoid function is a special case of softmax for $K = 2$ classes. For binary classification with logits z_0, z_1:

$$\text{softmax}(z_1) = \frac{e^{z_1}}{e^{z_0} + e^{z_1}} = \frac{1}{1 + e^{-(z_1 - z_0)}} = \sigma(z_1 - z_0).$$

So sigmoid operates on the *difference* of two logits.
For $K > 2$ classes, multiclass logistic regression (multinomial logistic regression) uses:

$$P(y = k|\boldsymbol{x}) = \frac{e^{\boldsymbol{w}_k^\top \boldsymbol{x}}}{\sum_{j=1}^{K} e^{\boldsymbol{w}_j^\top \boldsymbol{x}}} = \text{softmax}_k(\boldsymbol{z}).$$

The loss becomes the multiclass cross-entropy $\mathcal{L} = -\sum_k y_k \log \hat{p}_k$, and the gradient retains the elegant form $\nabla_{\boldsymbol{z}} \mathcal{L} = \hat{\boldsymbol{p}} - \boldsymbol{y}$ (predicted probability vector minus one-hot target).

Multiple Choice

Q6.4. The sigmoid function $\sigma(z) = 1/(1 + e^{-z})$ has which of the following properties?
(a) $\sigma(0) = 0$ and $\sigma'(0) = 0.5$
(b) $\sigma(0) = 0.5$ and $\sigma'(0) = 0.25$
(c) $\sigma(0) = 0.5$ and $\sigma'(0) = 0.5$
(d) $\sigma(0) = 1$ and $\sigma'(0) = 0.25$

Answer: (b). $\sigma(0) = 1/(1 + 1) = 0.5$. The derivative is $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, so $\sigma'(0) = 0.5 \times 0.5 = 0.25$. This maximum derivative of 0.25 is one reason deep sigmoid networks suffer from vanishing gradients—each layer multiplies gradients by at most 0.25.

Q6.5. Why is cross-entropy preferred over MSE as the loss function for logistic regression?
(a) Cross-entropy is always smaller than MSE
(b) MSE gradients vanish when the prediction is confidently wrong; cross-entropy gradients remain large

(c) Cross-entropy is convex while MSE is always non-convex
(d) Cross-entropy does not require computing the sigmoid function

Answer: (b). When the model predicts $\hat{p} \approx 0$ but the true label is $y = 1$, the MSE gradient involves $\sigma'(z)$ which is near zero for large $|z|$, making learning very slow. Cross-entropy's gradient is $(\hat{p} - y)\boldsymbol{x}$, which remains large for confident wrong predictions, providing strong corrective signal. Additionally, cross-entropy with sigmoid outputs is convex in the weights, while MSE with sigmoid is not.

Q6.6. In softmax regression with K classes, how many independent parameter vectors are needed?
(a) K
(b) $K - 1$
(c) $K + 1$
(d) K^2

Answer: (b). Since softmax probabilities must sum to 1, only $K - 1$ weight vectors are independently identifiable. Adding a constant vector to all $\boldsymbol{w}_k$ does not change the softmax output. In practice, we often parameterise all K vectors and rely on regularisation to break the degeneracy, but the model has $K - 1$ degrees of freedom.

Computational Problems

Q6.7. A logistic regression model has weights $\boldsymbol{w} = (0.5, -1.2, 0.3)^\top$ (including bias as w_0). For the input $\boldsymbol{x} = (1, 2, -1)^\top$ (with $x_0 = 1$ for bias), compute the predicted probability and classify with threshold 0.5.

Solution.

$$z = \boldsymbol{w}^\top \boldsymbol{x} = 0.5(1) + (-1.2)(2) + 0.3(-1) = 0.5 - 2.4 - 0.3 = -2.2.$$

$$\hat{p} = \sigma(-2.2) = \frac{1}{1 + e^{2.2}} = \frac{1}{1 + 9.025} \approx 0.0998.$$

Since $\hat{p} = 0.10 < 0.5$, the model predicts class 0. The cross-entropy loss if the true label is $y = 1$: $\ell = -\log(0.0998) \approx 2.31$ (a large penalty for being confidently wrong).

Q6.8. Given logits $\boldsymbol{z} = (2.0, 1.0, 0.1)$ for a 3-class problem, compute the softmax probabilities. Then compute the cross-entropy loss if the true class is $k = 0$.

Solution.

$$e^{z_0} = e^{2.0} \approx 7.389, \quad e^{z_1} = e^{1.0} \approx 2.718, \quad e^{z_2} = e^{0.1} \approx 1.105.$$

$$\text{Sum} = 7.389 + 2.718 + 1.105 = 11.212.$$

$$\hat{p}_0 = \frac{7.389}{11.212} \approx 0.659, \quad \hat{p}_1 = \frac{2.718}{11.212} \approx 0.242, \quad \hat{p}_2 = \frac{1.105}{11.212} \approx 0.099.$$

Cross-entropy loss with true class $k = 0$ (one-hot $\boldsymbol{y} = (1, 0, 0)$):

$$\mathcal{L} = -\log(\hat{p}_0) = -\log(0.659) \approx 0.417.$$

The gradient w.r.t. the logits is $\hat{\boldsymbol{p}} - \boldsymbol{y} = (0.659 - 1,\ 0.242 - 0,\ 0.099 - 0) = (-0.341,\ 0.242,\ 0.099)$.

Q6.9. Show that the decision boundary of logistic regression is linear by finding the equation of the boundary for $\boldsymbol{w} = (w_0, w_1, w_2) = (-3, 1, 1)$.

Solution. The decision boundary is where $\hat{p} = 0.5$, i.e. $\sigma(z) = 0.5$, which requires $z = 0$:

$$w_0 + w_1 x_1 + w_2 x_2 = 0 \quad \Longrightarrow \quad -3 + x_1 + x_2 = 0 \quad \Longrightarrow \quad x_2 = 3 - x_1.$$

This is a straight line in (x_1, x_2) space with slope -1 and intercept 3. Points above this line ($x_1 + x_2 > 3$) are classified as class 1; points below as class 0. The weight vector $(w_1, w_2) = (1, 1)$ is the normal to the boundary, pointing toward the positive class.

Interview-Style Questions

Q6.10. *A stakeholder asks you to use logistic regression for a problem with 50 classes and highly overlapping clusters. What concerns would you raise, and what alternatives would you suggest?*

Model Answer. Key concerns with multinomial logistic regression here:

1. **Linear boundaries only:** With 50 overlapping classes, linear decision boundaries are almost certainly insufficient. The model will misclassify many points in overlap regions.
2. **Parameter explosion:** 50 classes $\times$ d features $= 50d$ parameters. With limited data, overfitting is a serious risk.
3. **No feature interactions:** Logistic regression cannot capture nonlinear relationships unless features are manually engineered.

Alternatives:

- **Neural network with softmax output:** Hidden layers learn nonlinear feature representations that can separate overlapping clusters.
- **Kernel SVM (one-vs-rest):** Can learn nonlinear boundaries via the kernel trick, but scales poorly with 50 classes.
- **Tree-based models:** Random forests or gradient boosting (XGBoost) handle multiclass naturally and capture nonlinearity.
- **Logistic regression with features:** If interpretability is paramount, engineer polynomial or interaction features and use regularised softmax regression.

Q6.11. *Explain why the output of logistic regression is a calibrated probability. When might it not be well-calibrated?*

Model Answer. Logistic regression is trained by maximising the likelihood $P(\boldsymbol{y}|X; \boldsymbol{w})$, directly modelling the conditional probability $P(y = 1|\boldsymbol{x})$. Under the assumption that the true decision boundary is linear and the log-odds are a linear function of $\boldsymbol{x}$, the output $\hat{p} = \sigma(\boldsymbol{w}^\top \boldsymbol{x})$ is a well-calibrated probability—i.e., among all predictions of $\hat{p} = 0.7$, approximately 70% are truly positive.

Calibration degrades when:

- **Model misspecification:** The true boundary is nonlinear (e.g., quadratic), so the linear assumption is violated.
- **Dataset shift:** Training and test distributions differ, invalidating the learned relation-

ship.

- **Strong regularisation:** Excessive L^2 penalty shrinks logits toward zero, compressing probabilities toward 0.5.
- **Class imbalance:** With severe imbalance, the model may be overconfident for the majority class.

Post-hoc calibration methods (Platt scaling, isotonic regression) can recalibrate any classifier's outputs.

Python Visualisation

Listing 6.1: Logistic regression decision boundary and sigmoid curve

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression

X, y = make_classification(n_samples=200, n_features=2,
    n_redundant=0, n_clusters_per_class=1,
    random_state=42, class_sep=1.5)

clf = LogisticRegression()
clf.fit(X, y)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(13, 5))

# Left: Decision boundary
xx, yy = np.meshgrid(np.linspace(X[:,0].min()-1,
    X[:,0].max()+1, 300),
    np.linspace(X[:,1].min()-1, X[:,1].max()+1, 300))
Z = clf.predict_proba(
    np.c_[xx.ravel(), yy.ravel()])[:,1].reshape(xx.shape)
ax1.contourf(xx, yy, Z, levels=50, cmap='RdBu_r', alpha=0.7)
ax1.contour(xx, yy, Z, levels=[0.5], colors='k', linewidths=2)
ax1.scatter(X[:,0], X[:,1], c=y, cmap='RdBu_r',
            edgecolors='k', s=30)
ax1.set_title('Decision Boundary (P(y=1|x))')
ax1.set_xlabel('$x_1$'); ax1.set_ylabel('$x_2$')

# Right: Sigmoid curve
z = np.linspace(-6, 6, 200)
ax2.plot(z, 1/(1+np.exp(-z)), 'b-', lw=2)
ax2.axhline(0.5, color='gray', ls='--', lw=1)
ax2.axvline(0, color='gray', ls='--', lw=1)
ax2.fill_between(z, 1/(1+np.exp(-z)), 0.5,
    where=(1/(1+np.exp(-z))>0.5), alpha=0.2, color='blue')
ax2.fill_between(z, 1/(1+np.exp(-z)), 0.5,
    where=(1/(1+np.exp(-z))<0.5), alpha=0.2, color='red')
ax2.set_xlabel('$z = w^T x$'); ax2.set_ylabel('$\sigma(z)$')
ax2.set_title('Sigmoid Function')

plt.tight_layout()
plt.savefig('ch6_logistic_boundary.pdf')
```

```
plt.show()
```

Key Takeaways

- Sigmoid squashes linear scores to probabilities; decision boundary is a hyperplane.
- Cross-entropy = negative log-likelihood. Penalizes confident wrong predictions severely.
- Gradient: $(\hat{p} - y)\boldsymbol{x}$ — same elegant form as linear regression.
- Softmax + cross-entropy is the standard for multiclass classification.
- Logistic regression can only learn linear boundaries—nonlinear requires neural networks.

Chapter 7

Loss Functions and Regularization

7.1 Introduction

Every ML model is defined by two choices: (1) the **model architecture** and (2) the **loss function**. Regularization modifies the loss to prevent overfitting.

7.2 Regression Losses

MSE: $\frac{1}{m}\sum(y_i - \hat{y}_i)^2$. Quadratic penalty. Sensitive to outliers. Gaussian noise.
MAE: $\frac{1}{m}\sum|y_i - \hat{y}_i|$. Linear penalty. Robust to outliers. Laplace noise.
Huber: Quadratic for small errors, linear for large. Best of both worlds.

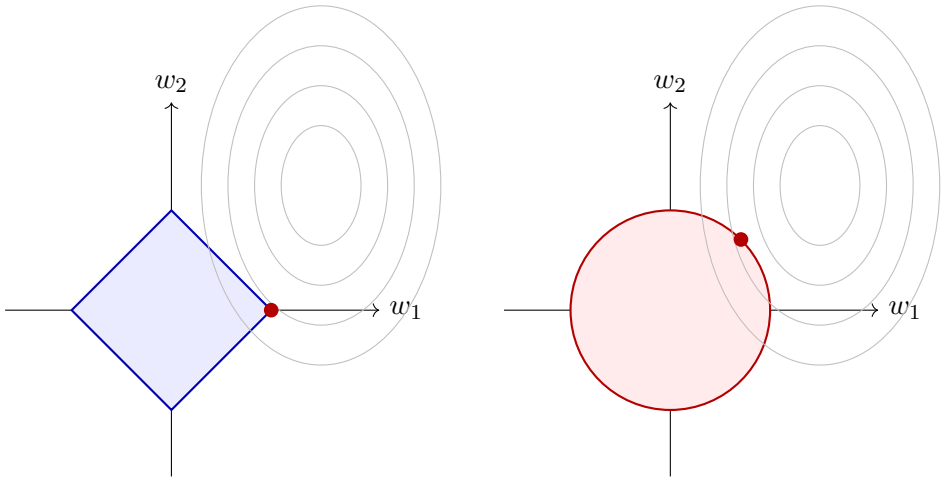
$$L_\delta(e) = \begin{cases} \frac{1}{2}e^2 & |e| \le \delta \\ \delta(|e| - \delta/2) & |e| > \delta \end{cases}$$

7.3 Classification Losses

Cross-Entropy: $-\frac{1}{m}\sum_i \sum_k y_{ik} \log \hat{p}_{ik}$. Standard for probabilistic classifiers.
Hinge Loss: $\frac{1}{m}\sum \max(0, 1 - y_i f(\boldsymbol{x}_i))$. Used in SVMs. Maximum margin.
Focal Loss: $-\alpha_t(1 - \hat{p}_t)^\gamma \log \hat{p}_t$. Down-weights easy examples; essential for class-imbalanced problems.

7.4 Regularization

$\mathcal{L}_{\text{reg}} = \mathcal{L}_{\text{data}} + \lambda\,\Omega(\boldsymbol{w})$
L^2 **(Ridge):** $\Omega = \frac{1}{2}\|\boldsymbol{w}\|_2^2$. Shrinks all weights. Gaussian prior.
L^1 **(Lasso):** $\Omega = \|\boldsymbol{w}\|_1$. Drives weights to zero (sparsity). Laplace prior.
Elastic Net: $\Omega = \alpha\|\boldsymbol{w}\|_1 + \frac{1-\alpha}{2}\|\boldsymbol{w}\|_2^2$. Combines both.



L^1: hits corner → sparse L^2: smooth → not sparse

Figure 7.1: L^1 diamond corners sit on axes, producing sparse solutions. L^2 circle has no corners, so tangent points have all weights nonzero.

7.5 Dropout

Dropout randomly sets each neuron's output to zero with probability p during training. At test time, outputs are scaled by $(1-p)$.

Three perspectives on why dropout works:

1. **Ensemble:** Implicitly trains 2^n subnetworks; the full network averages them.
2. **Co-adaptation prevention:** No neuron can rely on any specific other neuron.
3. **Noise regularization:** Multiplicative noise on activations $\approx L^2$ regularization for linear models.

7.6 Early Stopping and Batch Normalization

Early stopping: Stop training when validation loss starts increasing. "Free" regularization—equivalent to L^2 regularization for quadratic losses.
Batch normalization: Normalizes layer inputs to zero mean and unit variance within each mini-batch: $\hat{x} = \frac{x-\mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2+\epsilon}}$, then scale and shift with learnable γ, β.

BatchNorm benefits: Allows higher learning rates, reduces sensitivity to initialization, acts as a mild regularizer, smooths the loss landscape.

Practice Questions & Answers

Conceptual Questions

Q7.1. Compare MSE, MAE, and Huber loss for regression. How does each handle outliers, and what distributional assumptions does each encode?

Answer.

- **MSE** $= \frac{1}{m}\sum(y_i - \hat{y}_i)^2$: Corresponds to MLE under Gaussian noise. Squares amplify large residuals, making it very sensitive to outliers. Gradient is $2(\hat{y} - y)$, proportional to error magnitude.
- **MAE** $= \frac{1}{m}\sum|y_i - \hat{y}_i|$: Corresponds to MLE under Laplacian noise. Treats all errors linearly, making it *robust* to outliers. Gradient is ± 1 (constant magnitude), so it does not give extra signal for large errors. Non-differentiable at zero.
- **Huber loss**: Piecewise—quadratic for $|e| \leq \delta$, linear for $|e| > \delta$:

$$L_\delta(e) = \begin{cases} \frac{1}{2}e^2 & |e| \leq \delta \\ \delta(|e| - \frac{\delta}{2}) & |e| > \delta \end{cases}$$

  Combines MSE's smoothness near zero with MAE's outlier robustness. The threshold δ controls the transition. Differentiable everywhere.

Rule of thumb: Use MSE for clean Gaussian data. Use Huber when outliers are present but you still want smooth gradients near zero. Use MAE for heavy-tailed noise or when median prediction is desired.

Q7.2. Explain L^1 vs. L^2 regularisation from three perspectives: geometric, Bayesian, and effect on the loss landscape.

Answer.

Geometric: The constraint region for L^2 is a smooth hypersphere $\|\boldsymbol{w}\|_2 \leq t$. The MSE contours (ellipses) are most likely to touch this ball at a point where all weights are small but nonzero. The L^1 region is a hypercube (diamond) with corners on the axes—contours often touch at a corner, setting some weights exactly to zero.

Bayesian: L^2 regularisation is equivalent to placing a Gaussian prior $w_j \sim \mathcal{N}(0, \tau^2)$ on each weight (MAP estimation). L^1 corresponds to a Laplace prior $w_j \sim \text{Laplace}(0, b)$, which has heavier tails but a sharp peak at zero, encouraging sparsity.

Loss landscape: L^2 adds a smooth bowl ($\lambda\|\boldsymbol{w}\|_2^2$) to the loss, moving the minimum toward zero but preserving smoothness. L^1 adds a "tent" ($\lambda\|\boldsymbol{w}\|_1$) with sharp ridges along the axes, creating kinks where the subgradient can push weights to exactly zero.

Q7.3. How does dropout work as a regulariser? Explain the training-time and inference-time behaviour, and the connection to ensemble learning.

Answer.
Training: At each forward pass, each neuron is independently "dropped" (set to zero) with probability p (typically $p = 0.5$ for hidden layers). This forces the network to not rely on any single neuron and learn redundant representations.
Inference: No neurons are dropped, but all weights are scaled by $(1 - p)$ to compensate for the fact that more neurons are active ("inverted dropout" scales during training instead: divide activations by $(1 - p)$).
Ensemble interpretation: Each dropout mask defines a different sub-network. With n neurons, there are 2^n possible sub-networks. Training with dropout approximately trains all 2^n sub-networks simultaneously with shared weights. At inference, the full network approximates the geometric mean of all sub-network predictions—an ensemble. This is why dropout reduces variance (overfitting) at the cost of slightly increased bias.
Key property: Dropout prevents *co-adaptation*—neurons cannot rely on specific other neurons being present, forcing each to be independently useful.

Multiple Choice

Q7.4. Which loss function is most robust to outliers?
(a) Mean Squared Error (MSE)
(b) Mean Absolute Error (MAE)
(c) Huber loss with $\delta = 100$
(d) Squared Huber loss

Answer: (b). MAE grows linearly with the residual, giving outliers bounded influence on the gradient (± 1). MSE grows quadratically, amplifying outlier influence. Huber with $\delta = 100$ behaves like MSE for residuals up to 100, offering little outlier protection. MAE provides the strongest robustness among these options.

Q7.5. Early stopping is often described as "free regularisation" because:
(a) It requires no additional hyperparameters
(b) It is equivalent to L^2 regularisation for quadratic losses, and adds no computational cost beyond monitoring validation loss
(c) It eliminates the need for a validation set
(d) It makes the model converge faster

Answer: (b). Early stopping halts training when validation loss starts increasing. For quadratic loss functions, Bishop (1995) showed this is equivalent to L^2 regularisation, where the effective λ is inversely related to the number of training steps. It is "free" because it requires only monitoring the validation loss (which you should do anyway), adding no extra computation. It does require a patience hyperparameter and a validation set (so (a) and (c) are incorrect).

Q7.6. Batch normalisation normalises activations to zero mean and unit variance. The learnable parameters γ and β are included because:
(a) They prevent numerical overflow in the exponential function
(b) They allow the network to undo the normalisation if that is optimal

(c) They replace the bias term in each layer
(d) They are needed only during inference, not training

Answer: (b). Without γ and β, normalisation forces every layer's output to have mean 0 and variance 1, which may be too restrictive. The learnable scale (γ) and shift (β) parameters allow the network to recover any mean and variance if that is beneficial, including the identity transformation $\gamma = \sigma_{\mathcal{B}}, \beta = \mu_{\mathcal{B}}$ that completely undoes the normalisation. This ensures BatchNorm never reduces the model's representational capacity.

Computational Problems

Q7.7. Given true values $\boldsymbol{y} = (3, -0.5, 2, 7)$ and predictions $\hat{\boldsymbol{y}} = (2.5, 0.0, 2, 8)$, compute the MSE, MAE, and Huber loss with $\delta = 1.0$.

Solution. Residuals: $\boldsymbol{e} = \boldsymbol{y} - \hat{\boldsymbol{y}} = (0.5, -0.5, 0, -1)$.

$$\text{MSE} = \frac{1}{4}(0.25 + 0.25 + 0 + 1) = \frac{1.5}{4} = 0.375$$

$$\text{MAE} = \frac{1}{4}(0.5 + 0.5 + 0 + 1) = \frac{2.0}{4} = 0.5$$

$$\begin{aligned}\text{Huber}(\delta{=}1): \quad & L(0.5) = \tfrac{1}{2}(0.5)^2 = 0.125 \quad \text{(quadratic, since } |0.5| \le 1) \\ & L(-0.5) = 0.125 \\ & L(0) = 0 \\ & L(-1) = 1 \cdot (1 - 0.5) = 0.5 \quad \text{(linear, since } |-1| = 1 = \delta)\end{aligned}$$

$$\text{Huber} = \frac{1}{4}(0.125 + 0.125 + 0 + 0.5) = \frac{0.75}{4} = 0.1875$$

Note: MSE > Huber > MAE is not always the ordering; it depends on the residual distribution relative to δ.

Q7.8. A model has weights $\boldsymbol{w} = (3, -1, 0.5, 0, 2)$. Compute the L^1 and L^2 penalty terms. If $\lambda = 0.1$, what is the total regularised MSE loss if the unregularised MSE is 2.5?

Solution.

$$\|\boldsymbol{w}\|_1 = |3| + |-1| + |0.5| + |0| + |2| = 6.5$$

$$\|\boldsymbol{w}\|_2^2 = 9 + 1 + 0.25 + 0 + 4 = 14.25$$

$$\text{Lasso loss} = \text{MSE} + \lambda\|\boldsymbol{w}\|_1 = 2.5 + 0.1(6.5) = 3.15$$

$$\text{Ridge loss} = \text{MSE} + \lambda\|\boldsymbol{w}\|_2^2 = 2.5 + 0.1(14.25) = 3.925$$

Ridge penalises the large weight $w_1 = 3$ much more severely ($9 \times 0.1 = 0.9$ contribution) than Lasso ($3 \times 0.1 = 0.3$), because squaring amplifies large values.

Q7.9. In a dropout layer with $p = 0.3$ (drop probability) applied to a hidden layer with activations $\boldsymbol{h} = (1.0, 2.0, -0.5, 3.0)$, what is the expected value of each activation during

training? What scaling is applied at test time (or equivalently, during training with inverted dropout)?

Solution. During training, each activation is zeroed with probability $p = 0.3$ and kept with probability $1 - p = 0.7$:

$$\mathbb{E}[\tilde{h}_j] = (1 - p) \cdot h_j = 0.7 \cdot h_j.$$

So: $\mathbb{E}[\tilde{\boldsymbol{h}}] = (0.7, 1.4, -0.35, 2.1)$.

Inverted dropout (used in practice): During training, divide surviving activations by $(1-p) = 0.7$, so a kept activation becomes $h_j/0.7$. This ensures $\mathbb{E}[\tilde{h}_j] = h_j$, and no scaling is needed at test time.

Standard dropout: Keep activations unchanged during training; at test time, multiply all weights by $(1 - p) = 0.7$. Both approaches are mathematically equivalent.

Interview-Style Questions

Q7.10. *You notice your neural network's training loss keeps decreasing but the validation loss starts increasing after epoch 20. What regularisation strategies would you apply, and in what order?*

Model Answer. This is textbook overfitting. I would apply regularisation in order of simplicity and cost:

1. **Early stopping** (immediate, free): Stop at epoch 20 (or wherever validation loss was minimised). Save the best model checkpoint.
2. **Weight decay (L^2):** Add $\lambda\|\boldsymbol{w}\|_2^2$ to the loss. Equivalent to decaying weights by a factor each step. Tune λ via cross-validation.
3. **Dropout:** Add dropout layers (typically $p = 0.2$–0.5) after dense layers. Prevents co-adaptation.
4. **Data augmentation:** If applicable (e.g., images), augment training data with flips, rotations, crops. Increases effective dataset size.
5. **Reduce model size:** Fewer layers or narrower layers reduce capacity and thus variance.
6. **Batch normalisation:** Acts as a mild regulariser due to mini-batch noise in the mean/-variance estimates.

I would *not* apply all at once. Start with early stopping + weight decay, evaluate, then add dropout if needed. Monitor the train-val gap to diagnose progress.

Q7.11. *A colleague argues that batch normalisation makes dropout unnecessary. Do you agree? Explain.*

Model Answer. Partially agree, but with nuances:
The argument: BatchNorm provides mild regularisation because the normalisation statistics (mean, variance) are computed per mini-batch, injecting noise that depends on which samples are in the batch. This noise acts similarly to dropout's stochastic masking.
Why it's not a full replacement:

- BatchNorm's regularisation effect is *weak* compared to dropout. For highly overparam-

eterised models, dropout may still be necessary.

- They regularise differently: dropout forces redundant representations (each neuron must be useful alone), while BatchNorm smooths the loss landscape and stabilises training.
- Empirically, combining BatchNorm and dropout can be tricky—dropout changes the variance of inputs to BatchNorm, causing a "variance shift" at test time. If used together, apply dropout *after* BatchNorm, or use smaller drop rates.

Practice: Modern architectures (ResNets, Transformers) often use BatchNorm or LayerNorm *without* dropout in convolutional/attention layers, but may still use dropout in final fully-connected layers. The best strategy depends on the architecture and dataset size.

Python Visualisation

Listing 7.1: Comparing MSE, MAE, and Huber loss functions and their gradients

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

residuals = np.linspace(-4, 4, 500)
delta = 1.0

mse = residuals**2
mae = np.abs(residuals)
huber = np.where(np.abs(residuals) <= delta,
                 0.5 * residuals**2,
                 delta * (np.abs(residuals) - 0.5 * delta))

grad_mse = 2 * residuals
grad_mae = np.sign(residuals)
grad_huber = np.where(np.abs(residuals) <= delta,
                      residuals,
                      delta * np.sign(residuals))

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(13, 5))

for loss, name, color, ls in [
    (mse, 'MSE', 'blue', '-'),
    (mae, 'MAE', 'red', '--'),
    (huber, f'Huber ($\\delta$={delta})', 'green', '-.')]:
    ax1.plot(residuals, loss, color=color, ls=ls,
             lw=2, label=name)
ax1.set_xlabel('Residual ($y - \\hat{y}$)')
ax1.set_ylabel('Loss')
ax1.set_title('Loss Functions')
ax1.legend(); ax1.set_ylim(-0.5, 10); ax1.grid(alpha=0.3)

for grad, name, color, ls in [
    (grad_mse, 'MSE', 'blue', '-'),
    (grad_mae, 'MAE', 'red', '--'),
    (grad_huber, f'Huber ($\\delta$={delta})', 'green', '-.')]:
    ax2.plot(residuals, grad, color=color, ls=ls,
             lw=2, label=name)
ax2.set_xlabel('Residual ($y - \\hat{y}$)')
```

```
ax2.set_ylabel('Gradient')
ax2.set_title('Gradients of Loss Functions')
ax2.legend(); ax2.grid(alpha=0.3)

plt.tight_layout()
plt.savefig('ch7_loss_functions.pdf')
plt.show()
```

Key Takeaways

- MSE for regression, cross-entropy for classification. Huber for outlier-robust regression.
- L^2 shrinks weights (ridge); L^1 zeros out weights (lasso). Regularization = prior belief.
- Dropout prevents co-adaptation; early stopping is free regularization.
- Batch normalization smooths the loss landscape and enables higher learning rates.
- The choice of loss function encodes your assumptions about the data distribution.

Part III

Neural Networks

Chapter 8

Neural Networks

8.1 Introduction

Neural networks are compositions of linear transformations and nonlinear activation functions. This simple recipe—linear, nonlinear, linear, nonlinear, repeat—can approximate *any* continuous function to arbitrary accuracy. This chapter explains why that works, how the architecture is designed, and what each component does.

8.2 From Linear Models to Neural Networks

Linear models are powerful but limited: they can only represent linear decision boundaries. XOR is the classic example—no single line can separate the two classes.

A **perceptron** computes $\hat{y} = \text{step}(\boldsymbol{w}^\top \boldsymbol{x} + b)$, where $\text{step}(z) = 1$ if $z \geq 0$ else 0. It can learn any linearly separable function (AND, OR) but **cannot learn XOR**.

The XOR Problem: XOR outputs 1 when inputs differ: $(0,0) \to 0$, $(0,1) \to 1$, $(1,0) \to 1$, $(1,1) \to 0$. No single line in 2D can separate the 1s from the 0s. This was Minsky and Papert's famous 1969 critique that nearly killed neural network research for a decade. The solution? Add a hidden layer.

8.3 Multi-Layer Networks

A **feedforward neural network** with L layers computes:

$$\boldsymbol{z}^{(\ell)} = W^{(\ell)} \boldsymbol{a}^{(\ell-1)} + \boldsymbol{b}^{(\ell)} \quad \text{(linear transformation)}$$
$$\boldsymbol{a}^{(\ell)} = g(\boldsymbol{z}^{(\ell)}) \quad \text{(nonlinear activation)}$$

where $\boldsymbol{a}^{(0)} = \boldsymbol{x}$ (the input), $W^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ is the weight matrix, and g is the activation function.

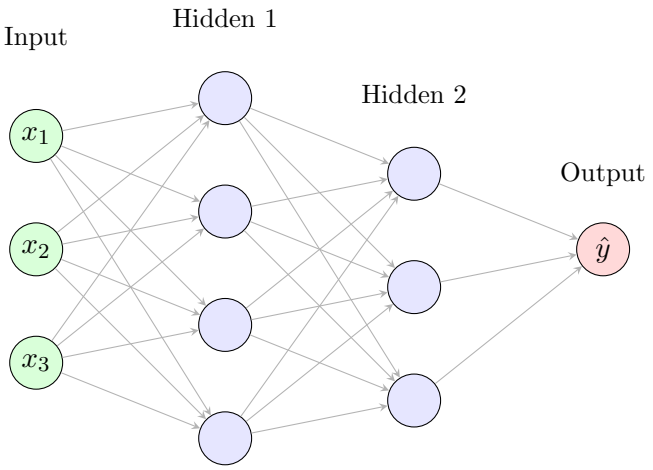


Figure 8.1: A feedforward neural network with 3 inputs, two hidden layers (4 and 3 neurons), and 1 output. Each arrow is a learnable weight.

What each layer does: Each layer applies a linear transformation (rotate and scale) followed by a nonlinearity (warp). The first hidden layer learns simple features (edges, thresholds). Deeper layers compose these into increasingly abstract representations. The output layer maps the final representation to the prediction.

8.4 Activation Functions

Without nonlinear activations, a multi-layer network collapses to a single linear transformation: $W_2(W_1\boldsymbol{x}) = (W_2W_1)\boldsymbol{x} = W'\boldsymbol{x}$. Nonlinearity is what gives depth its power.

ReLU (Rectified Linear Unit): $g(z) = \max(0, z)$.

- Derivative: $g'(z) = 1$ for $z > 0$, 0 for $z < 0$. No vanishing gradient for positive inputs!
- Simple, fast, sparse activations. Dominates modern deep learning.
- **Dying ReLU problem:** If $z < 0$ for all inputs, the gradient is zero and the neuron never updates. Fix: Leaky ReLU ($\max(0.01z, z)$) or parametric PReLU.

Sigmoid: $\sigma(z) = 1/(1 + e^{-z})$. Range $(0, 1)$. Used in output layer for binary classification. Vanishing gradient for $|z| \gg 0$.

Tanh: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Range $(-1, 1)$. Zero-centered (better than sigmoid for hidden layers). Still has vanishing gradient.

GELU: $z \cdot \Phi(z)$ where Φ is the Gaussian CDF. Smooth approximation to ReLU. Used in Transformers (GPT, BERT).

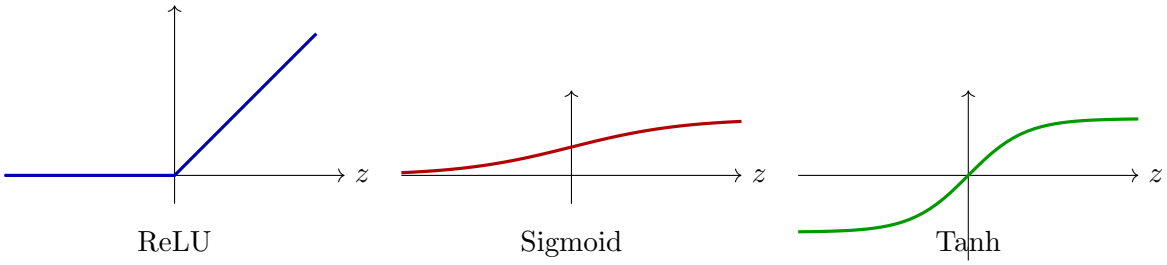


Figure 8.2: Common activation functions. ReLU is piecewise linear—simple and avoids vanishing gradients for $z > 0$.

8.5 The Universal Approximation Theorem

Theorem (Cybenko, 1989; Hornik, 1991): A feedforward network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to arbitrary accuracy, given a suitable activation function.

What this means and doesn't mean:

- **Does mean:** Neural networks are powerful enough to represent any function we'd want to learn. They are "universal function approximators."
- **Doesn't mean:** A single layer is *practical*. It may need exponentially many neurons. The theorem says nothing about whether gradient descent can *find* the right weights.
- **In practice: Deep** networks (many layers) are far more efficient than wide shallow ones. Depth enables **hierarchical feature learning**: early layers learn simple patterns, later layers compose them into complex ones. This is empirically observed: a 10-layer network can do what a 1-layer network with billions of neurons cannot.

8.6 Weight Initialization

Xavier/Glorot initialization (for sigmoid/tanh):

$$W_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

He initialization (for ReLU):

$$W_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$$

Why initialization matters: If weights are too large, activations explode (outputs saturate). If too small, activations vanish (all zeros). Proper initialization keeps the variance of activations roughly constant across layers, ensuring stable forward and backward passes.

8.7 Solving XOR

Example 8.1: A 2-layer ReLU network solves XOR.
$W^{(1)} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$, $\boldsymbol{b}^{(1)} = (0, -1)^\top$, $W^{(2)} = (1, -2)$, $b^{(2)} = 0$.
Neuron 1: $\text{ReLU}(x_1 + x_2)$ = "at least one is 1" (OR-like).
Neuron 2: $\text{ReLU}(x_1 + x_2 - 1)$ = "both are 1" (AND-like).
Output: $\text{ReLU}(\text{OR} - 2 \cdot \text{AND}) = \text{XOR}$!

Verify: $(0,0) \to \text{ReLU}(0-0) = 0$. $(1,0) \to \text{ReLU}(1-0) = 1$. $(0,1) \to \text{ReLU}(1-0) = 1$. $(1,1) \to \text{ReLU}(2-2) = 0$.

Practice Questions & Answers

Conceptual Questions

Q8.1: Why can't a single-layer perceptron solve the XOR problem? What is the minimum network architecture needed?

A8.1: A single-layer perceptron computes $\hat{y} = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$, which defines a single linear decision boundary (a hyperplane) in input space. XOR is not linearly separable: no single straight line can separate the $(0,0), (1,1)$ class from the $(1,0), (0,1)$ class. The minimum architecture is a two-layer network with at least 2 hidden neurons. Each hidden neuron learns one linear boundary, and the output neuron combines them. Geometrically, the hidden layer remaps the inputs into a new space where XOR *becomes* linearly separable.

Q8.2: Compare the ReLU, sigmoid, and tanh activation functions. When would you prefer each one?

A8.2: **Sigmoid** $\sigma(z) = 1/(1+e^{-z})$: outputs in $(0,1)$; useful for binary output probabilities. Suffers from vanishing gradients since $\sigma'(z) \leq 0.25$ everywhere, making deep training difficult. **Tanh** $\tanh(z)$: outputs in $(-1,1)$; zero-centered, which helps optimization converge faster than sigmoid. Still saturates at extremes. **ReLU** $\max(0,z)$: no saturation for $z > 0$, gradient is exactly 1, enabling fast training. Suffers from "dying ReLU" problem (neurons stuck at 0). Use ReLU (or variants like Leaky ReLU, GELU) as the default for hidden layers. Use sigmoid only for final-layer binary classification. Use tanh rarely, perhaps in RNN hidden states for historical reasons.

Q8.3: What does the Universal Approximation Theorem guarantee, and what does it *not* guarantee?

A8.3: The theorem states that a feedforward network with a single hidden layer containing enough neurons and a nonlinear activation can approximate any continuous function on a compact domain to arbitrary accuracy. What it does *not* guarantee: (1) how many neurons are needed—the required width may be exponentially large; (2) that gradient descent will find the right weights—the theorem is purely existential; (3) generalization—an approximating network may overfit badly. In practice, we use *deep narrow* networks instead of shallow wide ones because depth enables exponentially more efficient representations through compositionality.

Multiple Choice

Q8.4: A neural network has input dimension 5, one hidden layer with 10 neurons, and output dimension 3. How many learnable parameters does it have (including biases)?
(a) 80
(b) 83
(c) 93
(d) 95

Answer: (c) 93. First layer: $5 \times 10 + 10 = 60$ (weights + biases). Second layer: $10 \times 3 + 3 = 33$. Total: $60 + 33 = 93$.

Q8.5: Which activation function can cause the "dying neuron" problem where a neuron permanently outputs zero?
(a) Sigmoid
(b) Tanh
(c) ReLU
(d) Softmax

Answer: (c) ReLU. If a neuron's pre-activation becomes permanently negative (e.g., due to a large negative bias or a large gradient update), ReLU outputs 0 and its gradient is also 0, so the neuron can never recover. Leaky ReLU fixes this by allowing a small gradient for negative inputs.

Q8.6: What is the primary purpose of Xavier (Glorot) initialization?
(a) To make all weights equal
(b) To keep the variance of activations roughly constant across layers
(c) To ensure weights are always positive
(d) To minimize the initial loss value

Answer: (b) Keep the variance of activations roughly constant across layers. Xavier initialization sets $w \sim \mathcal{N}(0, 2/(n_{\text{in}} + n_{\text{out}}))$, which prevents signals from exploding or vanishing as they propagate through layers. For ReLU activations, He initialization (Var $= 2/n_{\text{in}}$) is preferred because ReLU zeroes out half the activations.

Computational Problems

Q8.7: A single neuron has weights $\boldsymbol{w} = (0.5, -0.3, 0.8)^\top$ and bias $b = -0.1$. Compute the output for input $\boldsymbol{x} = (1, 2, -1)^\top$ using: (a) sigmoid activation, (b) ReLU activation.

Solution:

$$z = \boldsymbol{w}^\top \boldsymbol{x} + b = 0.5(1) + (-0.3)(2) + 0.8(-1) + (-0.1) = 0.5 - 0.6 - 0.8 - 0.1 = -1.0$$

(a) Sigmoid: $\sigma(-1.0) = \dfrac{1}{1 + e^{1.0}} = \dfrac{1}{1 + 2.718} \approx 0.269$

(b) ReLU: $\max(0, -1.0) = 0$

Q8.8: A two-layer network has $W^{(1)} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$, $\boldsymbol{b}^{(1)} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$, $\boldsymbol{w}^{(2)} = (1, -2)^\top$, $b^{(2)} = 0$, with ReLU activations. Compute the output for $\boldsymbol{x} = (1, 0)^\top$.

Solution:

$$\boldsymbol{z}^{(1)} = W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\boldsymbol{h}^{(1)} = \text{ReLU}\left(\begin{pmatrix} 1 \\ 0 \end{pmatrix}\right) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$z^{(2)} = (\boldsymbol{w}^{(2)})^\top \boldsymbol{h}^{(1)} + b^{(2)} = 1(1) + (-2)(0) + 0 = 1$$

$$\hat{y} = \text{ReLU}(1) = 1$$

Q8.9: Write Python code to visualize different activation functions side by side.

Listing 8.1: Visualizing activation functions

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

z = np.linspace(-5, 5, 300)
sigmoid = 1 / (1 + np.exp(-z))
tanh = np.tanh(z)
relu = np.maximum(0, z)
leaky_relu = np.where(z > 0, z, 0.01 * z)

fig, axes = plt.subplots(1, 4, figsize=(14, 3))
for ax, func, name in zip(axes,
    [sigmoid, tanh, relu, leaky_relu],
    ['Sigmoid', 'Tanh', 'ReLU', 'Leaky ReLU']):
    ax.plot(z, func, 'b-', linewidth=2)
    ax.axhline(0, color='k', linewidth=0.5)
    ax.axvline(0, color='k', linewidth=0.5)
    ax.set_title(name, fontsize=12)
    ax.set_xlabel('z'); ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig('activation_functions.pdf')
plt.show()
```

Interview-Style Questions

Q8.10: *"Walk me through what happens inside a neural network from input to prediction. Assume a simple 2-layer network for classification."*

Model Answer: Starting with an input vector $\boldsymbol{x}$, the first layer computes $\boldsymbol{z}^{(1)} = W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$—a linear transformation that projects the input into a higher-dimensional space. We

then apply a nonlinear activation: $\boldsymbol{h}^{(1)} = \text{ReLU}(\boldsymbol{z}^{(1)})$, which introduces the ability to model complex boundaries. The second layer computes $\boldsymbol{z}^{(2)} = W^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}$, producing raw scores (logits) for each class. Finally, softmax converts logits to probabilities: $p_k = e^{z_k} / \sum_j e^{z_j}$. The predicted class is $\arg\max_k p_k$. Training adjusts all weights $W^{(1)}, W^{(2)}, \boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}$ via backpropagation to minimize cross-entropy loss between predicted probabilities and true labels.

Q8.11: *"Why don't we just use a very wide single hidden layer instead of going deep? What's the benefit of depth?"*

Model Answer: While the Universal Approximation Theorem says a single hidden layer *can* approximate any function, the number of neurons required may be exponentially large. Depth allows the network to build hierarchical representations—each layer extracts increasingly abstract features by composing simpler ones. For example, in image recognition, early layers detect edges, middle layers combine edges into textures and parts, and final layers recognize objects. This compositionality means deep networks can represent complex functions with exponentially fewer parameters than shallow ones. Empirically, deep networks generalize better because they learn reusable features. The trade-off is that deep networks are harder to train (vanishing gradients, harder optimization landscape), which is why we need techniques like ReLU, batch normalization, residual connections, and careful initialization.

Key Takeaways

- Neural networks = linear transformations + nonlinear activations, composed layer by layer.
- Nonlinearity is essential: without it, any number of layers collapses to a single linear function.
- ReLU dominates because it avoids vanishing gradients and is computationally efficient.
- The Universal Approximation Theorem guarantees expressiveness; depth provides practical efficiency.
- Proper initialization (Xavier/He) keeps activations stable across layers.
- Each layer learns increasingly abstract representations of the input.

Chapter 9

Backpropagation

9.1 Introduction

Backpropagation is the algorithm that makes training neural networks feasible. It computes the gradient of the loss with respect to *every* parameter using the chain rule—efficiently, in a single backward pass through the computational graph.

The Key Insight: A naive approach would compute each partial derivative independently, requiring a separate forward pass per parameter. For a network with P parameters, that's P forward passes! Backpropagation does it in **one forward pass + one backward pass**, regardless of P. This is what makes training networks with millions of parameters practical.

9.2 Computational Graphs

A **computational graph** represents a function as a directed acyclic graph (DAG) where:

- **Nodes** are operations (addition, multiplication, activation functions).
- **Edges** carry data (tensors) between operations.
- The **forward pass** computes the function value from inputs to output.
- The **backward pass** computes gradients from output to inputs.

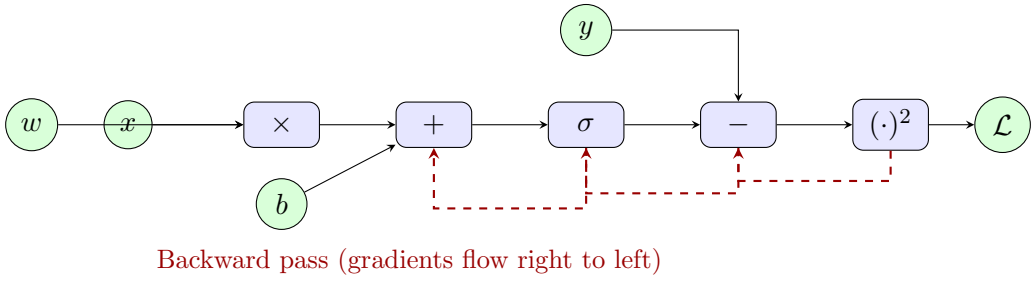


Figure 9.1: Forward pass flows left-to-right; backward pass (dashed red) flows right-to-left, carrying gradients.

9.3 The Backpropagation Algorithm

For each layer ℓ, define the "error signal" $\boldsymbol{\delta}^{(\ell)}$:
Step 1 (Forward pass): Compute $\boldsymbol{z}^{(\ell)} = W^{(\ell)}\boldsymbol{a}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}$ and $\boldsymbol{a}^{(\ell)} = g(\boldsymbol{z}^{(\ell)})$ for all layers. Store all intermediate values.
Step 2 (Output error): $\boldsymbol{\delta}^{(L)} = \nabla_{\boldsymbol{a}^{(L)}}\mathcal{L} \odot g'(\boldsymbol{z}^{(L)})$
Step 3 (Propagate backward): For $\ell = L-1, \ldots, 1$:

$$\boldsymbol{\delta}^{(\ell)} = \left((W^{(\ell+1)})^\top \boldsymbol{\delta}^{(\ell+1)}\right) \odot g'(\boldsymbol{z}^{(\ell)})$$

Step 4 (Parameter gradients):

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}(\boldsymbol{a}^{(\ell-1)})^\top, \qquad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}$$

Intuition: $\boldsymbol{\delta}^{(\ell)}$ is the "blame" or "responsibility" of layer ℓ for the final loss:

- The blame of the output layer comes directly from the loss function (Step 2).
- The blame of each hidden layer comes from the layers it feeds into, weighted by the connections (Step 3). A neuron with large outgoing weights bears more blame.
- The weight gradient is "blame $\times$ activation" (Step 4). Large activations with large blame lead to large weight updates.

This is just the chain rule applied systematically!

9.4 Worked Example: Full Backprop Pass

Example 9.1: Simple network: input $x = 2$, one hidden neuron ($w_1 = 0.5$, $b_1 = -0.1$, ReLU), one output neuron ($w_2 = -1$, $b_2 = 0.3$, identity), target $y = 1$. Loss: $\mathcal{L} = (\hat{y} - y)^2$.

Forward Pass:

$$\begin{aligned} z_1 &= w_1 x + b_1 = 0.5(2) + (-0.1) = 0.9 \\ a_1 &= \text{ReLU}(0.9) = 0.9 \\ z_2 &= w_2 a_1 + b_2 = (-1)(0.9) + 0.3 = -0.6 \\ \hat{y} &= z_2 = -0.6 \quad \text{(identity activation)} \\ \mathcal{L} &= (-0.6 - 1)^2 = 2.56 \end{aligned}$$

Backward Pass:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - y) = 2(-0.6 - 1) = -3.2$$
$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot a_1 = -3.2 \times 0.9 = -2.88$$
$$\frac{\partial \mathcal{L}}{\partial b_2} = -3.2$$
$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot w_2 = -3.2 \times (-1) = 3.2$$
$$\frac{\partial \mathcal{L}}{\partial z_1} = \frac{\partial \mathcal{L}}{\partial a_1} \cdot \text{ReLU}'(0.9) = 3.2 \times 1 = 3.2$$
$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial z_1} \cdot x = 3.2 \times 2 = 6.4$$
$$\frac{\partial \mathcal{L}}{\partial b_1} = 3.2$$

With $\eta = 0.01$: $w_1' = 0.5 - 0.01(6.4) = 0.436$, $w_2' = -1 - 0.01(-2.88) = -0.9712$.

9.5 The Vanishing and Exploding Gradient Problem

With sigmoid activations, $0 < \sigma'(z) \leq 0.25$. Through L layers, the gradient includes a product of L such terms:

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} \propto \prod_{\ell=1}^{L} \sigma'(z^{(\ell)}) \cdot W^{(\ell)}$$

If each factor < 1: gradient $\to 0$ exponentially (**vanishing**). If each factor > 1: gradient $\to \infty$ (**exploding**).

Solutions to vanishing gradients:

1. **ReLU:** Derivative is exactly 1 for $z > 0$. No multiplication by a number < 1.
2. **Residual (Skip) connections:** $\boldsymbol{a}^{(\ell)} = g(\boldsymbol{z}^{(\ell)}) + \boldsymbol{a}^{(\ell-1)}$. Gradients can flow directly through the skip connection, bypassing the vanishing factors. This is the key innovation of ResNets.
3. **Careful initialization:** Xavier/He initialization keeps gradient magnitudes stable.
4. **Batch normalization:** Keeps pre-activation values in a well-behaved range.

9.6 Automatic Differentiation

Modern Practice: No one implements backpropagation by hand anymore. Frameworks like PyTorch and JAX provide **automatic differentiation**: you define the forward computation, and the framework automatically builds the computational graph and computes all gradients.

Listing 9.1: Automatic differentiation in PyTorch

```
1 import torch
```

```

x = torch.tensor(2.0, requires_grad=True)
w1 = torch.tensor(0.5, requires_grad=True)
b1 = torch.tensor(-0.1, requires_grad=True)
w2 = torch.tensor(-1.0, requires_grad=True)
b2 = torch.tensor(0.3, requires_grad=True)

# Forward pass (PyTorch builds the graph automatically)
a1 = torch.relu(w1 * x + b1)
y_hat = w2 * a1 + b2
loss = (y_hat - 1.0)**2

# Backward pass (one call computes ALL gradients)
loss.backward()
print(f"dL/dw1 = {w1.grad:.2f}")  # 6.40
print(f"dL/dw2 = {w2.grad:.2f}")  # -2.88
```

Practice Questions & Answers

Conceptual Questions

Q9.1: Explain the difference between the forward pass and the backward pass in backpropagation. What is computed during each?

A9.1: The **forward pass** computes the network's output given an input: we propagate $\boldsymbol{x}$ through each layer, computing $\boldsymbol{z}^{(\ell)} = W^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}$ and $\boldsymbol{h}^{(\ell)} = \sigma(\boldsymbol{z}^{(\ell)})$, caching all intermediate values. At the end, we compute the loss L. The **backward pass** computes gradients of the loss with respect to every parameter using the chain rule in reverse: starting from $\partial L/\partial \hat{y}$, we propagate error signals backward through each layer, computing $\partial L/\partial W^{(\ell)}$ and $\partial L/\partial \boldsymbol{b}^{(\ell)}$. The key insight is that each node's local gradient is multiplied by the upstream gradient, so we only need one backward pass to get *all* parameter gradients—this is what makes backpropagation efficient ($O(n)$ rather than $O(n^2)$ if we computed each gradient independently).

Q9.2: What causes vanishing gradients in deep networks, and what are three techniques to mitigate them?

A9.2: Vanishing gradients occur when the chain rule produces a product of many small numbers. Specifically, if the local gradient at each layer is < 1 (e.g., sigmoid's maximum derivative is 0.25), then after L layers the gradient shrinks as $\sim 0.25^L$, making early layers learn extremely slowly. Three mitigations: (1) **ReLU activations**: gradient is exactly 1 for positive inputs, preventing shrinkage. (2) **Residual connections** (skip connections): the gradient has a direct path through the identity shortcut, so it doesn't have to pass through every nonlinearity. (3) **Careful initialization** (Xavier/He): keeps the variance of activations

and gradients ≈ 1 across layers. Other techniques include batch normalization (normalizes activations) and LSTM/GRU gating (for recurrent networks).

Q9.3: Why is automatic differentiation (autograd) different from both numerical differentiation and symbolic differentiation?

A9.3: Numerical differentiation uses finite differences: $\partial f/\partial x_i \approx (f(x + \epsilon e_i) - f(x))/\epsilon$. It's easy to implement but requires $O(n)$ forward passes for n parameters and suffers from numerical instability (choosing ϵ). **Symbolic differentiation** manipulates mathematical expressions algebraically (like Mathematica). It gives exact results but expressions can explode in size ("expression swell") for complex computational graphs. **Automatic differentiation** (what PyTorch/TensorFlow use) is the best of both worlds: it computes exact gradients by applying the chain rule to elementary operations, stored as a computational graph. Reverse-mode autodiff (backpropagation) computes all gradients in a single backward pass, with cost proportional to the forward pass. It avoids both numerical error and expression swell.

Multiple Choice

Q9.4: During backpropagation through a ReLU activation with pre-activation $z = -3$, what is the local gradient?

(a) 1
(b) -3
(c) 0
(d) 0.01

Answer: (c) 0. The derivative of $\text{ReLU}(z) = \max(0, z)$ is 1 if $z > 0$ and 0 if $z < 0$. Since $z = -3 < 0$, the local gradient is 0, meaning the upstream gradient is completely blocked—this neuron contributes nothing to parameter updates.

Q9.5: In a computational graph, the node $f = x \cdot y$ has inputs $x = 3, y = 4$, and an upstream gradient of $\frac{\partial L}{\partial f} = 2$. What are $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$?

(a) $\frac{\partial L}{\partial x} = 6$, $\frac{\partial L}{\partial y} = 8$
(b) $\frac{\partial L}{\partial x} = 8$, $\frac{\partial L}{\partial y} = 6$
(c) $\frac{\partial L}{\partial x} = 2$, $\frac{\partial L}{\partial y} = 2$
(d) $\frac{\partial L}{\partial x} = 4$, $\frac{\partial L}{\partial y} = 3$

Answer: (b) $\frac{\partial L}{\partial x} = 8$, $\frac{\partial L}{\partial y} = 6$. The local gradients are $\frac{\partial f}{\partial x} = y = 4$ and $\frac{\partial f}{\partial y} = x = 3$. By the chain rule: $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial x} = 2 \times 4 = 8$ and $\frac{\partial L}{\partial y} = 2 \times 3 = 6$. Notice the "swap": the gradient with respect to x depends on y's value and vice versa.

Q9.6: What happens to the gradient if it passes through 50 consecutive sigmoid layers during backpropagation?

(a) It stays approximately the same magnitude

(b) It explodes exponentially
(c) It vanishes toward zero
(d) It oscillates between positive and negative values

Answer: (c) It vanishes toward zero. The maximum derivative of sigmoid is $\sigma'(0) = 0.25$. After 50 layers, the gradient is multiplied by at most $0.25^{50} \approx 10^{-30}$, which is essentially zero. This is why sigmoid activations make training deep networks nearly impossible, and why ReLU (with gradient 1 for positive inputs) became the standard.

Computational Problems

Q9.7: Consider the function $f(x, y) = (x + y) \cdot (y + 1)$. Draw the computational graph and compute $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ at $x = 2, y = 3$ using backpropagation.

Solution: Let $a = x + y = 5$, $b = y + 1 = 4$, $f = a \cdot b = 20$.

Forward pass: $a = 5, b = 4, f = 20$.

Backward pass (starting from $\frac{\partial f}{\partial f} = 1$):

$$\frac{\partial f}{\partial a} = b = 4, \quad \frac{\partial f}{\partial b} = a = 5$$
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a} \cdot \frac{\partial a}{\partial x} = 4 \cdot 1 = 4$$
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a} \cdot \frac{\partial a}{\partial y} + \frac{\partial f}{\partial b} \cdot \frac{\partial b}{\partial y} = 4 \cdot 1 + 5 \cdot 1 = 9$$

Note that y has two paths to f (through both a and b), so we *sum* the gradients from each path. Verification: $f = xy + x + y^2 + y$, so $\frac{\partial f}{\partial x} = y + 1 = 4$ and $\frac{\partial f}{\partial y} = x + 2y + 1 = 9$. ✓

Q9.8: A one-hidden-layer network has: hidden layer $h = \text{ReLU}(w_1 x)$, output $\hat{y} = w_2 h$, loss $L = (\hat{y} - y)^2$. Given $x = 2, y = 1, w_1 = 0.5, w_2 = 1.5$, compute all gradients via backpropagation.

Solution:

$$\text{Forward:} \quad z_1 = w_1 x = 0.5 \times 2 = 1, \quad h = \text{ReLU}(1) = 1$$
$$\hat{y} = w_2 h = 1.5 \times 1 = 1.5, \quad L = (1.5 - 1)^2 = 0.25$$

$$\text{Backward:} \quad \frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y) = 2(0.5) = 1.0$$
$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} \cdot h = 1.0 \times 1 = 1.0$$
$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial \hat{y}} \cdot w_2 = 1.0 \times 1.5 = 1.5$$
$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial h} \cdot \mathbb{1}[z_1 > 0] = 1.5 \times 1 = 1.5 \quad (\text{since } z_1 = 1 > 0)$$
$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z_1} \cdot x = 1.5 \times 2 = 3.0$$

Q9.9: Write Python code to visualize gradient flow through sigmoid vs. ReLU across increasing network depths.

Listing 9.2: Gradient flow: Sigmoid vs. ReLU through multiple layers

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

depths = np.arange(1, 51)
# Sigmoid: max gradient per layer is 0.25
sigmoid_grad = 0.25 ** depths
# ReLU: ~50% neurons active, gradient = 1 for active
relu_grad = 0.5 ** depths  # worst case: half die each layer
relu_ideal = np.ones_like(depths, dtype=float)  # best case

fig, ax = plt.subplots(figsize=(8, 5))
ax.semilogy(depths, sigmoid_grad, 'r-', lw=2, label='Sigmoid (max)')
ax.semilogy(depths, relu_grad, 'b--', lw=2, label='ReLU (50% active)')
ax.semilogy(depths, relu_ideal, 'g:', lw=2, label='ReLU (ideal)')
ax.set_xlabel('Network Depth (layers)', fontsize=12)
ax.set_ylabel('Gradient Magnitude', fontsize=12)
ax.set_title('Gradient Vanishing: Sigmoid vs ReLU', fontsize=13)
ax.legend(fontsize=11); ax.grid(True, alpha=0.3)
ax.axhline(1e-10, color='k', ls='--', alpha=0.5)
ax.text(35, 3e-10, 'Effectively zero', fontsize=9)
plt.tight_layout()
plt.savefig('gradient_flow_comparison.pdf')
plt.show()
```

Interview-Style Questions

Q9.10: *"If I give you a neural network as a black box, how would you verify that your backpropagation implementation is correct?"*

Model Answer: The standard technique is **gradient checking**: for each parameter θ_i, compute the numerical gradient using centered finite differences: $\frac{\partial L}{\partial \theta_i} \approx \frac{L(\theta_i+\epsilon)-L(\theta_i-\epsilon)}{2\epsilon}$ with $\epsilon \approx 10^{-5}$. Then compare with the analytical gradient from backpropagation. The relative error should be $< 10^{-5}$ for double precision: $\frac{|g_{\text{analytic}}-g_{\text{numeric}}|}{|g_{\text{analytic}}|+|g_{\text{numeric}}|}$. Important caveats: (1) disable dropout and batch norm (they add stochasticity); (2) use a small network because numerical gradients require two forward passes *per parameter*; (3) check a random subset of parameters; (4) don't forget to check biases too. In practice, frameworks like PyTorch provide `torch.autograd.gradcheck()` for this purpose.

Q9.11: *"Explain the exploding gradient problem and how gradient clipping addresses it."*

Model Answer: Exploding gradients occur when the product of layer-wise gradients grows exponentially. If the weight matrices have spectral norm > 1 and activations don't saturate,

repeated multiplication during backpropagation can produce gradients of magnitude 10^{10} or more. This causes enormous weight updates that destabilize training—loss shoots to infinity or becomes NaN. **Gradient clipping** is the most common fix: before each update, compute the global gradient norm $\|\boldsymbol{g}\| = \sqrt{\sum_i g_i^2}$. If $\|\boldsymbol{g}\| > \tau$ (a threshold, typically 1–5), rescale: $\boldsymbol{g} \leftarrow \frac{\tau}{\|\boldsymbol{g}\|}\boldsymbol{g}$. This preserves gradient direction while bounding magnitude. It's especially critical in RNNs where gradients flow through many time steps. Other mitigations include weight regularization, proper initialization, and using LSTM/GRU architectures that naturally bound gradient flow through gates.

Key Takeaways

- Backpropagation = chain rule applied to computational graphs. One forward + one backward pass computes all gradients.
- At each node: upstream gradient $\times$ local gradient = downstream gradient.
- The error signal $\boldsymbol{\delta}^{(\ell)}$ propagates backward, assigning "blame" to each layer.
- Vanishing gradients (sigmoid/tanh) cripple deep networks. ReLU, skip connections, and proper initialization fix this.
- Modern frameworks (PyTorch, JAX) implement backprop automatically. Understanding the algorithm helps debug training issues.

Part IV

Advanced Architectures

Chapter 10

Convolutional Neural Networks

10.1 Introduction

A 224×224 RGB image has 150,528 features. A fully-connected layer mapping these to just 1,000 hidden neurons would need 150 million parameters—for a *single* layer. CNNs exploit the spatial structure of images to reduce parameters dramatically.

Two Key Insights:

1. **Locality:** A pixel's meaning depends on its neighbors, not distant pixels. An edge detector only needs to look at a small patch.
2. **Translation invariance:** An edge is an edge regardless of where it appears in the image. The same detector should work everywhere.

Convolution implements both: a small filter slides across the image, applying the same weights at every position. This is **weight sharing**—one set of weights, many positions.

10.2 The Convolution Operation

The 2D convolution (technically cross-correlation) of input I with kernel K:

$$(I * K)_{ij} = \sum_{p=0}^{k-1} \sum_{q=0}^{k-1} I_{i+p,\, j+q} \cdot K_{p,q}$$

Output size formula: For input size n, kernel size k, padding p, stride s:

$$\text{output size} = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1$$

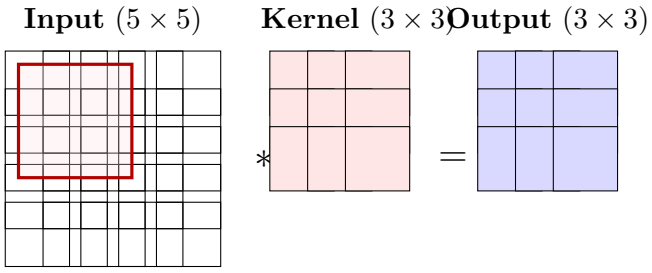


Figure 10.1: A 3×3 kernel slides over a 5×5 input, producing a 3×3 output. Each output value is a dot product of the kernel with the corresponding input patch.

Listing 10.1: 2D convolution and edge detection

```python
import numpy as np

def conv2d(image, kernel):
    H, W = image.shape
    kH, kW = kernel.shape
    output = np.zeros((H - kH + 1, W - kW + 1))
    for i in range(output.shape[0]):
        for j in range(output.shape[1]):
            output[i, j] = np.sum(image[i:i+kH, j:j+kW] * kernel)
    return output

# Vertical edge detector
vertical_edge = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1.]])

# Horizontal edge detector
horizontal_edge = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1.]])
```

Parameter savings: A 3×3 kernel applied to a 224×224 image has just 9 parameters (plus bias), but produces a 222×222 feature map. A fully-connected layer doing the same would need $224^2 \times 222^2 \approx 2.5$ billion parameters! This is the power of weight sharing.

10.3 Pooling

Max pooling takes the maximum of each non-overlapping region. A 2×2 max pool with stride 2 halves spatial dimensions while keeping the most prominent feature in each region. **Average pooling** takes the mean instead. **Global average pooling** reduces an entire feature map to a single number.

Pooling provides:

- **Translation invariance:** If a feature moves slightly, the max in the region stays the same.
- **Dimensionality reduction:** Halving each dimension reduces computation by 4x.
- **Larger receptive field:** After pooling, each neuron in the next layer "sees" a larger portion of the original image.

10.4 CNN Architecture

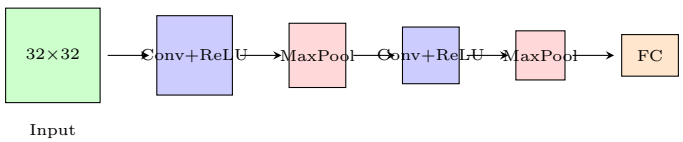


Figure 10.2: A typical CNN: Conv+ReLU extracts features, MaxPool reduces spatial dimensions. Spatial size shrinks while depth (channels) grows. The final FC layer maps features to predictions.

10.5 What CNNs Learn

Hierarchical Feature Learning:

- **Layer 1:** Edges, color gradients, simple textures.
- **Layer 2:** Corners, curves, simple shapes.
- **Layers 3–4:** Object parts (eyes, wheels, windows).
- **Deep layers:** Entire objects and scenes.

This hierarchy is not hand-designed—the network *discovers* it from data during training. This is one of the most remarkable properties of deep learning.

10.6 Landmark CNN Architectures

Key milestones:

- **LeNet-5** (1998): 2 conv + 3 FC layers. Digit recognition.
- **AlexNet** (2012): 5 conv + 3 FC. ReLU + dropout. Won ImageNet by a huge margin. Launched the deep learning revolution.
- **VGGNet** (2014): Very deep (16–19 layers), all 3×3 convolutions. Showed that depth matters.
- **ResNet** (2015): Skip connections enable 152+ layers. Won ImageNet. Skip connections solve vanishing gradients.

Practice Questions & Answers

Conceptual Questions

Q10.1: Why do CNNs use weight sharing instead of fully-connected layers for image data? Quantify the parameter savings for a single convolutional layer.

A10.1: Weight sharing exploits two key properties of images: **locality** (a pixel's meaning depends on its neighbors, not distant pixels) and **translation invariance** (an edge detector should work the same way regardless of position). A convolutional layer with K filters of size $f \times f$ on C input channels has $K \cdot (f^2C + 1)$ parameters—independent of input spatial dimensions. A fully-connected layer on the same input would have $K \cdot (H \times W \times C + 1)$ parameters. For a $224 \times 224 \times 3$ image with 64 filters of size 3×3: conv layer has $64 \times (9 \times 3 + 1) = 1{,}792$ parameters, while a FC layer would have $64 \times (224 \times 224 \times 3 + 1) \approx 9.6$ million. That's a 5,400× reduction.

Q10.2: Explain the difference between max pooling and average pooling. When might you prefer one over the other?

A10.2: **Max pooling** takes the maximum value in each pooling window, retaining the strongest activation (most prominent feature). It provides slight translation invariance because if a feature shifts by a few pixels, the max may still capture it. **Average pooling** takes the mean, producing a smoother, more distributed summary. Max pooling is preferred in most classification tasks because it captures whether a feature is present anywhere in the region (making it more robust to small translations). Average pooling is often used in the final layer (global average pooling) to collapse spatial dimensions before the classifier, replacing large fully-connected layers. Average pooling is also preferred when you want to preserve information about how active a region is overall, such as in some segmentation architectures.

Q10.3: How do residual (skip) connections in ResNets solve the degradation problem, and why do they help gradient flow?

A10.3: The **degradation problem** is that very deep plain networks perform *worse* than shallower ones, even on training data—which is surprising because a deeper network could just learn identity mappings for extra layers. In practice, it's hard for layers to learn identity via nonlinear transformations. ResNets add skip connections: $\boldsymbol{h}^{(\ell+1)} = \text{ReLU}(\boldsymbol{z}^{(\ell+1)} + \boldsymbol{h}^{(\ell)})$, so the layer only needs to learn the *residual* $F(\boldsymbol{x}) = H(\boldsymbol{x}) - \boldsymbol{x}$, which defaults to zero. For gradient flow: $\frac{\partial L}{\partial \boldsymbol{h}^{(\ell)}} = \frac{\partial L}{\partial \boldsymbol{h}^{(\ell+1)}} \cdot \left(\frac{\partial F}{\partial \boldsymbol{h}^{(\ell)}} + I \right)$. The identity term I provides a gradient "highway" that bypasses the nonlinear path, preventing vanishing gradients even in 100+ layer networks.

Multiple Choice

Q10.4: An input image is 32×32 with 3 channels. A convolutional layer has 16 filters of size 5×5, stride 1, and no padding. What is the output shape?

(a) $32 \times 32 \times 16$
(b) $28 \times 28 \times 16$
(c) $28 \times 28 \times 3$
(d) $30 \times 30 \times 16$

Answer: (b) $28 \times 28 \times 16$. Output size $= \lfloor (n - f)/s \rfloor + 1 = \lfloor (32 - 5)/1 \rfloor + 1 = 28$. The number of output channels equals the number of filters (16). The 3 input channels are handled within each filter (each filter is $5 \times 5 \times 3$).

Q10.5: How many learnable parameters does a single 3×3 convolutional filter have when applied to an input with 64 channels?

(a) 9
(b) 576
(c) 577
(d) 10

Answer: (c) 577. Each filter spans all input channels: $3 \times 3 \times 64 = 576$ weights, plus 1 bias $= 577$. This is why modern CNNs can have many channels but remain manageable in parameter count—the spatial kernel is small.

Q10.6: Which of the following best describes what happens in the early layers of a CNN trained on natural images?

(a) They detect high-level objects like faces and cars
(b) They learn color histograms of the entire image
(c) They detect low-level features like edges, corners, and textures
(d) They perform dimensionality reduction like PCA

Answer: (c) They detect low-level features like edges, corners, and textures. This hierarchical feature learning is a hallmark of CNNs: early layers learn Gabor-like edge filters at various orientations, middle layers combine these into textures and parts (e.g., eyes, wheels), and final layers compose them into object-level representations. This was confirmed by visualizing learned filters in networks like AlexNet.

Computational Problems

Q10.7: Compute the 2D convolution (valid, no padding, stride 1) of the input matrix with the given kernel:

$$\text{Input} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}, \quad \text{Kernel} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Solution: Output size: $(3-2)/1+1=2$, so the output is 2×2.

$$\begin{aligned} \text{Top-left:} \quad & 1\cdot 1+0\cdot 0+0\cdot 0+1\cdot 1=2 \\ \text{Top-right:} \quad & 0\cdot 1+1\cdot 0+1\cdot 0+0\cdot 1=0 \\ \text{Bottom-left:} \quad & 0\cdot 1+1\cdot 0+1\cdot 0+0\cdot 1=0 \\ \text{Bottom-right:} \quad & 1\cdot 1+0\cdot 0+0\cdot 0+1\cdot 1=2 \end{aligned}$$

$$\text{Output} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

This kernel is a diagonal edge/identity detector—it responds strongly to features along the main diagonal.

Q10.8: A CNN has the following architecture. Compute the output spatial dimensions and total parameters at each layer. Input: $64 \times 64 \times 3$.

1. Conv: 32 filters, 3×3, stride 1, padding 1
2. Max pool: 2×2, stride 2
3. Conv: 64 filters, 3×3, stride 1, padding 1
4. Max pool: 2×2, stride 2

Solution:

Layer 1 (Conv):	Output: $64 \times 64 \times 32$ (padding=1 preserves size)
	Params: $32 \times (3 \times 3 \times 3 + 1) = 32 \times 28 = 896$
Layer 2 (Pool):	Output: $32 \times 32 \times 32$ (halved spatially, 0 params)
Layer 3 (Conv):	Output: $32 \times 32 \times 64$ (padding=1 preserves size)
	Params: $64 \times (3 \times 3 \times 32 + 1) = 64 \times 289 = 18{,}496$
Layer 4 (Pool):	Output: $16 \times 16 \times 64$ (halved spatially, 0 params)
Total params:	$896 + 18{,}496 = 19{,}392$

Q10.9: Write Python code to visualize the effect of different convolutional kernels on an image.

Listing 10.2: Visualizing convolution with different kernels

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import convolve2d

# Create a simple test image (checkerboard + gradient)
img = np.zeros((64, 64))
img[::8, :] = 1; img[:, ::8] = 1  # grid
img += np.linspace(0, 0.5, 64).reshape(1, -1)  # gradient

kernels = {
    'Identity': np.array([[0,0,0],[0,1,0],[0,0,0]]),
    'Edge (horizontal)': np.array([[-1,-1,-1],[0,0,0],[1,1,1]]),
    'Edge (vertical)': np.array([[-1,0,1],[-1,0,1],[-1,0,1]]),
    'Sharpen': np.array([[0,-1,0],[-1,5,-1],[0,-1,0]]),
    'Gaussian blur': np.array([[1,2,1],[2,4,2],[1,2,1]])/16,
}

fig, axes = plt.subplots(1, len(kernels)+1, figsize=(16, 3))
axes[0].imshow(img, cmap='gray'); axes[0].set_title('Original')
axes[0].axis('off')
for ax, (name, k) in zip(axes[1:], kernels.items()):
    out = convolve2d(img, k, mode='same', boundary='wrap')
    ax.imshow(out, cmap='gray'); ax.set_title(name, fontsize=9)
    ax.axis('off')
plt.tight_layout()
plt.savefig('convolution_kernels.pdf')
plt.show()
```

Interview-Style Questions

Q10.10: *"Explain the output size formula for convolution and why padding matters."*

Model Answer: For input size n, filter size f, stride s, and padding p, the output size is $\lfloor (n+2p-f)/s \rfloor + 1$. Without padding ($p = 0$), the output shrinks by $f-1$ pixels—a 3×3 filter on a 32×32 input gives 30×30. After several layers, the spatial dimensions shrink dramatically, and edge pixels contribute to fewer outputs than center pixels (the "boundary effect"). "Same" padding ($p = \lfloor f/2 \rfloor$) preserves spatial dimensions, ensuring every pixel contributes equally. "Valid" padding ($p = 0$) is used when you want the network to progressively reduce resolution. In practice, most modern architectures use same padding within convolutional blocks and explicit pooling or strided convolutions to control downsampling.

Q10.11: *"Why did ResNets represent a breakthrough, and how do skip connections affect training?"*

Model Answer: Before ResNets, networks deeper than about 20 layers actually got *worse*—not from overfitting, but because optimizers couldn't effectively learn identity mappings through nonlinear layers. Skip connections changed the learning objective from $H(\boldsymbol{x})$ to $F(\boldsymbol{x}) = H(\boldsymbol{x}) - \boldsymbol{x}$, where F is the residual. If the optimal transformation is close to identity, the network just needs to push F toward zero, which is much easier. For training, skip connections create "gradient highways": during backpropagation, gradients flow directly through the skip path without being attenuated by activations or weight matrices. This allowed training of 152-layer (and even 1000+ layer) networks. The idea was so fundamental that virtually all modern architectures (Transformers, DenseNets, U-Nets) use some form of skip connections.

Key Takeaways

- CNNs exploit locality and translation invariance via weight sharing across spatial positions.
- Convolution = sliding dot product. Massive parameter savings vs. fully-connected layers.
- Pooling provides translation invariance and reduces spatial dimensions.
- CNNs automatically learn hierarchical features: edges → parts → objects.
- Output size formula: $\lfloor (n + 2p - k)/s \rfloor + 1$.
- ResNets showed that skip connections enable very deep networks by solving the vanishing gradient problem.

Chapter 11

Sequence Models and Attention

11.1 Introduction

Many data types are sequences where **order matters**: text, speech, time series, DNA, music. A sentence read backward is nonsense. A stock price at time t depends on prices at times $t-1, t-2, \ldots$. Sequence models are designed to capture these temporal dependencies.

11.2 Recurrent Neural Networks (RNNs)

An RNN maintains a **hidden state** $\boldsymbol{h}_t$ that captures information from all previous time steps:

$$\boldsymbol{h}_t = \tanh(W_h \boldsymbol{h}_{t-1} + W_x \boldsymbol{x}_t + \boldsymbol{b})$$

The same weights $W_h, W_x, \boldsymbol{b}$ are shared across all time steps (weight sharing in time, just as CNNs share weights in space).

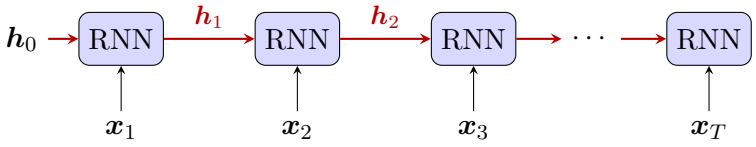


Figure 11.1: An RNN unrolled through time. The hidden state (red arrows) carries memory forward from step to step. Each box is the *same* RNN cell with shared weights.

Problem: As sequences get long, the memory of early inputs fades exponentially—the **vanishing gradient problem** applied to time. After 20–30 steps, the gradient from the output to early inputs is essentially zero. The network "forgets" the beginning of long sequences.

11.3 Long Short-Term Memory (LSTM)

An LSTM cell maintains two states:

- **Cell state** $\boldsymbol{c}_t$: Long-term memory (the "conveyor belt").
- **Hidden state** $\boldsymbol{h}_t$: Short-term output (what gets passed to the next layer).

Three gates control information flow:

$$\begin{aligned}
\boldsymbol{f}_t &= \sigma(W_f[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_f) && \text{(\textbf{Forget gate}: what to erase)} \\
\boldsymbol{i}_t &= \sigma(W_i[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_i) && \text{(\textbf{Input gate}: what to store)} \\
\tilde{\boldsymbol{c}}_t &= \tanh(W_c[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_c) && \text{(Candidate memory)} \\
\boldsymbol{c}_t &= \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t && \text{(Update cell state)} \\
\boldsymbol{o}_t &= \sigma(W_o[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_o) && \text{(\textbf{Output gate}: what to read)} \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t) && \text{(Output)}
\end{aligned}$$

Why LSTMs work: The cell state update $\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \ldots$ is **additive**, not multiplicative. Gradients flow through the cell state without being multiplied by weights or squashed by activation functions. This is the same principle behind ResNets' skip connections—additive shortcuts preserve gradients.

Analogy: Think of the cell state as a notebook:

- **Forget gate:** Erases some old notes ($\boldsymbol{f}_t \approx 0$ erases, $\boldsymbol{f}_t \approx 1$ keeps).
- **Input gate:** Writes new notes ($\boldsymbol{i}_t$ controls what gets written).
- **Output gate:** Reads specific notes for the current task ($\boldsymbol{o}_t$ controls what gets read).

11.4 The Attention Mechanism

Given a **query** $\boldsymbol{q}$, a set of **keys** $\boldsymbol{k}_1, \ldots, \boldsymbol{k}_T$, and **values** $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_T$:

$$\alpha_t = \frac{\exp(\boldsymbol{q}^\top \boldsymbol{k}_t / \sqrt{d_k})}{\sum_{j=1}^{T} \exp(\boldsymbol{q}^\top \boldsymbol{k}_j / \sqrt{d_k})}, \qquad \text{output} = \sum_{t=1}^{T} \alpha_t \boldsymbol{v}_t$$

The $\sqrt{d_k}$ scaling prevents the dot products from becoming too large (which would make softmax saturate).

Intuition: Attention is a **soft dictionary lookup**:

- The **query** asks: "What am I looking for?"
- The **keys** answer: "Here is what I contain."
- The **attention weights** α_t form a probability distribution over positions—how much to "attend" to each position.
- The **output** is a weighted average of values, emphasizing the most relevant information.

Unlike RNNs, attention has **direct connections** between any two positions (path length 1, regardless of sequence length). This solves the long-range dependency problem without relying on memory propagation.

11.5 The Transformer

The **Transformer** (Vaswani et al., 2017) replaces recurrence entirely with **self-attention**:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

where $Q = XW_Q$, $K = XW_K$, $V = XW_V$ are linear projections of the input.
Key components:

- **Multi-head attention:** Run h attention heads in parallel with different projections, then concatenate. Each head can attend to different types of relationships.
- **Positional encoding:** Since self-attention is order-invariant, position information is injected via sinusoidal encodings or learned embeddings.
- **Feed-forward layers:** A two-layer MLP applied independently to each position.
- **Residual connections + LayerNorm:** Around every sub-layer.

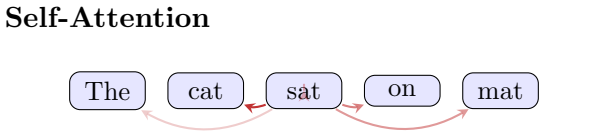


Figure 11.2: Self-attention: each word computes attention over all other words. Arrow thickness = attention weight. "sat" attends most to "cat" because it's the subject of the verb.

Why Transformers replaced RNNs:

1. **Parallelization:** All positions processed simultaneously (matrix multiplication). RNNs must process sequentially. On GPUs, this is a massive speedup.
2. **Long-range dependencies:** Direct path between any two positions (length 1). RNNs need $O(T)$ steps, during which information degrades.
3. **Scalability:** Efficient on modern hardware. Scales to billions of parameters (GPT-4, Claude, etc.).

Cost: $O(T^2)$ memory and computation (every position attends to every other). For very long sequences, this becomes a bottleneck—addressed by sparse attention, linear attention, and other approximations.

Listing 11.1: Self-attention in NumPy

```python
import numpy as np

def self_attention(X, Wq, Wk, Wv):
    """X: (seq_len, d_model), W*: (d_model, d_k)"""
    Q, K, V = X @ Wq, X @ Wk, X @ Wv
    d_k = Q.shape[-1]
    scores = Q @ K.T / np.sqrt(d_k)  # (seq_len, seq_len)
    weights = np.exp(scores) / np.exp(scores).sum(axis=-1, keepdims=True)
    return weights @ V  # (seq_len, d_k)

# Example: 5 tokens, 64-dim embeddings, 32-dim attention
```

```python
X = np.random.randn(5, 64)
Wq = np.random.randn(64, 32) * 0.1
Wk = np.random.randn(64, 32) * 0.1
Wv = np.random.randn(64, 32) * 0.1
output = self_attention(X, Wq, Wk, Wv)
print(f"Output shape: {output.shape}")  # (5, 32)
```

Practice Questions & Answers

Conceptual Questions

Q11.1: Why do vanilla RNNs struggle with long-range dependencies, and how do LSTMs address this?

A11.1: In a vanilla RNN, the hidden state is updated as $\boldsymbol{h}_t = \tanh(W_h \boldsymbol{h}_{t-1} + W_x \boldsymbol{x}_t + \boldsymbol{b})$. During backpropagation through time (BPTT), the gradient is multiplied by $W_h^\top$ at each step: $\frac{\partial L}{\partial \boldsymbol{h}_t} \propto (W_h^\top)^T \cdot \prod \text{diag}(\tanh')$. If the spectral radius of W_h is < 1, this product vanishes exponentially; if > 1, it explodes. LSTMs fix this with a **cell state** $\boldsymbol{c}_t$ that is updated *additively*: $\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t$. The forget gate $\boldsymbol{f}_t \in (0, 1)$ controls what to keep; the input gate $\boldsymbol{i}_t$ controls what to add. Because the cell state update is additive (like a residual connection in time), the gradient can flow through the cell state for hundreds of time steps without vanishing, as long as the forget gate stays close to 1.

Q11.2: Explain the attention mechanism as a "soft dictionary lookup." What are queries, keys, and values?

A11.2: Think of a Python dictionary: given a query key, you look up the exact match and return its value. Attention is a *soft* (differentiable) version: given a **query** $\boldsymbol{q}$, you compare it against all **keys** $\{\boldsymbol{k}_1, \ldots, \boldsymbol{k}_n\}$ to compute similarity scores. These scores become weights (via softmax), and the output is a weighted sum of the **values** $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$: $\text{Attention}(\boldsymbol{q}, K, V) = \text{softmax}\left(\frac{\boldsymbol{q}K^\top}{\sqrt{d_k}}\right)V$. Instead of returning one value (hard lookup), you return a *blend* of all values, weighted by relevance. The $\sqrt{d_k}$ scaling prevents dot products from becoming too large (which would make softmax outputs nearly one-hot). In self-attention, queries, keys, and values all come from the same sequence, projected through different learned matrices W_Q, W_K, W_V.

Q11.3: What is multi-head attention and why is it better than single-head attention?

A11.3: Multi-head attention runs h parallel attention operations, each with its own learned projections: $\text{head}_i = \text{Attention}(XW_Q^i, XW_K^i, XW_V^i)$, then concatenates and projects: $\text{MultiHead} = [\text{head}_1; \ldots; \text{head}_h]W_O$. Each head projects queries/keys/values into a lower-dimensional subspace ($d_k = d_{\text{model}}/h$). The benefit: different heads can attend to different types of relationships simultaneously. For example, in language, one head might capture

syntactic dependencies (subject-verb agreement), another might capture semantic similarity, and another might focus on positional proximity. Single-head attention with the same total dimensionality would be forced to average all these different relationship types into one set of attention weights, losing expressiveness. The total computation is similar since each head uses dimension $d_k = d/h$.

Multiple Choice

Q11.4: In an LSTM cell, what is the role of the forget gate $\boldsymbol{f}_t$?
(a) It decides what new information to store in the cell state
(b) It decides what portion of the previous cell state to retain
(c) It controls which parts of the cell state to output
(d) It normalizes the cell state to prevent explosion

Answer: (b) It decides what portion of the previous cell state to retain. The forget gate $\boldsymbol{f}_t = \sigma(W_f[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_f) \in (0,1)^d$ element-wise multiplies the previous cell state: $\boldsymbol{f}_t \odot \boldsymbol{c}_{t-1}$. Values near 1 mean "keep this memory"; values near 0 mean "forget it." Option (a) describes the input gate, option (c) describes the output gate.

Q11.5: In scaled dot-product attention, why do we divide by $\sqrt{d_k}$?
(a) To normalize the output to unit length
(b) To prevent the dot products from growing too large, which would push softmax into regions with tiny gradients
(c) To make the computation numerically stable for half-precision floating point
(d) To account for the number of attention heads

Answer: (b) Prevent dot products from growing too large. If $\boldsymbol{q}$ and $\boldsymbol{k}$ have entries with mean 0 and variance 1, their dot product has variance d_k. For large d_k (e.g., 512), the dot products become large in magnitude, pushing softmax outputs toward one-hot vectors where the gradient is nearly zero. Dividing by $\sqrt{d_k}$ rescales the variance back to 1, keeping softmax in a region with healthy gradients.

Q11.6: What is the key advantage of Transformers over RNNs for processing sequences?
(a) Transformers use fewer parameters
(b) Transformers can process all positions in parallel, not sequentially
(c) Transformers don't need positional information
(d) Transformers only work with text data

Answer: (b) Transformers can process all positions in parallel. RNNs process tokens sequentially ($h_1 \to h_2 \to \ldots$), creating a bottleneck that prevents parallelization and limits the effective context window. Self-attention computes all pairwise interactions simultaneously, enabling massive parallelism on GPUs. Note that Transformers *do* need positional encodings (option c is wrong) since attention is permutation-invariant by default.

Computational Problems

Q11.7: Compute the attention output for a single query. Given:

$$\boldsymbol{q} = (1, 0), \quad K = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad V = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \quad d_k = 2$$

Compute the attention weights (softmax of scaled dot products) and the output.

Solution:

$$\begin{aligned} \text{Scores: } \boldsymbol{q}K^\top &= (1,0)\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} = (1, 0, 1) \\ \text{Scaled: } \frac{\boldsymbol{q}K^\top}{\sqrt{d_k}} &= \frac{(1,0,1)}{\sqrt{2}} = (0.707, 0, 0.707) \\ \text{Softmax: } \alpha &= \text{softmax}(0.707, 0, 0.707) \\ &= \frac{(e^{0.707}, e^0, e^{0.707})}{e^{0.707} + e^0 + e^{0.707}} = \frac{(2.028, 1.0, 2.028)}{5.056} \\ &= (0.401, 0.198, 0.401) \\ \text{Output: } \boldsymbol{\alpha}^\top V &= 0.401 \times 1 + 0.198 \times 0 + 0.401 \times 1 = 0.802 \end{aligned}$$

The query attends most to keys 1 and 3 (which both have a 1 in the first position matching the query), and least to key 2.

Q11.8: An LSTM has hidden dimension 128 and input dimension 64. How many parameters are in the LSTM cell (excluding biases)?

Solution: An LSTM has four gates (forget, input, cell candidate, output), each computing $W[\boldsymbol{h}_{t-1}; \boldsymbol{x}_t]$. The concatenated input has dimension $128 + 64 = 192$. Each gate's weight matrix is 128×192.

$$\text{Parameters} = 4 \times (128 \times 192) = 4 \times 24{,}576 = 98{,}304$$

Including biases: $4 \times 128 = 512$ additional parameters, for a total of 98,816. This is why LSTMs are expensive—4 weight matrices per cell, each spanning both hidden and input dimensions.

Q11.9: Write Python code to visualize attention weights as a heatmap for a toy self-attention example.

Listing 11.2: Visualizing self-attention weights as a heatmap

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

def scaled_dot_product_attention(Q, K, V):
    d_k = Q.shape[-1]
    scores = Q @ K.T / np.sqrt(d_k)
    weights = np.exp(scores - scores.max(axis=-1, keepdims=True))
    weights /= weights.sum(axis=-1, keepdims=True)
```

```
    return weights @ V, weights

# Simulate a short sentence with random embeddings
np.random.seed(42)
tokens = ['The', 'cat', 'sat', 'on', 'the', 'mat']
n, d = len(tokens), 16
X = np.random.randn(n, d)
Wq = np.random.randn(d, d) * 0.1
Wk = np.random.randn(d, d) * 0.1
Wv = np.random.randn(d, d) * 0.1

Q, K, V = X @ Wq, X @ Wk, X @ Wv
output, attn_weights = scaled_dot_product_attention(Q, K, V)

fig, ax = plt.subplots(figsize=(6, 5))
im = ax.imshow(attn_weights, cmap='Blues', vmin=0, vmax=0.5)
ax.set_xticks(range(n)); ax.set_xticklabels(tokens, fontsize=11)
ax.set_yticks(range(n)); ax.set_yticklabels(tokens, fontsize=11)
ax.set_xlabel('Key (attended to)', fontsize=12)
ax.set_ylabel('Query (attending)', fontsize=12)
ax.set_title('Self-Attention Weights', fontsize=13)
plt.colorbar(im, ax=ax, shrink=0.8)
for i in range(n):
    for j in range(n):
        ax.text(j, i, f'{attn_weights[i,j]:.2f}',
                ha='center', va='center', fontsize=8)
plt.tight_layout()
plt.savefig('self_attention_heatmap.pdf')
plt.show()
```

Interview-Style Questions

Q11.10: *"Walk me through the Transformer architecture. What makes it different from RNNs, and why has it become dominant?"*

Model Answer: A Transformer block has two sub-layers: (1) **multi-head self-attention** and (2) a **position-wise feed-forward network**, each wrapped with residual connections and layer normalization. The input tokens are embedded and summed with positional encodings (since attention is order-agnostic). Self-attention lets every token directly attend to every other token in $O(1)$ sequential steps (vs. $O(n)$ for RNNs), enabling massive parallelism. The key differences from RNNs: (1) no sequential bottleneck—all positions are processed simultaneously; (2) direct connections between any two positions—no information has to pass through intermediate states; (3) the attention pattern is input-dependent (dynamic connectivity vs. fixed recurrence). Transformers dominate because they scale better with compute and data: parallelism enables training on huge datasets, and self-attention captures long-range dependencies that RNNs struggle with.

Q11.11: *"Compare the forget gate in an LSTM to a residual connection in a ResNet. What's the conceptual similarity?"*

Model Answer: Both mechanisms solve the same fundamental problem: enabling information and gradients to flow across many computational steps without degradation. In an LSTM, the cell state update is $\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t$. When $\boldsymbol{f}_t \approx 1$ and $\boldsymbol{i}_t \approx 0$, the cell state passes through unchanged—a "copy" operation. In a ResNet, $\boldsymbol{h}^{(\ell+1)} = F(\boldsymbol{h}^{(\ell)}) + \boldsymbol{h}^{(\ell)}$. When $F \approx 0$, the input passes through unchanged. Both create gradient "highways": the LSTM's additive update gives $\frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{c}_{t-1}} = \boldsymbol{f}_t$ (close to 1), and the ResNet's skip connection gives $\frac{\partial \boldsymbol{h}^{(\ell+1)}}{\partial \boldsymbol{h}^{(\ell)}} = \frac{\partial F}{\partial \boldsymbol{h}} + I$. The identity term ensures gradients never fully vanish. The key difference: LSTM gates are *dynamic* (input-dependent), while ResNet skip connections are *static* (always identity). This makes LSTMs more flexible but harder to analyze theoretically.

Key Takeaways

- RNNs process sequences with shared weights across time, but suffer from vanishing gradients on long sequences.
- LSTMs use gated memory cells with *additive* updates, allowing gradients to flow across many time steps.
- Attention computes a weighted average over all positions, with weights from query-key similarity. It's a soft dictionary lookup.
- Transformers replace recurrence entirely with self-attention, enabling parallelism and direct long-range connections.
- Multi-head attention lets different heads learn different types of relationships.
- The Transformer is the architecture behind GPT, BERT, Claude, and all modern large language models.

Chapter 12

Dimensionality Reduction

12.1 Introduction

Real-world datasets often have hundreds or thousands of features, but the data may "live" on a much lower-dimensional structure—a manifold embedded in high-dimensional space. Dimensionality reduction finds this structure, enabling visualization, denoising, and more efficient learning.

The Curse of Dimensionality: In 1D, 10 evenly spaced points cover $[0, 1]$ well. In 10D, you need 10^{10} points for the same coverage. As dimensions increase:

- All points appear approximately equidistant (distance-based methods break down).
- The volume of the space grows exponentially, making data sparse.
- Most of the "volume" concentrates near the surface of the hypersphere.

Dimensionality reduction fights this curse by projecting data onto the most informative subspace.

12.2 Principal Component Analysis (PCA)

PCA finds the directions (principal components) that capture the most variance:

1. **Center the data:** $\tilde{X} = X - \bar{\boldsymbol{x}}$ (subtract the mean of each feature).
2. **Compute covariance matrix:** $\Sigma = \frac{1}{m-1}\tilde{X}^\top \tilde{X}$.
3. **Eigendecompose:** $\Sigma = V \Lambda V^\top$. The eigenvectors V are the principal components; eigenvalues Λ are the variances.
4. **Project:** Keep top-k eigenvectors: $Z = \tilde{X} V_k$.

Variance retained: $\frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{n} \lambda_i}$. Typically choose k to retain 95% of variance.

Why eigenvectors? We want to find the direction $\boldsymbol{v}$ that maximizes the variance of the projected data: $\max_{\|\boldsymbol{v}\|=1} \boldsymbol{v}^\top \Sigma \boldsymbol{v}$. By the method of Lagrange multipliers, the solution satisfies $\Sigma \boldsymbol{v} = \lambda \boldsymbol{v}$—the eigenvector equation! The maximum variance is the eigenvalue λ.

Listing 12.1: PCA from scratch in NumPy

```
import numpy as np

def pca(X, k):
    """Reduce X (m x n) to k dimensions."""
    # Step 1: Center
    X_centered = X - X.mean(axis=0)

    # Step 2: Covariance matrix
```

```python
    cov = (X_centered.T @ X_centered) / (len(X) - 1)

    # Step 3: Eigendecompose
    eigenvalues, eigenvectors = np.linalg.eigh(cov)
    idx = np.argsort(eigenvalues)[::-1]  # Sort descending

    # Step 4: Project onto top-k components
    V_k = eigenvectors[:, idx[:k]]
    Z = X_centered @ V_k

    variance_retained = eigenvalues[idx[:k]].sum() / eigenvalues.sum()
    return Z, variance_retained

# Example: Reduce 50D data to 2D
X = np.random.randn(1000, 50)
Z, var_ret = pca(X, k=2)
print(f"Retained {var_ret:.1%} of variance in 2D")
```

Example 12.1: A dataset has eigenvalues $(10, 5, 2, 0.5, 0.1)$. Total variance $= 17.6$. For $k = 2$: $\frac{10+5}{17.6} = 85.2\%$. For $k = 3$: $\frac{10+5+2}{17.6} = 96.6\% \geq 95\%$. So $k = 3$ principal components suffice.

PCA limitations:

- Only captures **linear** relationships. If data lies on a curved manifold (e.g., Swiss roll), PCA cannot "unroll" it.
- Maximizes variance, which may not correspond to the most useful features for a downstream task. Variance doesn't always equal information.
- Sensitive to scaling: always standardize features before applying PCA.

12.3 t-SNE

t-SNE (t-distributed Stochastic Neighbor Embedding) is a nonlinear technique designed specifically for **visualization** (reducing to 2D or 3D):

1. In high-d: compute pairwise similarities using Gaussian kernels.
2. In low-d: compute similarities using heavy-tailed Student-t distribution.
3. Minimize KL divergence between the two similarity distributions.

The Student-t distribution's heavy tails prevent distant points from being crushed together, creating well-separated clusters.

t-SNE caveats (important!):

- **Distances between clusters are meaningless.** Only local structure (within clusters) is reliable.
- **Perplexity matters.** The perplexity hyperparameter controls the effective number of neighbors. Try multiple values (5–50).
- **Non-deterministic.** Different random seeds give different plots. Run multiple times.

- **Can't project new points.** Must re-run on the entire dataset.
- **Use only for visualization**, never as a preprocessing step for another algorithm.

	PCA	**t-SNE**
Type	Linear	Nonlinear
Speed	Fast ($O(n^2 d)$)	Slow ($O(n^2)$ per iteration)
New data	Easy (matrix multiply)	Must re-run
Preserves	Global variance	Local neighborhoods
Use case	Preprocessing, denoising	Visualization
Deterministic	Yes	No

12.4 Autoencoders

An **autoencoder** learns compressed representations by training a neural network to reconstruct its input through a bottleneck:

$$\begin{aligned} \boldsymbol{z} &= f_{\text{enc}}(\boldsymbol{x}) \quad \text{(encoder: compress)} \\ \hat{\boldsymbol{x}} &= f_{\text{dec}}(\boldsymbol{z}) \quad \text{(decoder: reconstruct)} \\ \mathcal{L} &= \|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2 \quad \text{(reconstruction loss)} \end{aligned}$$

The bottleneck $\boldsymbol{z}$ (latent representation) has fewer dimensions than $\boldsymbol{x}$, forcing the network to learn the most important features.

Listing 12.2: Autoencoder in PyTorch

```
import torch.nn as nn

class Autoencoder(nn.Module):
    def __init__(self, input_dim=784, latent_dim=32):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, 256), nn.ReLU(),
            nn.Linear(256, 128), nn.ReLU(),
            nn.Linear(128, latent_dim))
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 128), nn.ReLU(),
            nn.Linear(128, 256), nn.ReLU(),
            nn.Linear(256, input_dim), nn.Sigmoid())

    def forward(self, x):
        z = self.encoder(x)     # Compress: 784 -> 32
        return self.decoder(z)  # Reconstruct: 32 -> 784
```

Key insight: A linear autoencoder with MSE loss learns exactly the same subspace as PCA! The encoder weights span the top-k principal components. But nonlinear autoencoders can capture **curved manifolds** that PCA misses.
Variational Autoencoders (VAEs) add a probabilistic twist: the latent space is regularized

to follow a Gaussian distribution, enabling generation of new data by sampling from the latent space.

Practice Questions & Answers

Conceptual Questions

Q12.1: What is the "curse of dimensionality" and how does it affect machine learning algorithms? Give a concrete example.

A12.1: The curse of dimensionality refers to phenomena that arise when data lives in high-dimensional spaces. Key effects: (1) **Data sparsity**: the volume of space grows exponentially with dimension, so fixed-size datasets become increasingly sparse. To maintain the same density, you need exponentially more data. (2) **Distance concentration**: in high dimensions, the distance between the nearest and farthest neighbors converges, making distance-based methods (k-NN, clustering) unreliable. Concretely: in $d = 1000$ dimensions, for random unit vectors, $\|\boldsymbol{x}_i - \boldsymbol{x}_j\| \approx \sqrt{2d} \pm O(1)$ for all pairs, so nearest-neighbor search becomes meaningless. (3) **Overfitting**: more dimensions means more parameters needed, increasing the risk of fitting noise. Example: with 100 binary features, the feature space has $2^{100} \approx 10^{30}$ cells—no dataset can cover this, so most of the space is unseen during training.

Q12.2: Explain PCA step by step. What is the objective it optimizes, and what role does the covariance matrix play?

A12.2: PCA finds the directions of maximum variance in the data. Steps: (1) **Center** the data: $\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i - \bar{\boldsymbol{x}}$. (2) Compute the **covariance matrix**: $\Sigma = \frac{1}{m} X^\top X$. (3) **Eigendecompose**: $\Sigma = Q \Lambda Q^\top$, where columns of Q are eigenvectors and Λ has eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots$. (4) **Project**: take the top k eigenvectors as the new basis: $\boldsymbol{z}_i = Q_k^\top \boldsymbol{x}_i$. The objective is to maximize variance of the projected data (equivalently, minimize reconstruction error). The covariance matrix captures how features co-vary: its eigenvectors point in the directions of maximum spread, and eigenvalues quantify how much variance each direction explains. The fraction of variance retained is $\sum_{i=1}^{k} \lambda_i / \sum_{i=1}^{d} \lambda_i$. We typically choose k to retain $\geq 95\%$.

Q12.3: Compare t-SNE and PCA. When would you use each, and what are t-SNE's limitations?

A12.3: **PCA** is linear, deterministic, fast ($O(dp^2)$ or $O(dmk)$ with randomized SVD), and preserves global structure (large distances). Use it for: preprocessing before another algorithm, denoising, feature decorrelation, and when interpretability matters. **t-SNE** is nonlinear, stochastic, slow ($O(n^2)$ or $O(n \log n)$ with Barnes-Hut), and preserves local neighborhoods. Use it for: 2D/3D visualization of clusters. Limitations of t-SNE: (1) distances between clusters are meaningless—only within-cluster structure is reliable; (2) the perplex-

ity hyperparameter greatly affects results; (3) it's non-parametric (can't project new points without re-running); (4) different runs give different results; (5) it doesn't preserve global structure (clusters may appear equally spaced even if some are closer in original space); (6) it's too slow and non-invertible for preprocessing—never use t-SNE as a feature extraction step before classification.

Multiple Choice

Q12.4: The covariance matrix of a dataset has eigenvalues $\{10, 5, 3, 1, 0.5, 0.3, 0.1, 0.1\}$. How many principal components are needed to retain at least 90% of the variance?
(a) 2
(b) 3
(c) 4
(d) 5

Answer: (b) 3. Total variance = 10+5+3+1+0.5+0.3+0.1+0.1 = 20. Top 2: (10+5)/20 = 75% (not enough). Top 3: $(10 + 5 + 3)/20 = 90\%$. So 3 components suffice to retain exactly 90%.

Q12.5: Which of the following is true about a linear autoencoder trained with MSE loss?
(a) It learns features completely different from PCA
(b) Its encoder weights span the same subspace as the top PCA components
(c) It always outperforms PCA because it has more parameters
(d) It can capture nonlinear manifolds in the data

Answer: (b) Its encoder weights span the same subspace as the top PCA components. A linear autoencoder minimizes $\|X - XW_eW_d\|^2$, and the optimal solution has W_e spanning the top-k eigenvectors of $X^\top X$ (though not necessarily aligned with them individually—the subspace is the same). To capture nonlinear structure, you need nonlinear activations, making it a proper nonlinear autoencoder.

Q12.6: Why is it essential to standardize features (zero mean, unit variance) before applying PCA?
(a) PCA requires Gaussian-distributed features
(b) Features with larger scales dominate the principal components
(c) Standardization makes the covariance matrix diagonal
(d) It is only needed for numerical stability

Answer: (b) Features with larger scales dominate the principal components. PCA maximizes variance, so if feature A is in meters (range 0–10) and feature B is in millimeters (range 0–10000), PCA will declare feature B the most important direction simply because it has larger numbers, not because it carries more information. Standardizing ensures all features contribute equally. Note: (a) is false—PCA doesn't assume Gaussianity; (c) is false—standardization produces a *correlation* matrix, not a diagonal matrix.

Computational Problems

Q12.7: Given the 2D dataset $X = \{(2,1),(3,3),(5,4),(6,6),(8,7)\}$, compute the first principal component direction and the fraction of variance it explains.

Solution:

$$\bar{\boldsymbol{x}} = (4.8, 4.2)$$

$$\text{Centered data:} \quad X_c = \{(-2.8,-3.2),(-1.8,-1.2),(0.2,-0.2),(1.2,1.8),(3.2,2.8)\}$$

$$\Sigma = \frac{1}{5}X_c^\top X_c = \frac{1}{5}\begin{pmatrix} 22.96 & 21.68 \\ 21.68 & 21.36 \end{pmatrix} = \begin{pmatrix} 4.592 & 4.336 \\ 4.336 & 4.272 \end{pmatrix}$$

Eigenvalues: $\lambda_{1,2} = \frac{8.864 \pm \sqrt{8.864^2 - 4(0.783)}}{2} = \frac{8.864 \pm 8.687}{2}$.

So $\lambda_1 \approx 8.776$, $\lambda_2 \approx 0.088$.

Variance explained by PC1: $\frac{8.776}{8.776+0.088} \approx 99.0\%$.

The first eigenvector (PC1 direction): solving $(\Sigma - \lambda_1 I)\boldsymbol{v} = 0$ gives $\boldsymbol{v}_1 \approx (0.716, 0.698)^\top$—nearly the 45-degree line, confirming the data lies approximately along the diagonal.

Q12.8: A dataset has 1000 samples with 50 features. After PCA, the eigenvalue spectrum is $\lambda_k \propto 1/k^2$. Approximately how many components are needed to retain 95% of the variance?

Solution: The eigenvalues are $\lambda_k = C/k^2$ for some constant C. Total variance:

$$\sum_{k=1}^{50} \frac{1}{k^2} \approx \frac{\pi^2}{6} - \sum_{k=51}^{\infty} \frac{1}{k^2} \approx 1.6251$$

We need $\sum_{k=1}^{K} \frac{1}{k^2} \geq 0.95 \times 1.6251 = 1.5438$.

Computing partial sums: $\sum_{k=1}^{5} 1/k^2 = 1 + 0.25 + 0.111 + 0.0625 + 0.04 = 1.4636$ (90.1%).

$\sum_{k=1}^{7} = 1.4636 + 0.0278 + 0.0204 = 1.5118$ (93.0%).

$\sum_{k=1}^{9} \approx 1.5398$ (94.7%). $\sum_{k=1}^{10} \approx 1.5498$ (95.4%).

So approximately **10 components** out of 50 are needed—the $1/k^2$ decay means most variance is concentrated in the first few components.

Q12.9: Write Python code to demonstrate PCA on a 2D dataset and visualize the principal components.

Listing 12.3: PCA visualization: projecting 2D data onto principal components

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

np.random.seed(42)
# Generate correlated 2D data
mean = [3, 5]
```

```
cov = [[2, 1.5], [1.5, 3]]
X = np.random.multivariate_normal(mean, cov, 200)

# PCA by hand
X_centered = X - X.mean(axis=0)
C = np.cov(X_centered, rowvar=False)
eigenvalues, eigenvectors = np.linalg.eigh(C)
idx = np.argsort(eigenvalues)[::-1]
eigenvalues, eigenvectors = eigenvalues[idx], eigenvectors[:, idx]
explained = eigenvalues / eigenvalues.sum() * 100

fig, axes = plt.subplots(1, 2, figsize=(12, 5))
# Left: original data with PC directions
ax = axes[0]
ax.scatter(X[:, 0], X[:, 1], alpha=0.4, s=20)
origin = X.mean(axis=0)
for i, (ev, evec) in enumerate(zip(eigenvalues, eigenvectors.T)):
    ax.annotate('', xy=origin + evec*np.sqrt(ev)*2,
                xytext=origin,
                arrowprops=dict(arrowstyle='->', lw=2.5,
                color=['red','blue'][i]))
    ax.text(*(origin + evec*np.sqrt(ev)*2.3),
            f'PC{i+1} ({explained[i]:.1f}%)', fontsize=11,
            color=['red','blue'][i], fontweight='bold')
ax.set_xlabel('$x_1$'); ax.set_ylabel('$x_2$')
ax.set_title('Data with Principal Components')
ax.set_aspect('equal'); ax.grid(True, alpha=0.3)

# Right: projected data
Z = X_centered @ eigenvectors
ax = axes[1]
ax.scatter(Z[:, 0], Z[:, 1], alpha=0.4, s=20, c='green')
ax.set_xlabel('PC1'); ax.set_ylabel('PC2')
ax.set_title('Data in PC Space (decorrelated)')
ax.set_aspect('equal'); ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('pca_visualization.pdf')
plt.show()
```

Interview-Style Questions

Q12.10: *"When would you choose an autoencoder over PCA for dimensionality reduction?"*

Model Answer: PCA is optimal when the data lies near a *linear* subspace—it finds the best k-dimensional linear projection in the MSE sense. But real-world data often lies on curved, nonlinear manifolds. For example, images of a face rotating form a curved path in pixel space that no linear subspace captures well. Autoencoders with nonlinear activations can learn these curved manifolds, often achieving much lower reconstruction error for the same bottleneck dimension. I would choose autoencoders when: (1) the data has known nonlinear structure (images, audio, text embeddings); (2) I need a learned encoder for real-time inference on new

data; (3) I want to combine dimensionality reduction with other objectives (e.g., denoising autoencoders, VAEs for generation). I would stick with PCA when: (1) the dataset is small (autoencoders need lots of data); (2) interpretability matters (PCA components have clear meaning); (3) computation is limited; (4) as a preprocessing step before another algorithm.

Q12.11: *"You've run PCA and the first two components explain only 30% of the variance. What does this tell you, and what would you do next?"*

Model Answer: Low variance in the first two PCs means the data's variability is spread across many dimensions—there's no dominant low-dimensional linear structure. This could indicate: (1) the data is genuinely high-dimensional (many features carry independent information); (2) the data lies on a nonlinear manifold (PCA's linear assumption fails); or (3) features weren't standardized, and variance is distributed arbitrarily. What I'd do: First, verify standardization—if features have different scales, PCA is misleading. Second, plot the eigenvalue spectrum (scree plot)—maybe 10–20 components retain 95%, which is still a useful reduction from hundreds of features. Third, if the scree plot shows no clear elbow, try nonlinear methods: t-SNE or UMAP for visualization, or autoencoders for compression. Fourth, consider domain-specific feature engineering—maybe the raw features aren't the right representation. Finally, 30% in 2D doesn't mean PCA failed; it just means 2D isn't enough. Using 20 components retaining 95% variance is perfectly fine for downstream models.

Key Takeaways

- The curse of dimensionality makes high-dimensional data hard to work with; dimensionality reduction is often essential.
- PCA is linear, fast, and interpretable—project onto the top eigenvectors of the covariance matrix. Choose k to retain $\geq 95\%$ of variance.
- t-SNE excels at visualization by preserving local neighborhoods, but distances between clusters are meaningless.
- Autoencoders learn nonlinear compressed representations; a linear autoencoder is equivalent to PCA.
- Choose PCA for preprocessing and denoising, t-SNE for visualization, autoencoders for learning complex manifold structure.
- Always standardize features before dimensionality reduction.