Mathematical Foundations of Deep Learning and Embodied Intelligence

From Neural Networks to Diffusion Policies and VLA Models

February 12, 2026

Contents

Chapter 1

Mathematical Preliminaries

1.1 Vectors and Matrices

1.1.1 Vector Spaces and Norms

A vector $\mathbf{v} \in \mathbb{R}^n$ is an ordered list of n real numbers:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \tag{1.1}$$

The Euclidean norm (or L^2 norm) of a vector is defined as:

$$\|\mathbf{v}\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2} = \sqrt{\mathbf{v}^T\mathbf{v}} \tag{1.2}$$

More generally, the L^p norm is:

$$\|\mathbf{v}\|_p = \left(\sum_{i=1}^{n} |v_i|^p\right)^{1/p} \tag{1.3}$$

For $p = 1$ (Manhattan distance):

$$\|\mathbf{v}\|_1 = \sum_{i=1}^{n} |v_i| \tag{1.4}$$

For $p = \infty$ (maximum absolute value):

$$\|\mathbf{v}\|_\infty = \max_i |v_i| \tag{1.5}$$

1.1.2 Dot Product and Geometric Interpretation

The dot product (inner product) of two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$ is:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = \mathbf{a}^T\mathbf{b} \tag{1.6}$$

Geometric interpretation: The dot product relates to the angle θ between vectors via:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\|\mathbf{b}\| \cos\theta \tag{1.7}$$

This implies:

$$\cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|} \tag{1.8}$$

Key observations:

- When $\theta = 0$ (parallel): $\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\|\mathbf{b}\|$ (maximum)
- When $\theta = 90$ (orthogonal): $\mathbf{a} \cdot \mathbf{b} = 0$
- When $\theta = 180$ (anti-parallel): $\mathbf{a} \cdot \mathbf{b} = -\|\mathbf{a}\|\|\mathbf{b}\|$ (minimum)

In neural networks, the dot product $\mathbf{w} \cdot \mathbf{x}$ measures the alignment between weights and inputs. Large alignment (small angle) produces large activations.

1.1.3 Matrix Operations

A matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ has m rows and n columns:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \tag{1.9}$$

Matrix-vector multiplication: If $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{x} \in \mathbb{R}^n$, then $\mathbf{y} = \mathbf{A}\mathbf{x} \in \mathbb{R}^m$ where:

$$y_i = \sum_{j=1}^{n} a_{ij} x_j = \mathbf{a}_i^T \mathbf{x} \tag{1.10}$$

where $\mathbf{a}_i$ is the i-th row of $\mathbf{A}$. Notice that each output y_i is the dot product of row i with $\mathbf{x}$.

Matrix-matrix multiplication: If $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{n \times p}$, then $\mathbf{C} = \mathbf{A}\mathbf{B} \in \mathbb{R}^{m \times p}$ where:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = \mathbf{a}_i^T \mathbf{b}_j \tag{1.11}$$

where $\mathbf{a}_i$ is row i of $\mathbf{A}$ and $\mathbf{b}_j$ is column j of $\mathbf{B}$. Thus, the element at position (i, j) is the dot product of row i of $\mathbf{A}$ with column j of $\mathbf{B}$.

1.1.4 Important Matrix Properties

Transpose: Swapping rows and columns. If $\mathbf{A} \in \mathbb{R}^{m \times n}$, then $\mathbf{A}^T \in \mathbb{R}^{n \times m}$ with:

$$(\mathbf{A}^T)_{ij} = a_{ji} \tag{1.12}$$

Properties of transpose:

$$(\mathbf{A}\mathbf{B})^T = \mathbf{B}^T \mathbf{A}^T \quad \text{(reverse order!)} \tag{1.13}$$

Frobenius norm: For matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2} = \sqrt{\text{trace}(\mathbf{A}^T\mathbf{A})} \tag{1.14}$$

1.2 Calculus: Foundations of Optimization

1.2.1 Derivatives and the Chain Rule

For a univariate function $f : \mathbb{R} \to \mathbb{R}$, the derivative at point x is:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{1.15}$$

Geometrically, $f'(x)$ is the slope of the tangent line to f at x.

Chain Rule: If $y = f(u)$ and $u = g(x)$, then:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx} \tag{1.16}$$

More generally, for compositions $y = f(g(h(x)))$:

$$\frac{dy}{dx} = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{dx} \tag{1.17}$$

Example: Let $y = \sin(x^2)$. Set $u = x^2$, so $y = \sin(u)$.

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx} = \cos(u) \cdot 2x = 2x\cos(x^2) \tag{1.18}$$

1.2.2 Partial Derivatives and Gradients

For a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$, the partial derivative with respect to variable x_i is:

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h} \tag{1.19}$$

The gradient is the vector of all partial derivatives:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \tag{1.20}$$

Directional derivative: The rate of change of f in direction $\mathbf{d}$ (unit vector) is:

$$\nabla_{\mathbf{d}} f = \mathbf{d}^T \nabla f = \nabla f \cdot \mathbf{d} \tag{1.21}$$

The gradient ∇f points in the direction of steepest ascent. Negative gradient $-\nabla f$ points in the direction of steepest descent.

Example: Let $f(x, y) = x^2 + xy + 3y^2$.

$$\frac{\partial f}{\partial x} = 2x + y, \quad \frac{\partial f}{\partial y} = x + 6y \tag{1.22}$$

$$\nabla f = \begin{bmatrix} 2x + y \\ x + 6y \end{bmatrix} \tag{1.23}$$

At point $(x, y) = (1, 2)$:

$$\nabla f|_{(1,2)} = \begin{bmatrix} 2(1) + 2 \\ 1 + 6(2) \end{bmatrix} = \begin{bmatrix} 4 \\ 13 \end{bmatrix} \tag{1.24}$$

1.2.3 Matrix Calculus

For a scalar function $f(\mathbf{A})$ that depends on a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the gradient (or derivative) is a matrix:

$$\frac{\partial f}{\partial \mathbf{A}} = \begin{bmatrix} \frac{\partial f}{\partial a_{11}} & \frac{\partial f}{\partial a_{12}} & \cdots \\ \frac{\partial f}{\partial a_{21}} & \frac{\partial f}{\partial a_{22}} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \tag{1.25}$$

Key identities for matrix derivatives:

1. Linear function: $f(\mathbf{x}) = \mathbf{a}^T\mathbf{x}$, then $\frac{\partial f}{\partial \mathbf{x}} = \mathbf{a}$
2. Quadratic form: $f(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x}$, then $\frac{\partial f}{\partial \mathbf{x}} = (\mathbf{A} + \mathbf{A}^T)\mathbf{x} = 2\mathbf{A}\mathbf{x}$ (if $\mathbf{A}$ is symmetric)
3. Matrix product: $f(\mathbf{A}) = \text{trace}(\mathbf{A}^T\mathbf{B})$, then $\frac{\partial f}{\partial \mathbf{A}} = \mathbf{B}$
4. For loss $\mathcal{L}(\mathbf{x}) = \frac{1}{2}\|\mathbf{y} - \mathbf{W}\mathbf{x}\|_2^2$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = (\mathbf{W}\mathbf{x} - \mathbf{y})\mathbf{x}^T \tag{1.26}$$

Notation, Dimensions, and Label Conventions

This section fixes the main symbols, tensor shapes, and label conventions used throughout the notes to avoid ambiguity across chapters.

Basic symbols and sets

- Scalars are denoted by lowercase letters (e.g., $x \in \mathbb{R}$), vectors by bold lowercase (e.g., $\mathbf{x} \in \mathbb{R}^d$), and matrices by uppercase (e.g., $A \in \mathbb{R}^{m \times n}$).
- The Euclidean norm is $\|\mathbf{v}\|_2$ and the Frobenius norm is $\|A\|_F$.
- The all-ones vector of length m is $\mathbf{1} \in \mathbb{R}^m$.

Data, mini-batches, and shapes

Let the input dimension be d, the number of classes be K, and the dataset size be N.

- Single example: $(\mathbf{x}, \mathbf{y})$ with $\mathbf{x} \in \mathbb{R}^d$.
- Mini-batch of size m (column-stacked convention):

$$X = \begin{bmatrix} \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \end{bmatrix} \in \mathbb{R}^{d \times m}.$$

  This matches the vectorized forward form used in the network chapters.

Network architecture and parameters

Consider an L-layer feedforward network with layer widths

$$n_0 = d, \quad n_1, \ldots, n_{L-1}, \quad n_L = \begin{cases} 1 & \text{binary output (sigmoid)} \\ K & K\text{-class output (softmax).} \end{cases}$$

The parameters are $\theta = \{W^{(\ell)}, \mathbf{b}^{(\ell)}\}_{\ell=1}^{L}$.

For each layer $\ell = 1, \ldots, L$:

$$W^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}, \qquad \mathbf{b}^{(\ell)} \in \mathbb{R}^{n_\ell}.$$

Forward pass (single example):

$$\mathbf{a}^{(0)} = \mathbf{x}, \qquad \mathbf{z}^{(\ell)} = W^{(\ell)}\mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}, \qquad \mathbf{a}^{(\ell)} = \sigma^{(\ell)}(\mathbf{z}^{(\ell)}).$$

Here $\mathbf{z}^{(\ell)}, \mathbf{a}^{(\ell)} \in \mathbb{R}^{n_\ell}$.

Vectorized mini-batch forward pass (column-stacked):

$$A^{(0)} = X \in \mathbb{R}^{n_0 \times m}, \qquad Z^{(\ell)} = W^{(\ell)}A^{(\ell-1)} + \mathbf{b}^{(\ell)}\mathbf{1}^\top \in \mathbb{R}^{n_\ell \times m}, \qquad A^{(\ell)} = \sigma^{(\ell)}(Z^{(\ell)}).$$

This is the same convention used in the notes' fully-vectorized section.

Backpropagation symbols

The loss for one example is denoted by $L(\hat{\mathbf{y}}, \mathbf{y})$.

The key backpropagation quantity is the delta:

$$\boldsymbol{\delta}^{(\ell)} \;=\; \frac{\partial L}{\partial \mathbf{z}^{(\ell)}} \in \mathbb{R}^{n_\ell}.$$

With this convention, gradients take the outer-product form (single example):

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

These formulas match the derivations in the backpropagation chapter.

For a mini-batch of size m, define the averaged objective (empirical risk on the batch)

$$\mathcal{L}_{\text{batch}}(\theta) = \frac{1}{m}\sum_{i=1}^{m} L\big(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}\big),$$

and compute averaged gradients accordingly (as in the mini-batch gradient section).

Label conventions (important)

Binary classification appears in two common encodings:

- **Probability/BCE convention:** $y \in \{0, 1\}$, $\hat{y} \in (0, 1)$ and $\hat{y} = \sigma(z)$ (sigmoid). This is the convention used in the BCE chapter and the worked numerical example.
- **Perceptron convention:** $y \in \{-1, +1\}$ and prediction $\hat{y} = \text{sign}(w^\top x + b)$, used for the perceptron convergence theorem statement.

A simple conversion between the two binary encodings is

$$y_{\pm 1} = 2y_{01} - 1, \qquad y_{01} = \frac{y_{\pm 1} + 1}{2}.$$

Use $y \in \{-1, +1\}$ when discussing the classic perceptron theorem, and $y \in \{0, 1\}$ when using sigmoid/BCE.

Multi-class classification uses one-hot vectors:

$$\mathbf{y} \in \{0, 1\}^K, \qquad \sum_{k=1}^{K} y_k = 1, \qquad \mathbf{p} = \text{softmax}(\mathbf{z}) \in (0, 1)^K, \qquad \sum_{k=1}^{K} p_k = 1.$$

This matches the softmax + categorical cross-entropy setup and its gradient simplification.

Common nonlinearities (quick reference)

- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$, $\sigma'(z) = \sigma(z)(1-\sigma(z))$.
- ReLU: $\mathrm{ReLU}(z) = \max(0, z)$ with subgradient $\mathrm{ReLU}'(z) = 1$ for $z > 0$, 0 for $z < 0$, and any value in $[0, 1]$ at $z = 0$.
- Softmax: $p_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$, Jacobian $\frac{\partial p_i}{\partial z_j} = p_i(\delta_{ij} - p_j)$.

Chapter 2

The Perceptron

2.1 Decision Boundary as a Hyperplane

The perceptron is a binary linear classifier. Given an input vector $\mathbf{x} \in \mathbb{R}^d$, weights $\mathbf{w} \in \mathbb{R}^d$, and bias $b \in \mathbb{R}$, define the *score*

$$z(\mathbf{x}) \;=\; \mathbf{w}^\top \mathbf{x} + b. \tag{2.1}$$

The perceptron prediction is

$$\hat{y}(\mathbf{x}) \;=\; \mathrm{step}(z(\mathbf{x})) \;=\; \begin{cases} 1 & \text{if } \mathbf{w}^\top \mathbf{x} + b \geq 0, \\ 0 & \text{if } \mathbf{w}^\top \mathbf{x} + b < 0. \end{cases} \tag{2.2}$$

(Equivalently, using labels $y \in \{-1, +1\}$ one often writes $\hat{y} = \mathrm{sign}(\mathbf{w}^\top \mathbf{x} + b)$.)

2.1.1 Hyperplane interpretation

The *decision boundary* is the set of points where the score is exactly zero:

$$\mathcal{H} \;=\; \{\mathbf{x} \in \mathbb{R}^d \,:\, \mathbf{w}^\top \mathbf{x} + b = 0\}. \tag{2.3}$$

This set $\mathcal{H}$ is a **hyperplane** in $\mathbb{R}^d$ with normal vector $\mathbf{w}$.

The hyperplane splits $\mathbb{R}^d$ into two half-spaces:

$$\mathcal{H}_+ = \{\mathbf{x} \in \mathbb{R}^d \,:\, \mathbf{w}^\top \mathbf{x} + b > 0\}, \tag{2.4}$$

$$\mathcal{H}_- = \{\mathbf{x} \in \mathbb{R}^d \,:\, \mathbf{w}^\top \mathbf{x} + b < 0\}. \tag{2.5}$$

Geometrically, $\mathbf{w}$ points toward the side classified as positive.

2.1.2 Scaling invariance

For any $\alpha > 0$, replacing $(\mathbf{w}, b)$ by $(\alpha\mathbf{w}, \alpha b)$ leaves the boundary unchanged:

$$(\alpha\mathbf{w})^\top \mathbf{x} + \alpha b = \alpha(\mathbf{w}^\top \mathbf{x} + b), \tag{2.6}$$

so $\mathbf{w}^\top \mathbf{x} + b = 0$ iff $(\alpha\mathbf{w})^\top \mathbf{x} + \alpha b = 0$. Only the *direction* of $\mathbf{w}$ and the *relative offset* b matter for classification.

2.2 Signed Distance to the Hyperplane

Let $\mathcal{H} = \{\mathbf{x} : \mathbf{w}^\top \mathbf{x} + b = 0\}$ with $\mathbf{w} \neq \mathbf{0}$. The **signed distance** from a point $\mathbf{x}$ to the hyperplane is

$$d(\mathbf{x}, \mathcal{H}) \;=\; \frac{\mathbf{w}^\top \mathbf{x} + b}{\|\mathbf{w}\|_2}. \tag{2.7}$$

2.2.1 Derivation

Pick any point $\mathbf{x}_0 \in \mathcal{H}$ so that $\mathbf{w}^\top \mathbf{x}_0 + b = 0$. The vector from $\mathbf{x}_0$ to $\mathbf{x}$ is $\mathbf{x} - \mathbf{x}_0$. Projecting this vector onto the unit normal direction $\mathbf{u} = \mathbf{w}/\|\mathbf{w}\|_2$ gives the signed distance:

$$d(\mathbf{x}, \mathcal{H}) = \mathbf{u}^\top (\mathbf{x} - \mathbf{x}_0) = \frac{\mathbf{w}^\top (\mathbf{x} - \mathbf{x}_0)}{\|\mathbf{w}\|_2} = \frac{\mathbf{w}^\top \mathbf{x} - \mathbf{w}^\top \mathbf{x}_0}{\|\mathbf{w}\|_2}. \tag{2.8}$$

Using $\mathbf{w}^\top \mathbf{x}_0 = -b$ yields (2.7). In particular:

- $d(\mathbf{x}, \mathcal{H}) > 0$ iff $\mathbf{x} \in \mathcal{H}_+$,
- $d(\mathbf{x}, \mathcal{H}) < 0$ iff $\mathbf{x} \in \mathcal{H}_-$,
- $|d(\mathbf{x}, \mathcal{H})|$ equals the Euclidean distance to the boundary.

2.3 Perceptron Learning Algorithm

2.3.1 Label convention

For the convergence theorem, it is standard to use labels $y_i \in \{-1, +1\}$. Given training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$, define the prediction

$$\hat{y}_i = \operatorname{sign}(\mathbf{w}^\top \mathbf{x}_i + b). \tag{2.9}$$

2.3.2 Update rule

Initialize $\mathbf{w}_0 = \mathbf{0}$ and $b_0 = 0$. At iteration t, pick a misclassified example $(\mathbf{x}_i, y_i)$ such that

$$y_i(\mathbf{w}_t^\top \mathbf{x}_i + b_t) \leq 0. \tag{2.10}$$

Then apply the perceptron update:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \eta\, y_i\, \mathbf{x}_i, \tag{2.11}$$

$$b_{t+1} = b_t + \eta\, y_i, \tag{2.12}$$

where $\eta > 0$ is the learning rate. (Equivalently, one may absorb the bias into an augmented vector $\tilde{\mathbf{x}} = [\mathbf{x}^\top, 1]^\top$ and weight $\tilde{\mathbf{w}} = [\mathbf{w}^\top, b]^\top$ to write a single update.)

2.4 Perceptron Convergence Theorem (Sketch)

2.4.1 Linear separability and margin

Assume the data is linearly separable: there exists $(\mathbf{w}_\star, b_\star)$ such that

$$y_i(\mathbf{w}_\star^\top \mathbf{x}_i + b_\star) \geq 1 \quad \text{for all } i. \tag{2.13}$$

Let $R = \max_i \|\mathbf{x}_i\|_2$. Define the (geometric) margin of the separator $(\mathbf{w}_\star, b_\star)$ as

$$\gamma \;=\; \min_i \frac{y_i(\mathbf{w}_\star^\top \mathbf{x}_i + b_\star)}{\|\mathbf{w}_\star\|_2}. \tag{2.14}$$

Under (2.13), one has $\gamma \geq 1/\|\mathbf{w}_\star\|_2$.

2.4.2 Theorem

Theorem (Perceptron convergence). If the training set is linearly separable with margin $\gamma > 0$ and $\|\mathbf{x}_i\| \leq R$, then the perceptron makes at most

$$M \;\leq\; \left(\frac{R}{\gamma}\right)^2 \tag{2.15}$$

mistakes (updates), hence it terminates after finitely many updates.

2.4.3 Proof sketch

For simplicity take $\eta = 1$ and use augmented vectors to include b (the same argument works without augmentation). Let $\mathbf{w}_t$ denote the weight after t updates.

Step 1: Progress along the optimal separator. For an update using misclassified $(\mathbf{x}_i, y_i)$:

$$\mathbf{w}_{t+1}^\top \mathbf{w}_\star = (\mathbf{w}_t + y_i \mathbf{x}_i)^\top \mathbf{w}_\star = \mathbf{w}_t^\top \mathbf{w}_\star + y_i\, \mathbf{x}_i^\top \mathbf{w}_\star. \tag{2.16}$$

By separability, $y_i\, \mathbf{x}_i^\top \mathbf{w}_\star \geq \gamma \|\mathbf{w}_\star\|_2$ (up to the bias/augmentation convention), so after M mistakes:

$$\mathbf{w}_M^\top \mathbf{w}_\star \;\geq\; M\, \gamma\, \|\mathbf{w}_\star\|_2. \tag{2.17}$$

Step 2: Norm growth is controlled. The squared norm evolves as

$$\|\mathbf{w}_{t+1}\|_2^2 = \|\mathbf{w}_t + y_i \mathbf{x}_i\|_2^2 = \|\mathbf{w}_t\|_2^2 + 2 y_i\, \mathbf{w}_t^\top \mathbf{x}_i + \|\mathbf{x}_i\|_2^2. \tag{2.18}$$

Because the point is misclassified, $y_i(\mathbf{w}_t^\top \mathbf{x}_i) \leq 0$, hence

$$\|\mathbf{w}_{t+1}\|_2^2 \leq \|\mathbf{w}_t\|_2^2 + \|\mathbf{x}_i\|_2^2 \leq \|\mathbf{w}_t\|_2^2 + R^2. \tag{2.19}$$

By induction,

$$\|\mathbf{w}_M\|_2^2 \leq M R^2 \quad \Rightarrow \quad \|\mathbf{w}_M\|_2 \leq R\sqrt{M}. \tag{2.20}$$

Step 3: Combine via Cauchy–Schwarz. By Cauchy–Schwarz,

$$\mathbf{w}_M^\top \mathbf{w}_\star \le \|\mathbf{w}_M\|_2 \, \|\mathbf{w}_\star\|_2. \tag{2.21}$$

Plugging (2.17) and (2.20):

$$M\,\gamma\,\|\mathbf{w}_\star\|_2 \le \|\mathbf{w}_M\|_2\,\|\mathbf{w}_\star\|_2 \le R\sqrt{M}\,\|\mathbf{w}_\star\|_2. \tag{2.22}$$

Cancel $\|\mathbf{w}_\star\|_2 > 0$ and rearrange:

$$M\gamma \le R\sqrt{M} \quad \Rightarrow \quad \sqrt{M} \le \frac{R}{\gamma} \quad \Rightarrow \quad M \le \left(\frac{R}{\gamma}\right)^2, \tag{2.23}$$

which proves (2.15).

2.4.4 Remarks

- If the data is *not* linearly separable, the perceptron update may cycle and never converge.
- The bound depends on R (data scale) and γ (separability margin). Larger margins imply fewer updates.

Chapter 3

Feedforward Neural Networks

3.1 From Scalar Sums to Matrix Form

A feedforward neural network generalizes the perceptron by stacking multiple affine maps and nonlinearities.

3.1.1 Single neuron (scalar form)

Let $\mathbf{x} \in \mathbb{R}^d$ be an input, weights $\mathbf{w} \in \mathbb{R}^d$, bias $b \in \mathbb{R}$. A single neuron computes

$$z = \sum_{i=1}^{d} w_i x_i + b = \mathbf{w}^\top \mathbf{x} + b, \tag{3.1}$$

then outputs an activation $a = \sigma(z)$ for some nonlinearity σ.

3.1.2 Layer of neurons (summation index notation)

Consider layer ℓ with $n_{\ell-1}$ inputs and n_ℓ neurons. Let $\mathbf{a}^{(\ell-1)} \in \mathbb{R}^{n_{\ell-1}}$ be the input activation vector to layer ℓ. For neuron $j \in \{1, \dots, n_\ell\}$, define weights $W_{jk}^{(\ell)}$ from input unit k to neuron j and bias $b_j^{(\ell)}$. Then

$$z_j^{(\ell)} = \sum_{k=1}^{n_{\ell-1}} W_{jk}^{(\ell)} \, a_k^{(\ell-1)} + b_j^{(\ell)}. \tag{3.2}$$

3.1.3 Layer of neurons (matrix form)

Collect all $z_j^{(\ell)}$ into a vector $\mathbf{z}^{(\ell)} \in \mathbb{R}^{n_\ell}$, and define

$$\mathbf{W}^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}, \quad \mathbf{b}^{(\ell)} \in \mathbb{R}^{n_\ell}, \quad \mathbf{a}^{(\ell-1)} \in \mathbb{R}^{n_{\ell-1}}. \tag{3.3}$$

Then (3.2) becomes the compact vector form:

$$\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)} \mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}. \tag{3.4}$$

Applying an activation function element-wise,

$$\mathbf{a}^{(\ell)} = \sigma\big(\mathbf{z}^{(\ell)}\big). \tag{3.5}$$

3.1.4 Mini-batch (fully vectorized) form

For a mini-batch of size m, stack inputs as a matrix

$$\mathbf{A}^{(\ell-1)} = \begin{bmatrix} \mathbf{a}_1^{(\ell-1)} & \cdots & \mathbf{a}_m^{(\ell-1)} \end{bmatrix} \in \mathbb{R}^{n_{\ell-1} \times m}. \tag{3.6}$$

Then the affine map becomes

$$\mathbf{Z}^{(\ell)} = \mathbf{W}^{(\ell)} \mathbf{A}^{(\ell-1)} + \mathbf{b}^{(\ell)} \mathbf{1}^\top \in \mathbb{R}^{n_\ell \times m}, \tag{3.7}$$

where $\mathbf{1} \in \mathbb{R}^m$ is the all-ones vector. Activations:

$$\mathbf{A}^{(\ell)} = \sigma\big(\mathbf{Z}^{(\ell)}\big). \tag{3.8}$$

This matrix form is the basis of efficient GPU computation.

3.2 Network as Function Composition

Let the input layer be $\mathbf{a}^{(0)} = \mathbf{x} \in \mathbb{R}^{n_0}$. A depth-$L$ feedforward network is defined recursively by (3.4)–(3.5) for $\ell = 1, \ldots, L$.

The network output is

$$\hat{\mathbf{y}} = f(\mathbf{x}; \theta) = \mathbf{a}^{(L)}, \tag{3.9}$$

where $\theta = \{\mathbf{W}^{(\ell)}, \mathbf{b}^{(\ell)}\}_{\ell=1}^{L}$ is the set of parameters.

Writing out the full composition:

$$f(\mathbf{x}; \theta) = \sigma^{(L)}\Big(\mathbf{W}^{(L)}\, \sigma^{(L-1)}\big(\mathbf{W}^{(L-1)} \cdots \sigma^{(1)}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \cdots + \mathbf{b}^{(L-1)}\big) + \mathbf{b}^{(L)}\Big), \tag{3.10}$$

where $\sigma^{(\ell)}$ may differ by layer (e.g., ReLU in hidden layers and sigmoid/softmax at the output).

3.2.1 Parameter count

The number of trainable parameters is

$$\#\theta = \sum_{\ell=1}^{L} (n_\ell n_{\ell-1} + n_\ell) = \sum_{\ell=1}^{L} n_\ell (n_{\ell-1} + 1). \tag{3.11}$$

Large n_ℓ or large depth L increases capacity but also risks overfitting without regularization.

3.3 Activation Functions and Derivatives

Nonlinear activations are essential: without them, the entire network collapses to a single affine map.

3.3.1 Why nonlinearity is required

If $\sigma(z) = z$ for all layers (purely linear network), then

$$\mathbf{a}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}. \tag{3.12}$$

By induction, the entire network becomes

$$f(\mathbf{x}) = \mathbf{W}_{\text{eff}}\mathbf{x} + \mathbf{b}_{\text{eff}}, \tag{3.13}$$

for some effective matrix/vector $(\mathbf{W}_{\text{eff}}, \mathbf{b}_{\text{eff}})$, hence depth gives no additional expressive power. Therefore, σ must be nonlinear.

3.3.2 Sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{3.14}$$

Derivative:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)). \tag{3.15}$$

3.3.3 Hyperbolic tangent

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}. \tag{3.16}$$

Derivative:

$$\frac{d}{dz}\tanh(z) = 1 - \tanh^2(z). \tag{3.17}$$

3.3.4 ReLU

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & (z > 0), \\ 0 & (z \le 0). \end{cases} \tag{3.18}$$

Subgradient (used in practice):

$$\text{ReLU}'(z) = \begin{cases} 1 & (z > 0), \\ 0 & (z < 0), \\ \text{any value in } [0,1] & (z = 0). \end{cases} \tag{3.19}$$

3.3.5 Softmax

For $\mathbf{z} \in \mathbb{R}^K$,

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad i = 1, \dots, K. \tag{3.20}$$

Its Jacobian is

$$\frac{\partial \,\text{softmax}(\mathbf{z})_i}{\partial z_j} = p_i(\delta_{ij} - p_j), \tag{3.21}$$

where $\mathbf{p} = \text{softmax}(\mathbf{z})$ and δ_{ij} is the Kronecker delta. Matrix form:

$$\mathbf{J} = \text{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^{\top}. \tag{3.22}$$

3.3.6 Vanishing gradients (preview)

For sigmoid, $\sigma'(z) \leq 1/4$; repeated multiplication of such terms across many layers can make gradients small:

$$\prod_{\ell=1}^{L} \sigma'\big(z^{(\ell)}\big) \quad \text{tends to 0 as } L \text{ grows (in many regimes).} \tag{3.23}$$

This motivates ReLU-family activations and normalization methods, discussed later.

3.4 A Fully Worked Tiny Example (Optional)

Consider a $2 \to 2 \to 1$ network with ReLU hidden layer and sigmoid output. Let

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \mathbf{W}^{(1)} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}, \quad \mathbf{b}^{(1)} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}. \tag{3.24}$$

Hidden pre-activations:

$$\mathbf{z}^{(1)} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + b_1 \\ w_{21}x_1 + w_{22}x_2 + b_2 \end{bmatrix}, \tag{3.25}$$

hidden activations:

$$\mathbf{a}^{(1)} = \begin{bmatrix} \max(0, z_1^{(1)}) \\ \max(0, z_2^{(1)}) \end{bmatrix}. \tag{3.26}$$

Output layer:

$$z^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + b^{(2)} = u_1 a_1^{(1)} + u_2 a_2^{(1)} + b^{(2)}, \tag{3.27}$$

$$\hat{y} = \sigma(z^{(2)}) = \frac{1}{1 + e^{-z^{(2)}}}. \tag{3.28}$$

This concrete form makes it easy to check dimensions and confirm that each layer is an affine map followed by a nonlinearity.

Chapter 4

Loss Functions

A **loss function** quantifies how far a model prediction is from the target. Training typically minimizes the empirical risk (average loss over a dataset). Let $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{m}$ be a dataset and let the model output be $\hat{\mathbf{y}}^{(i)} = f_\theta(\mathbf{x}^{(i)})$.

4.1 Empirical Risk Minimization

Given a per-sample loss $\ell(\hat{\mathbf{y}}, \mathbf{y})$, define the empirical risk

$$\mathcal{L}(\theta) \;=\; \frac{1}{m}\sum_{i=1}^{m} \ell\big(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}\big)\,. \tag{4.1}$$

Gradient-based learning requires computing $\nabla_\theta \mathcal{L}(\theta)$, so we will derive gradients for common losses.

4.2 Regression: Mean Squared Error

4.2.1 Definition

For regression with $\mathbf{y} \in \mathbb{R}^K$ and prediction $\hat{\mathbf{y}} \in \mathbb{R}^K$, the Mean Squared Error (MSE) loss is

$$\ell_{\text{MSE}}(\hat{\mathbf{y}}, \mathbf{y}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2 = \sum_{k=1}^{K} (\hat{y}_k - y_k)^2. \tag{4.2}$$

A common scaled variant uses $\frac{1}{2}\|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$ to remove a factor 2 in gradients.

Over a dataset:

$$\mathcal{L}_{\text{MSE}}(\theta) = \frac{1}{m}\sum_{i=1}^{m} \|\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\|_2^2. \tag{4.3}$$

4.2.2 Gradient w.r.t. the prediction

From (4.2), the gradient w.r.t. $\hat{\mathbf{y}}$ is

$$\nabla_{\hat{\mathbf{y}}}\, \ell_{\text{MSE}}(\hat{\mathbf{y}}, \mathbf{y}) = 2(\hat{\mathbf{y}} - \mathbf{y}). \tag{4.4}$$

For the scaled loss $\frac{1}{2}\|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$, the gradient becomes $\hat{\mathbf{y}} - \mathbf{y}$.

4.3 Binary Classification: Binary Cross-Entropy

Binary classification assumes $y \in \{0, 1\}$ and model output $\hat{y} \in (0, 1)$ interpreted as $P(Y = 1 \mid \mathbf{x})$. Often $\hat{y} = \sigma(z)$ where $z \in \mathbb{R}$ is the logit and σ is the sigmoid.

4.3.1 Binary cross-entropy (negative log-likelihood)

The Binary Cross-Entropy (BCE) loss for one example is

$$\ell_{\text{BCE}}(\hat{y}, y) = -\Big[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\Big]. \tag{4.5}$$

Over a dataset:

$$\mathcal{L}_{\text{BCE}}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \ell_{\text{BCE}}(\hat{y}^{(i)}, y^{(i)}). \tag{4.6}$$

4.3.2 Gradient w.r.t. $\hat{y}$

Differentiate (4.5):

$$\frac{\partial \ell_{\text{BCE}}}{\partial \hat{y}} = -\left(\frac{y}{\hat{y}} - \frac{1 - y}{1 - \hat{y}}\right) = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})}. \tag{4.7}$$

4.3.3 Sigmoid + BCE simplification (logit gradient)

Let $\hat{y} = \sigma(z)$ with $\sigma'(z) = \hat{y}(1 - \hat{y})$ (from Chapter 3). By the chain rule:

$$\frac{\partial \ell_{\text{BCE}}}{\partial z} = \frac{\partial \ell_{\text{BCE}}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \cdot \hat{y}(1 - \hat{y}) = \hat{y} - y. \tag{4.8}$$

Thus, **with sigmoid + BCE, the error signal at the logit is simply prediction minus target**.

4.4 Multi-class Classification: Softmax and Categorical Cross-Entropy

Assume K classes. Let logits be $\mathbf{z} \in \mathbb{R}^K$ and probabilities be

$$\mathbf{p} = \text{softmax}(\mathbf{z}), \qquad p_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}. \tag{4.9}$$

Let the label be one-hot $\mathbf{y} \in \{0, 1\}^K$ with $\sum_k y_k = 1$.

4.4.1 Categorical cross-entropy

The Categorical Cross-Entropy (CCE) loss for one example is

$$\ell_{\text{CCE}}(\mathbf{p}, \mathbf{y}) = -\sum_{k=1}^{K} y_k \log(p_k). \tag{4.10}$$

Over a dataset:

$$\mathcal{L}_{\text{CCE}}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \ell_{\text{CCE}}(\mathbf{p}^{(i)}, \mathbf{y}^{(i)}). \tag{4.11}$$

4.4.2 Softmax Jacobian

The derivative of softmax has the form (Jacobian):

$$\frac{\partial p_i}{\partial z_j} = p_i(\delta_{ij} - p_j), \tag{4.12}$$

where δ_{ij} is the Kronecker delta ($\delta_{ij} = 1$ if $i = j$, else 0).

Equivalently, in matrix form:

$$\frac{\partial \mathbf{p}}{\partial \mathbf{z}} = \text{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top. \tag{4.13}$$

4.4.3 Softmax + CCE simplification (logit gradient)

We now compute $\nabla_{\mathbf{z}} \ell_{\text{CCE}}$. First,

$$\frac{\partial \ell_{\text{CCE}}}{\partial p_i} = -\frac{y_i}{p_i}. \tag{4.14}$$

Then apply chain rule:

$$\frac{\partial \ell_{\text{CCE}}}{\partial z_j} = \sum_{i=1}^{K} \frac{\partial \ell_{\text{CCE}}}{\partial p_i} \frac{\partial p_i}{\partial z_j} = \sum_{i=1}^{K} \left(-\frac{y_i}{p_i}\right) p_i(\delta_{ij} - p_j). \tag{4.15}$$

Cancel p_i:

$$= -\sum_{i=1}^{K} y_i(\delta_{ij} - p_j) = -\sum_{i=1}^{K} y_i\delta_{ij} + \sum_{i=1}^{K} y_i p_j. \tag{4.16}$$

Since $\sum_{i=1}^{K} y_i\delta_{ij} = y_j$ and $\sum_{i=1}^{K} y_i = 1$,

$$\frac{\partial \ell_{\text{CCE}}}{\partial z_j} = -p_j + p_j \cdot 1 + (p_j - y_j) = p_j - y_j. \tag{4.17}$$

Therefore,

$$\nabla_{\mathbf{z}} \ell_{\text{CCE}}(\text{softmax}(\mathbf{z}), \mathbf{y}) = \mathbf{p} - \mathbf{y}. \tag{4.18}$$

This is the multi-class analogue of (4.8).

4.5 (Optional) Huber Loss for Robust Regression

When regression data contains outliers, MSE can be overly sensitive. The Huber loss interpolates between MSE and MAE.

For scalar $y, \hat{y} \in \mathbb{R}$ with threshold $\delta > 0$:

$$\ell_{\text{Huber}}(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \le \delta, \\ \delta\left(|y - \hat{y}| - \frac{\delta}{2}\right) & \text{if } |y - \hat{y}| > \delta. \end{cases} \tag{4.19}$$

Its derivative w.r.t. $\hat{y}$ is

$$\frac{\partial \ell_{\text{Huber}}}{\partial \hat{y}} = \begin{cases} \hat{y} - y & \text{if } |y - \hat{y}| \le \delta, \\ \delta\,\text{sign}(\hat{y} - y) & \text{if } |y - \hat{y}| > \delta. \end{cases} \tag{4.20}$$

Chapter 5

Backpropagation Algorithm

5.1 Setup: Notation and Forward Pass

Consider an L-layer feedforward neural network. For a single example $(\mathbf{x}, \mathbf{y})$, define

$$\mathbf{a}^{(0)} = \mathbf{x}. \tag{5.1}$$

For each layer $\ell = 1, 2, \ldots, L$:

$$\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}, \tag{5.2}$$

$$\mathbf{a}^{(\ell)} = \sigma^{(\ell)}\big(\mathbf{z}^{(\ell)}\big), \tag{5.3}$$

where $\mathbf{W}^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$, $\mathbf{b}^{(\ell)} \in \mathbb{R}^{n_\ell}$, and $\sigma^{(\ell)}$ is applied element-wise unless stated otherwise.

Let the prediction be $\hat{\mathbf{y}} = \mathbf{a}^{(L)}$ and the loss be

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \mathcal{L}(\mathbf{a}^{(L)}, \mathbf{y}). \tag{5.4}$$

The goal of backpropagation is to compute the gradients $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}}$ efficiently for all ℓ.

5.2 The Delta Terms

5.2.1 Definition

Define the **delta** (error signal) at layer ℓ by

$$\boldsymbol{\delta}^{(\ell)} \;=\; \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \in \mathbb{R}^{n_\ell}. \tag{5.5}$$

Once $\boldsymbol{\delta}^{(\ell)}$ is known, the parameter gradients follow in simple outer-product form (see Section 5.3).

5.2.2 Output layer delta: general form

By the chain rule,

$$\boldsymbol{\delta}^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(L)}} \odot \frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{z}^{(L)}} = \nabla_{\mathbf{a}^{(L)}}\mathcal{L} \;\odot\; \sigma^{(L)\prime}\big(\mathbf{z}^{(L)}\big), \tag{5.6}$$

where $\odot$ denotes element-wise multiplication.

5.2.3 Hidden layer delta

For $\ell = L-1, L-2, \ldots, 1$, backpropagate the delta:

$$\boldsymbol{\delta}^{(\ell)} = \left(\mathbf{W}^{(\ell+1)}\right)^{\top} \boldsymbol{\delta}^{(\ell+1)} \odot \sigma^{(\ell)\prime}\left(\mathbf{z}^{(\ell)}\right). \tag{5.7}$$

Interpretation:

- $\left(\mathbf{W}^{(\ell+1)}\right)^{\top} \boldsymbol{\delta}^{(\ell+1)}$ maps the error signal back to layer ℓ.
- Multiplying by $\sigma^{(\ell)\prime}$ accounts for the nonlinearity at layer ℓ.

This is the central recurrence relation of backpropagation.

5.3 Gradients for Weights and Biases

5.3.1 Single example

From (5.2), each component is

$$z_i^{(\ell)} = \sum_{j=1}^{n_{\ell-1}} W_{ij}^{(\ell)} a_j^{(\ell-1)} + b_i^{(\ell)}. \tag{5.8}$$

Hence,

$$\frac{\partial z_i^{(\ell)}}{\partial W_{ij}^{(\ell)}} = a_j^{(\ell-1)}, \quad \frac{\partial z_i^{(\ell)}}{\partial b_i^{(\ell)}} = 1, \quad \frac{\partial z_i^{(\ell)}}{\partial b_k^{(\ell)}} = 0 \ (k \neq i). \tag{5.9}$$

Using $\delta_i^{(\ell)} = \frac{\partial \mathcal{L}}{\partial z_i^{(\ell)}}$, we obtain

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial z_i^{(\ell)}} \frac{\partial z_i^{(\ell)}}{\partial W_{ij}^{(\ell)}} = \delta_i^{(\ell)} \, a_j^{(\ell-1)}. \tag{5.10}$$

In matrix form:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)} \left(\mathbf{a}^{(\ell-1)}\right)^{\top}. \tag{5.11}$$

Similarly, for the bias:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}. \tag{5.12}$$

These match the compact formulas already appearing in the draft.

5.3.2 Mini-batch (average gradient)

For a mini-batch of size m, with deltas $\boldsymbol{\delta}^{(\ell),(i)}$ and activations $\mathbf{a}^{(\ell-1),(i)}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}} = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{\delta}^{(\ell),(i)} \left(\mathbf{a}^{(\ell-1),(i)}\right)^{\top}, \tag{5.13}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{\delta}^{(\ell),(i)}. \tag{5.14}$$

(Implementation note: in vectorized code, one stacks examples as matrices and these become matrix multiplications.)

5.3.3 Mini-batch matrix backpropagation (vectorized deltas)

We now rewrite the delta recursion in a fully vectorized mini-batch form, consistent with the matrix forward pass (Section 3.1.4) and the numerical vectorization in Chapter 6.

Mini-batch notation. Let the mini-batch size be m and stack activations as columns:

$$A^{(\ell)} = \left[a^{(\ell),(1)}, \dots, a^{(\ell),(m)}\right] \in \mathbb{R}^{n_\ell \times m}, \qquad Z^{(\ell)} = \left[z^{(\ell),(1)}, \dots, z^{(\ell),(m)}\right] \in \mathbb{R}^{n_\ell \times m}. \tag{5.15}$$

The vectorized forward pass is

$$Z^{(\ell)} = W^{(\ell)} A^{(\ell-1)} + b^{(\ell)} \mathbf{1}^\top, \qquad A^{(\ell)} = \sigma^{(\ell)}\big(Z^{(\ell)}\big), \tag{5.16}$$

where $\mathbf{1} \in \mathbb{R}^m$ is the all-ones vector and $\sigma^{(\ell)}$ is applied element-wise.

Vectorized deltas. Define the mini-batch delta matrix

$$\Delta^{(\ell)} \;=\; \frac{\partial \mathcal{L}_{\text{batch}}}{\partial Z^{(\ell)}} \in \mathbb{R}^{n_\ell \times m}, \tag{5.17}$$

where $\mathcal{L}_{\text{batch}}$ denotes the average loss over the mini-batch.

Output layer delta (matrix form). In complete generality, for the output layer $\ell = L$,

$$\Delta^{(L)} = \left(\frac{\partial \mathcal{L}_{\text{batch}}}{\partial A^{(L)}}\right) \odot \sigma^{(L)\prime}\big(Z^{(L)}\big), \tag{5.18}$$

where $\odot$ denotes element-wise multiplication.

Hidden layer delta recursion (matrix form). For $\ell = L-1, \dots, 1$, the hidden-layer deltas satisfy the vectorized recurrence

$$\Delta^{(\ell)} = \big(W^{(\ell+1)}\big)^\top \Delta^{(\ell+1)} \odot \sigma^{(\ell)\prime}\big(Z^{(\ell)}\big). \tag{5.19}$$

This is exactly the single-example formula (5.7) applied to all m columns in parallel.

Gradients from vectorized deltas. Using (5.16) and the definition of $\Delta^{(\ell)}$, the parameter gradients become

$$\frac{\partial \mathcal{L}_{\text{batch}}}{\partial W^{(\ell)}} = \frac{1}{m} \Delta^{(\ell)} \big(A^{(\ell-1)}\big)^\top \in \mathbb{R}^{n_\ell \times n_{\ell-1}}, \tag{5.20}$$

$$\frac{\partial \mathcal{L}_{\text{batch}}}{\partial b^{(\ell)}} = \frac{1}{m} \Delta^{(\ell)} \mathbf{1} \in \mathbb{R}^{n_\ell}. \tag{5.21}$$

The bias gradient follows because $b^{(\ell)}\mathbf{1}^\top$ replicates $b^{(\ell)}$ across the m columns.

Consistency check (shapes).

$$\big(W^{(\ell+1)}\big)^\top \Delta^{(\ell+1)} \in \mathbb{R}^{n_\ell \times m}, \qquad \Delta^{(\ell)} \big(A^{(\ell-1)}\big)^\top \in \mathbb{R}^{n_\ell \times n_{\ell-1}}, \tag{5.22}$$

so all matrix multiplications are dimensionally consistent.

Two common output simplifications (mini-batch). For the standard paired choices already derived in Section 5.4:

- **Sigmoid + BCE (binary)**: if $A^{(L)} = \hat{Y} \in \mathbb{R}^{1\times m}$ and $Y \in \mathbb{R}^{1\times m}$, then
$$\Delta^{(L)} = \hat{Y} - Y. \tag{5.23}$$
- **Softmax + CCE (multi-class)**: if $A^{(L)} = P \in \mathbb{R}^{K\times m}$ and one-hot labels $Y \in \mathbb{R}^{K\times m}$, then
$$\Delta^{(L)} = P - Y. \tag{5.24}$$

These are the column-wise extensions of (5.18) and (5.21).

5.4 Two Classic "Cancellations"

Backprop becomes particularly simple for common output-layer choices.

5.4.1 Sigmoid + Binary Cross-Entropy

Assume a single output logit z and sigmoid activation
$$\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}, \qquad \sigma'(z) = \hat{y}(1-\hat{y}). \tag{5.25}$$

Binary cross-entropy loss:
$$\mathcal{L}(\hat{y}, y) = -\left[y \log(\hat{y}) + (1-y)\log(1-\hat{y})\right]. \tag{5.26}$$

First,
$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = -\left(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}\right) = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}. \tag{5.27}$$

Then by chain rule:
$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} \cdot \hat{y}(1-\hat{y}) = \hat{y} - y. \tag{5.28}$$

Thus, the output delta is simply "prediction minus target."

5.4.2 Softmax + Categorical Cross-Entropy

Let logits be $\mathbf{z} \in \mathbb{R}^K$ and softmax probabilities
$$p_k = \text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}. \tag{5.29}$$

With one-hot label vector $\mathbf{y} \in \{0,1\}^K$, categorical cross-entropy:
$$\mathcal{L}(\mathbf{p}, \mathbf{y}) = -\sum_{k=1}^{K} y_k \log p_k. \tag{5.30}$$

A key result (derivable from the Jacobian of softmax) is:
$$\boldsymbol{\delta}^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \mathbf{p} - \mathbf{y}. \tag{5.31}$$

Again, the delta is "predicted probabilities minus true probabilities."

5.5 Vanishing Gradients (Why Depth Is Hard)

Equation (5.7) shows that earlier-layer deltas are products of many factors. In a deep network, schematically:

$$\boldsymbol{\delta}^{(1)} \approx \left(\prod_{\ell=2}^{L} \left(\mathbf{W}^{(\ell)}\right)^{\top} \mathrm{Diag}\left(\sigma^{(\ell)\prime}(\mathbf{z}^{(\ell)})\right) \right) \boldsymbol{\delta}^{(L)}. \tag{5.32}$$

If $\sigma^{(\ell)}$ is sigmoid, then $\sigma'(z) \leq 1/4$ for all z. Multiplying many numbers $\leq 1/4$ tends to drive the magnitude of gradients toward zero, slowing learning in early layers.

5.5.1 Mitigations (mathematical view)

- ReLU-type activations: $\mathrm{ReLU}'(z) = 1$ for $z > 0$, avoiding a ubiquitous small factor.
- Careful initialization to keep activations and gradients in a reasonable scale.
- Normalization (e.g., batch normalization) and skip connections to improve gradient flow.

These ideas motivate much of modern deep learning architecture design.

5.6 Algorithm Summary (One Iteration)

For one mini-batch:

1. Forward pass: compute $(\mathbf{z}^{(\ell)}, \mathbf{a}^{(\ell)})$ for $\ell = 1, \ldots, L$ using (5.2)–(5.3).
2. Output delta: compute $\boldsymbol{\delta}^{(L)}$ using (5.6) (or the simplified forms (5.28), (5.31) when applicable).
3. Backward recursion: compute $\boldsymbol{\delta}^{(\ell)}$ for $\ell = L-1, \ldots, 1$ using (5.7).
4. Gradients: compute $\partial\mathcal{L}/\partial\mathbf{W}^{(\ell)}$ and $\partial\mathcal{L}/\partial\mathbf{b}^{(\ell)}$ using (5.13)–(5.14).
5. Update parameters with an optimizer (SGD, momentum, Adam, etc.).

This completes one training step.

Chapter 6

Concrete Numerical Example: A Network from Scratch

This chapter constructs a tiny feedforward neural network for binary classification and derives forward propagation, loss computation, and backpropagation *numerically and symbolically*. The goal is to see every quantity (z, a, δ, gradients) explicitly.

6.1 Problem Setup

We consider a toy dataset with $m = 4$ samples, input dimension $d = 2$, and binary labels $y \in \{0, 1\}$:

$$\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{4}, \qquad \mathbf{x}^{(i)} \in \mathbb{R}^2,\ y^{(i)} \in \{0, 1\}. \tag{6.1}$$

Concretely,

i	$\mathbf{x}^{(i)}$	$y^{(i)}$
1	$[0.5,\ 0.2]^\top$	0
2	$[0.9,\ 0.8]^\top$	1
3	$[0.1,\ 0.3]^\top$	0
4	$[0.8,\ 0.9]^\top$	1

(6.2)

We use a $2 \to 3 \to 1$ network:

- Hidden layer: $n_1 = 3$ neurons with ReLU.
- Output layer: $n_2 = 1$ neuron with sigmoid producing $\hat{y} \in (0, 1)$.

6.2 Parameter Initialization

Layer 1 parameters:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \\ 0.5 & 0.6 \end{bmatrix} \in \mathbb{R}^{3\times 2}, \qquad \mathbf{b}^{(1)} = \begin{bmatrix} 0.01 \\ 0.02 \\ 0.03 \end{bmatrix} \in \mathbb{R}^3. \tag{6.3}$$

Layer 2 parameters:

$$\mathbf{W}^{(2)} = \begin{bmatrix} 0.7 & 0.8 & 0.9 \end{bmatrix} \in \mathbb{R}^{1\times 3}, \qquad b^{(2)} = 0.1. \tag{6.4}$$

6.3 Forward Propagation: General Form

For each sample $\mathbf{x}$:

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}, \tag{6.5}$$

$$\mathbf{a}^{(1)} = \mathrm{ReLU}(\mathbf{z}^{(1)}), \tag{6.6}$$

$$z^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + b^{(2)}, \tag{6.7}$$

$$\hat{y} = \sigma(z^{(2)}) = \frac{1}{1 + e^{-z^{(2)}}}. \tag{6.8}$$

We use binary cross-entropy (per-sample loss):

$$\mathcal{L}^{(i)} \;=\; -\Big(y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})\Big). \tag{6.9}$$

Batch loss:

$$\mathcal{L}_{\text{batch}} = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}^{(i)}. \tag{6.10}$$

6.4 Forward Propagation: Sample 1 (Fully Expanded)

Let $\mathbf{x}^{(1)} = [0.5, 0.2]^\top$, $y^{(1)} = 0$.

6.4.1 Layer 1 pre-activation

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x}^{(1)} + \mathbf{b}^{(1)}. \tag{6.11}$$

Compute entrywise:

$$z_1^{(1)} = 0.1 \cdot 0.5 + 0.2 \cdot 0.2 + 0.01 = 0.05 + 0.04 + 0.01 = 0.10, \tag{6.12}$$

$$z_2^{(1)} = 0.3 \cdot 0.5 + 0.4 \cdot 0.2 + 0.02 = 0.15 + 0.08 + 0.02 = 0.25, \tag{6.13}$$

$$z_3^{(1)} = 0.5 \cdot 0.5 + 0.6 \cdot 0.2 + 0.03 = 0.25 + 0.12 + 0.03 = 0.40. \tag{6.14}$$

Thus

$$\mathbf{z}^{(1)} = \begin{bmatrix} 0.10 \\ 0.25 \\ 0.40 \end{bmatrix}. \tag{6.15}$$

6.4.2 Layer 1 activation (ReLU)

$$\mathbf{a}^{(1)} = \mathrm{ReLU}(\mathbf{z}^{(1)}) = \begin{bmatrix} \max(0, 0.10) \\ \max(0, 0.25) \\ \max(0, 0.40) \end{bmatrix} = \begin{bmatrix} 0.10 \\ 0.25 \\ 0.40 \end{bmatrix}. \tag{6.16}$$

6.4.3 Layer 2 pre-activation

$$z^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + b^{(2)} \tag{6.17}$$

$$= 0.7 \cdot 0.10 + 0.8 \cdot 0.25 + 0.9 \cdot 0.40 + 0.1 \tag{6.18}$$

$$= 0.07 + 0.20 + 0.36 + 0.10 = 0.73. \tag{6.19}$$

6.4.4 Output (sigmoid)

$$\hat{y}^{(1)} = \sigma(0.73) = \frac{1}{1 + e^{-0.73}}. \tag{6.20}$$

Using $e^{-0.73} \approx 0.4819$:

$$\hat{y}^{(1)} \approx \frac{1}{1 + 0.4819} = \frac{1}{1.4819} \approx 0.6748. \tag{6.21}$$

6.4.5 Loss for sample 1

Since $y^{(1)} = 0$, the loss simplifies from (6.9) to

$$\mathcal{L}^{(1)} = -\log(1 - \hat{y}^{(1)}) = -\log(1 - 0.6748) = -\log(0.3252). \tag{6.22}$$

Numerically,

$$\mathcal{L}^{(1)} \approx 1.1223. \tag{6.23}$$

6.5 Forward Propagation: Sample 2 (Detailed)

Let $\mathbf{x}^{(2)} = [0.9, 0.8]^\top$, $y^{(2)} = 1$.

6.5.1 Layer 1

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x}^{(2)} + \mathbf{b}^{(1)}. \tag{6.24}$$

Entrywise:

$$z_1^{(1)} = 0.1 \cdot 0.9 + 0.2 \cdot 0.8 + 0.01 = 0.09 + 0.16 + 0.01 = 0.26, \tag{6.25}$$

$$z_2^{(1)} = 0.3 \cdot 0.9 + 0.4 \cdot 0.8 + 0.02 = 0.27 + 0.32 + 0.02 = 0.61, \tag{6.26}$$

$$z_3^{(1)} = 0.5 \cdot 0.9 + 0.6 \cdot 0.8 + 0.03 = 0.45 + 0.48 + 0.03 = 0.96. \tag{6.27}$$

Since all entries are positive,

$$\mathbf{a}^{(1)} = \mathrm{ReLU}(\mathbf{z}^{(1)}) = \begin{bmatrix} 0.26 \\ 0.61 \\ 0.96 \end{bmatrix}. \tag{6.28}$$

6.5.2 Layer 2 and output

$$z^{(2)} = 0.7 \cdot 0.26 + 0.8 \cdot 0.61 + 0.9 \cdot 0.96 + 0.1 \tag{6.29}$$

$$= 0.182 + 0.488 + 0.864 + 0.1 = 1.634, \tag{6.30}$$

$$\hat{y}^{(2)} = \sigma(1.634) = \frac{1}{1 + e^{-1.634}} \approx 0.8367. \tag{6.31}$$

Loss (since $y^{(2)} = 1$):

$$\mathcal{L}^{(2)} = -\log(\hat{y}^{(2)}) \approx -\log(0.8367) \approx 0.1779. \tag{6.32}$$

6.6 Batch Loss

Assume the remaining sample losses are (as in the current draft):

$$\mathcal{L}^{(3)} \approx 0.9502, \qquad \mathcal{L}^{(4)} \approx 0.2148. \tag{6.33}$$

Then the batch loss is

$$\mathcal{L}_{\text{batch}} = \frac{1}{4}\left(\mathcal{L}^{(1)} + \mathcal{L}^{(2)} + \mathcal{L}^{(3)} + \mathcal{L}^{(4)}\right) \tag{6.34}$$

$$= \frac{1}{4}(1.1223 + 0.1779 + 0.9502 + 0.2148) \tag{6.35}$$

$$= \frac{2.4652}{4} \approx 0.6163. \tag{6.36}$$

6.7 Backpropagation: General Form

Define the output delta (for sigmoid + binary cross-entropy) as

$$\delta^{(2)} \;=\; \frac{\partial \mathcal{L}}{\partial z^{(2)}} \;=\; \hat{y} - y. \tag{6.37}$$

Then

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} = \delta^{(2)}(\mathbf{a}^{(1)})^{\top}, \tag{6.38}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(2)}} = \delta^{(2)}. \tag{6.39}$$

For the hidden layer (ReLU):

$$\boldsymbol{\delta}^{(1)} \;=\; \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} \;=\; (\mathbf{W}^{(2)})^{\top}\delta^{(2)} \;\odot\; \mathrm{ReLU}'(\mathbf{z}^{(1)}), \tag{6.40}$$

and

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} = \boldsymbol{\delta}^{(1)}(\mathbf{x})^{\top}, \tag{6.41}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}} = \boldsymbol{\delta}^{(1)}. \tag{6.42}$$

Here

$$\mathrm{ReLU}'(z) = \begin{cases} 1 & (z > 0), \\ 0 & (z \le 0). \end{cases} \tag{6.43}$$

6.8 Backpropagation: Sample 1 (Fully Expanded)

For sample 1, $y^{(1)} = 0$ and $\hat{y}^{(1)} \approx 0.6748$.

6.8.1 Output delta

From (6.37),

$$\delta_1^{(2)} = \hat{y}^{(1)} - y^{(1)} = 0.6748 - 0 = 0.6748. \tag{6.44}$$

6.8.2 Gradients for layer 2

$$\frac{\partial \mathcal{L}^{(1)}}{\partial \mathbf{W}^{(2)}} = \delta_1^{(2)} (\mathbf{a}^{(1)})^\top \tag{6.45}$$

$$= 0.6748 \cdot [0.10,\ 0.25,\ 0.40] \tag{6.46}$$

$$= [0.06748,\ 0.16870,\ 0.26992], \tag{6.47}$$

and

$$\frac{\partial \mathcal{L}^{(1)}}{\partial b^{(2)}} = \delta_1^{(2)} = 0.6748. \tag{6.48}$$

6.8.3 Hidden delta

Using (6.40):

$$(\mathbf{W}^{(2)})^\top \delta_1^{(2)} = \begin{bmatrix} 0.7 \\ 0.8 \\ 0.9 \end{bmatrix} 0.6748 = \begin{bmatrix} 0.47236 \\ 0.53984 \\ 0.60732 \end{bmatrix}. \tag{6.49}$$

Since $\mathbf{z}^{(1)} = [0.10, 0.25, 0.40]^\top$ is strictly positive, $\mathrm{ReLU}'(\mathbf{z}^{(1)}) = [1, 1, 1]^\top$, so

$$\boldsymbol{\delta}_1^{(1)} = \begin{bmatrix} 0.47236 \\ 0.53984 \\ 0.60732 \end{bmatrix}. \tag{6.50}$$

6.8.4 Gradients for layer 1

$$\frac{\partial \mathcal{L}^{(1)}}{\partial \mathbf{W}^{(1)}} = \boldsymbol{\delta}_1^{(1)} (\mathbf{x}^{(1)})^\top \tag{6.51}$$

$$= \begin{bmatrix} 0.47236 \\ 0.53984 \\ 0.60732 \end{bmatrix} \begin{bmatrix} 0.5 & 0.2 \end{bmatrix} \tag{6.52}$$

$$= \begin{bmatrix} 0.23618 & 0.09447 \\ 0.26992 & 0.10797 \\ 0.30366 & 0.12146 \end{bmatrix}, \tag{6.53}$$

and

$$\frac{\partial \mathcal{L}^{(1)}}{\partial \mathbf{b}^{(1)}} = \boldsymbol{\delta}_1^{(1)} = \begin{bmatrix} 0.47236 \\ 0.53984 \\ 0.60732 \end{bmatrix}. \tag{6.54}$$

6.9 Mini-batch Gradients (Averaging)

For a mini-batch of size $m = 4$, define per-sample gradients $g^{(i)}$. For example, layer 2 weights:

$$\frac{\partial \mathcal{L}_{\text{batch}}}{\partial \mathbf{W}^{(2)}} = \frac{1}{4} \sum_{i=1}^{4} \frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{W}^{(2)}}. \tag{6.55}$$

In the current draft, the averaged gradient is summarized as

$$\frac{\partial \mathcal{L}_{\text{batch}}}{\partial \mathbf{W}^{(2)}} \approx [0.0289,\ 0.0425,\ 0.0568]. \tag{6.56}$$

6.10 Parameter Updates (SGD)

Using learning rate $\eta = 0.1$, the gradient descent update is

$$\theta_{\text{new}} = \theta - \eta \nabla_\theta \mathcal{L}_{\text{batch}}. \tag{6.57}$$

6.10.1 Update layer 2

$$\begin{aligned} \mathbf{W}^{(2)}_{\text{new}} &= \mathbf{W}^{(2)} - 0.1 \frac{\partial \mathcal{L}_{\text{batch}}}{\partial \mathbf{W}^{(2)}} && (6.58)\\ &= [0.7,\, 0.8,\, 0.9] - 0.1[0.0289,\, 0.0425,\, 0.0568] && (6.59)\\ &= [0.6971,\, 0.7958,\, 0.8943]. && (6.60) \end{aligned}$$

The bias update is similarly

$$b^{(2)}_{\text{new}} = b^{(2)} - 0.1 \frac{\partial \mathcal{L}_{\text{batch}}}{\partial b^{(2)}}. \tag{6.61}$$

(In the current draft, a numerical value leading to $b^{(2)}_{\text{new}} \approx 0.0831$ is reported.)

6.10.2 Update layer 1

Likewise,

$$\mathbf{W}^{(1)}_{\text{new}} = \mathbf{W}^{(1)} - 0.1 \frac{\partial \mathcal{L}_{\text{batch}}}{\partial \mathbf{W}^{(1)}}, \qquad \mathbf{b}^{(1)}_{\text{new}} = \mathbf{b}^{(1)} - 0.1 \frac{\partial \mathcal{L}_{\text{batch}}}{\partial \mathbf{b}^{(1)}}. \tag{6.62}$$

The current draft summarizes the updated parameters numerically as

$$\mathbf{W}^{(1)}_{\text{new}} \approx \begin{bmatrix} 0.0933 & 0.1974 \\ 0.2937 & 0.3981 \\ 0.4931 & 0.5971 \end{bmatrix}, \qquad \mathbf{b}^{(1)}_{\text{new}} \approx \begin{bmatrix} -0.0032 \\ 0.0094 \\ 0.0142 \end{bmatrix}. \tag{6.63}$$

6.11 Second Iteration (Forward Pass Check)

To verify that the update decreases the loss, recompute the forward pass for sample 1 using updated parameters. The current draft reports (for sample 1):

$$\mathbf{z}^{(1)}_{\text{new}} = \begin{bmatrix} 0.0872 \\ 0.2388 \\ 0.3834 \end{bmatrix}, \qquad \mathbf{a}^{(1)}_{\text{new}} = \begin{bmatrix} 0.0872 \\ 0.2388 \\ 0.3834 \end{bmatrix}, \tag{6.64}$$

$$z^{(2)}_{\text{new}} \approx 0.6772, \qquad \hat{y}^{(1)}_{\text{new}} = \sigma(0.6772) \approx 0.6636, \tag{6.65}$$

so the new loss becomes

$$\mathcal{L}^{(1)}_{\text{new}} = -\log(1 - \hat{y}^{(1)}_{\text{new}}) = -\log(1 - 0.6636) = -\log(0.3364) \approx 1.0898. \tag{6.66}$$

Thus the loss decreases from ≈ 1.1223 to ≈ 1.0898, consistent with gradient descent improving the objective.

Chapter 7

Advanced Numerical Demonstrations

This chapter extends the concrete network in Chapter 6 and demonstrates, with explicit numbers, how common optimization and regularization techniques modify updates and activations.

7.1 Optimization Dynamics

7.1.1 Effect of the learning rate

Consider a parameter vector (or weight matrix flattened) θ and a gradient estimate $g = \nabla_\theta \mathcal{L}(\theta)$. A single gradient descent step is

$$\theta_{\text{new}} = \theta - \eta g, \tag{7.1}$$

where $\eta > 0$ is the learning rate.

Concrete example (Layer 2 weights). Using the Chapter 6 batch-gradient estimate

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} \approx [0.0289,\ 0.0425,\ 0.0568], \tag{7.2}$$

the update magnitude depends linearly on η.

- If $\eta = 0.1$:

$$\Delta \mathbf{W}^{(2)} = -0.1\,[0.0289, 0.0425, 0.0568] = [-0.00289, -0.00425, -0.00568]. \tag{7.1}$$

- If $\eta = 0.5$ (more aggressive):

$$\Delta \mathbf{W}^{(2)} = -0.5\,[0.0289, 0.0425, 0.0568] = [-0.01445, -0.02125, -0.02840]. \tag{7.2}$$

A larger η yields faster movement but increases the risk of overshooting minima or divergence.

7.1.2 Loss curve (illustrative)

Denote the batch loss after iteration t by $\mathcal{L}_t$. An example monotone decrease (as seen in the earlier draft) is:

Iteration t	$\mathcal{L}_t$
0	0.6160
1	0.5847
5	0.5172

(7.3)

7.2 Regularization Example: L2

7.2.1 Definition

L2 regularization (weight decay) augments the loss by a penalty on weight magnitudes:

$$\mathcal{L}_{\text{reg}}(\theta) = \mathcal{L}(\theta) + \lambda \sum_{\ell} \|\mathbf{W}^{(\ell)}\|_F^2, \tag{7.4}$$

where $\lambda > 0$ controls the penalty strength and

$$\|\mathbf{W}\|_F^2 = \sum_i \sum_j W_{ij}^2 \tag{7.5}$$

is the squared Frobenius norm.

7.2.2 Concrete computation

Using the Chapter 6 weights:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \\ 0.5 & 0.6 \end{bmatrix}, \qquad \mathbf{W}^{(2)} = [0.7, 0.8, 0.9]. \tag{7.6}$$

Compute squared Frobenius norms:

$$\|\mathbf{W}^{(1)}\|_F^2 = 0.1^2 + 0.2^2 + 0.3^2 + 0.4^2 + 0.5^2 + 0.6^2 = 0.01 + 0.04 + 0.09 + 0.16 + 0.25 + 0.36 = 0.91, \tag{7.4}$$

$$\|\mathbf{W}^{(2)}\|_F^2 = 0.7^2 + 0.8^2 + 0.9^2 = 0.49 + 0.64 + 0.81 = 1.94. \tag{7.5}$$

If $\lambda = 0.01$, the total penalty (weights only) becomes

$$\lambda \left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2 \right) = 0.01(0.91 + 1.94) = 0.01(2.85) = 0.0285. \tag{7.6}$$

Hence, if $\mathcal{L} = 0.616$ then

$$\mathcal{L}_{\text{reg}} = 0.616 + 0.0285 = 0.6445. \tag{7.7}$$

7.2.3 Gradient effect (weight decay view)

Differentiating,

$$\nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}_{\text{reg}} = \nabla_{\mathbf{W}^{(\ell)}} \mathcal{L} + 2\lambda \mathbf{W}^{(\ell)}. \tag{7.7}$$

Thus the update becomes

$$\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \eta \left(\nabla_{\mathbf{W}^{(\ell)}} \mathcal{L} + 2\lambda \mathbf{W}^{(\ell)} \right), \tag{7.8}$$

which explicitly pulls weights toward zero each step.

7.3 Momentum Optimization

7.3.1 Update rule

Momentum maintains a velocity vector $\mathbf{v}_t$:

$$\mathbf{v}_{t+1} = \beta \mathbf{v}_t + \nabla_\theta \mathcal{L}(\theta_t), \tag{7.9}$$

$$\theta_{t+1} = \theta_t - \eta\, \mathbf{v}_{t+1}, \tag{7.10}$$

with momentum coefficient $\beta \in (0, 1)$ (often 0.9).

7.3.2 Two-step numerical example (Layer 2 weights)

Let $\beta = 0.9$, $\eta = 0.1$, and initialize $\mathbf{v}_0 = \mathbf{0}$. Use the gradient $g_1 = [0.0289, 0.0425, 0.0568]$.

Iteration 1.

$$\mathbf{v}_1 = 0.9\mathbf{0} + g_1 = [0.0289, 0.0425, 0.0568], \tag{7.8}$$

$$\begin{aligned}\mathbf{W}_1^{(2)} &= \mathbf{W}_0^{(2)} - 0.1\,\mathbf{v}_1 = [0.7, 0.8, 0.9] - 0.1[0.0289, 0.0425, 0.0568] \\ &= [0.6971, 0.7958, 0.8943]. \end{aligned} \tag{7.10}$$

Iteration 2. Suppose the new gradient is $g_2 = [0.0215, 0.0318, 0.0425]$.

$$\begin{aligned}\mathbf{v}_2 &= 0.9\mathbf{v}_1 + g_2 = 0.9[0.0289, 0.0425, 0.0568] + [0.0215, 0.0318, 0.0425] \\ &= [0.0260, 0.0383, 0.0511] + [0.0215, 0.0318, 0.0425] = [0.0475, 0.0701, 0.0936], \end{aligned} \tag{7.12}$$

$$\begin{aligned}\mathbf{W}_2^{(2)} &= \mathbf{W}_1^{(2)} - 0.1\mathbf{v}_2 = [0.6971, 0.7958, 0.8943] - 0.1[0.0475, 0.0701, 0.0936] \\ &= [0.6919, 0.7890, 0.8758]. \end{aligned} \tag{7.13}$$

Momentum accumulates consistent gradient directions, often accelerating convergence.

7.4 Batch Normalization (Numerical Calculation)

7.4.1 Definition

Given pre-activations $z_j^{(i)}$ for neuron j over a mini-batch of size m_B, batch normalization computes:

$$\mu_{B,j} = \frac{1}{m_B} \sum_{i=1}^{m_B} z_j^{(i)}, \tag{7.11}$$

$$\sigma_{B,j}^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(z_j^{(i)} - \mu_{B,j} \right)^2, \tag{7.12}$$

$$\hat{z}_j^{(i)} = \frac{z_j^{(i)} - \mu_{B,j}}{\sqrt{\sigma_{B,j}^2 + \epsilon}}, \tag{7.13}$$

$$y_j^{(i)} = \gamma_j \hat{z}_j^{(i)} + \beta_j, \tag{7.14}$$

where γ_j, β_j are learnable parameters and $\epsilon > 0$ ensures numerical stability.

7.4.2 Concrete computation for one neuron

Take a hypothetical batch of four pre-activations for neuron $j = 1$:

$$z_1^{(1)} = 0.10, \quad z_1^{(2)} = 0.26, \quad z_1^{(3)} = 0.08, \quad z_1^{(4)} = 0.22. \tag{7.15}$$

Mean:

$$\mu_{B,1} = \frac{0.10 + 0.26 + 0.08 + 0.22}{4} = \frac{0.66}{4} = 0.165. \tag{7.16}$$

Variance:

$$\begin{aligned}\sigma_{B,1}^2 &= \frac{1}{4}\left[(0.10 - 0.165)^2 + (0.26 - 0.165)^2 + (0.08 - 0.165)^2 + (0.22 - 0.165)^2\right] \\ &= \frac{1}{4}\left[0.004225 + 0.009025 + 0.007225 + 0.003025\right] = \frac{0.02350}{4} = 0.005875. \end{aligned} \tag{7.14}$$

With $\epsilon = 10^{-5}$, normalize sample 1:

$$\hat{z}_1^{(1)} = \frac{0.10 - 0.165}{\sqrt{0.005875 + 10^{-5}}} = \frac{-0.065}{\sqrt{0.005885}} \approx \frac{-0.065}{0.0767} \approx -0.848. \tag{7.14'}$$

If $\gamma_1 = 1.0$ and $\beta_1 = 0.0$, then

$$y_1^{(1)} = \gamma_1 \hat{z}_1^{(1)} + \beta_1 = -0.848. \tag{7.15}$$

This reproduces the style of the batch-normalization calculation in the current draft.

7.5 Dropout Regularization (Numerical Calculation)

7.5.1 Training-time dropout

Let the hidden activation vector be $\mathbf{h} \in \mathbb{R}^n$. Dropout samples a mask $\mathbf{m} \in \{0, 1\}^n$ i.i.d. as

$$m_k \sim \text{Bernoulli}(1 - p), \tag{7.17}$$

and applies

$$\mathbf{h}_{\text{drop}} = \mathbf{h} \odot \mathbf{m}. \tag{7.18}$$

Concrete example (Layer 1 activation of sample 1). Using $\mathbf{h}^{(1)} = [0.10, 0.25, 0.40]^\top$ and dropout probability $p = 0.5$, suppose the sampled mask is

$$\mathbf{m} = [1, 0, 1]^\top. \tag{7.16}$$

Then

$$\mathbf{h}_{\text{drop}}^{(1)} = [0.10, 0.25, 0.40]^\top \odot [1, 0, 1]^\top = [0.10, 0, 0.40]^\top. \tag{7.17}$$

For $\mathbf{W}^{(2)} = [0.7, 0.8, 0.9]$ and $b^{(2)} = 0.1$, the new pre-activation becomes

$$z_{\text{drop}}^{(2)} = \mathbf{W}^{(2)}\mathbf{h}_{\text{drop}}^{(1)} + b^{(2)} = 0.7(0.10) + 0.8(0) + 0.9(0.40) + 0.1 = 0.07 + 0 + 0.36 + 0.1 = 0.53. \tag{7.18}$$

Dropout introduces stochasticity that discourages co-adaptation of features.

7.5.2 Test-time scaling

A common convention is to scale activations at inference by $(1-p)$ (if not using inverted dropout):

$$\mathbf{h}_{\text{test}} = (1-p)\mathbf{h}. \tag{7.19}$$

This makes the expected activation match between train and test.

7.6 Multi-sample Vectorized Processing

Vectorization replaces loops over samples with matrix operations. Stack a mini-batch of m inputs as a matrix

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \end{bmatrix} \in \mathbb{R}^{d \times m}. \tag{7.20}$$

For a layer with weights $\mathbf{W} \in \mathbb{R}^{n \times d}$ and bias $\mathbf{b} \in \mathbb{R}^n$, the pre-activations for the entire batch are

$$\mathbf{Z} = \mathbf{W}\mathbf{X} + \mathbf{b}\mathbf{1}^\top, \tag{7.19}$$

where $\mathbf{1} \in \mathbb{R}^m$ is the all-ones vector. Then apply activation element-wise:

$$\mathbf{A} = \sigma(\mathbf{Z}). \tag{7.21}$$

This single matrix multiplication computes all m samples in parallel and is the core reason GPUs accelerate neural network training.

Chapter 8

Optimization Techniques

We train a neural network by minimizing an empirical risk over parameters θ. Let the dataset be $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{N}$ and define

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell\big(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)}\big)\,. \tag{8.1}$$

Optimization algorithms produce a sequence $\{\theta_t\}_{t \geq 0}$ that (typically) reduces $\mathcal{L}(\theta_t)$.

8.1 Stochastic Gradient Descent (SGD)

8.1.1 Full-batch gradient descent

The basic gradient descent iteration is

$$\theta_{t+1} = \theta_t - \eta_t \nabla \mathcal{L}(\theta_t), \tag{8.2}$$

where $\eta_t > 0$ is the learning rate (possibly time-dependent).

8.1.2 Mini-batch SGD

In deep learning, $\nabla \mathcal{L}(\theta)$ is expensive when N is large. Let $\mathcal{B}_t \subset \{1, \ldots, N\}$ be a mini-batch of size B sampled at iteration t. Define the mini-batch objective

$$\mathcal{L}_{\mathcal{B}_t}(\theta) = \frac{1}{B} \sum_{i \in \mathcal{B}_t} \ell\big(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)}\big)\,, \tag{8.3}$$

and its gradient estimate

$$g_t := \nabla \mathcal{L}_{\mathcal{B}_t}(\theta_t). \tag{8.4}$$

Then SGD updates

$$\theta_{t+1} = \theta_t - \eta_t g_t. \tag{8.5}$$

Under uniform sampling (with standard independence assumptions),

$$\mathbb{E}[g_t \mid \theta_t] = \nabla \mathcal{L}(\theta_t), \tag{8.6}$$

so g_t is an unbiased estimator of the full gradient.

8.1.3 Learning-rate schedules (common choices)

A constant learning rate $\eta_t = \eta$ is often suboptimal. Typical schedules include:

- **Step decay:** $\eta_t = \eta_0 \gamma^{\lfloor t/T \rfloor}$ for some $\gamma \in (0, 1)$ and step period T.
- **Polynomial decay:** $\eta_t = \eta_0 (1 + t)^{-\alpha}$ for $\alpha \in (0, 1]$.
- **Cosine decay (common in practice):** $\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})\,(1 + \cos(\pi t/T))$.

8.1.4 A basic descent inequality (smooth case)

Assume $\mathcal{L}$ has L-Lipschitz gradient:

$$\|\nabla \mathcal{L}(\theta) - \nabla \mathcal{L}(\theta')\|_2 \leq L \|\theta - \theta'\|_2. \tag{8.7}$$

Then a standard inequality implies

$$\mathcal{L}(\theta_{t+1}) \leq \mathcal{L}(\theta_t) - \eta_t \langle \nabla \mathcal{L}(\theta_t), g_t \rangle + \frac{L \eta_t^2}{2} \|g_t\|_2^2. \tag{8.8}$$

In the deterministic case $g_t = \nabla \mathcal{L}(\theta_t)$ and small enough η_t, the loss decreases each step.

8.2 Momentum

Momentum accelerates SGD in directions of consistent descent by accumulating a "velocity".

8.2.1 Heavy-ball momentum

Let $g_t = \nabla \mathcal{L}_{\mathcal{B}_t}(\theta_t)$. Define velocity v_t via

$$v_{t+1} = \beta v_t + g_t, \tag{8.9}$$
$$\theta_{t+1} = \theta_t - \eta_t v_{t+1}, \tag{8.10}$$

where $\beta \in [0, 1)$ is the momentum coefficient (often 0.9).

Unrolling (8.9) (with $v_0 = 0$) yields

$$v_t = \sum_{k=0}^{t-1} \beta^{t-1-k} g_k, \tag{8.11}$$

so v_t is an exponentially weighted moving average of past gradients.

8.2.2 Nesterov accelerated gradient (NAG)

A popular variant evaluates the gradient at a look-ahead point:

$$v_{t+1} = \beta v_t + \nabla \mathcal{L}_{\mathcal{B}_t}(\theta_t - \eta_t \beta v_t)\,, \tag{8.12}$$
$$\theta_{t+1} = \theta_t - \eta_t v_{t+1}. \tag{8.13}$$

This can reduce oscillations in narrow valleys compared to (8.10).

8.3 Adaptive Methods (RMSProp, Adam, AdamW)

Adaptive optimizers rescale updates coordinate-wise using second-moment statistics of gradients.

8.3.1 RMSProp (core idea)

Maintain an exponential moving average of squared gradients:

$$v_{t+1} = \beta_2 v_t + (1-\beta_2)(g_t \odot g_t), \tag{8.14}$$

and update

$$\theta_{t+1} = \theta_t - \eta_t \frac{g_t}{\sqrt{v_{t+1}} + \varepsilon}, \tag{8.15}$$

where all operations are element-wise.

8.3.2 Adam (Adaptive Moment Estimation)

Adam maintains first and second moment estimates

$$m_{t+1} = \beta_1 m_t + (1-\beta_1) g_t, \tag{8.16}$$
$$v_{t+1} = \beta_2 v_t + (1-\beta_2)(g_t \odot g_t), \tag{8.17}$$

with bias corrections

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1-\beta_1^{t+1}}, \tag{8.18}$$
$$\hat{v}_{t+1} = \frac{v_{t+1}}{1-\beta_2^{t+1}}. \tag{8.19}$$

The Adam update is

$$\theta_{t+1} = \theta_t - \eta_t \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1}} + \varepsilon}. \tag{8.20}$$

Typical defaults are $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$.

8.3.3 AdamW (decoupled weight decay)

With L2 regularization, one often writes $\nabla \mathcal{L}(\theta) + \lambda\theta$. However, for adaptive methods the effect of adding $\lambda\theta$ inside the gradient can differ from "decoupled" weight decay.

AdamW applies weight decay directly to parameters:

$$\theta_{t+1} = (1-\eta_t\lambda)\theta_t - \eta_t \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1}} + \varepsilon}. \tag{8.21}$$

This cleanly separates the shrinkage term from the adaptive gradient step.

8.4 Regularization as Optimization

Regularization modifies the training objective to encourage desirable parameter structure (small norm, sparsity, robustness).

8.4.1 L2 regularization (weight decay) and MAP interpretation

Add an L2 penalty on weights:

$$\mathcal{L}_{\text{reg}}(\theta) = \mathcal{L}(\theta) + \frac{\lambda}{2} \sum_{\ell} \|\mathbf{W}^{(\ell)}\|_F^2. \tag{8.22}$$

Then

$$\nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}_{\text{reg}} = \nabla_{\mathbf{W}^{(\ell)}} \mathcal{L} + \lambda \mathbf{W}^{(\ell)}. \tag{8.23}$$

With SGD, the update becomes

$$\mathbf{W}^{(\ell)} \leftarrow (1 - \eta_t \lambda)\mathbf{W}^{(\ell)} - \eta_t \nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}. \tag{8.24}$$

Thus weights shrink at each step by a factor $(1 - \eta_t \lambda)$.

Probabilistic view (sketch): minimizing (8.22) corresponds to MAP estimation under an (independent) Gaussian prior on weights, since $-\log p(\mathbf{W})$ is proportional to $\|\mathbf{W}\|_F^2$.

8.4.2 L1 regularization and sparsity

L1-regularized objective:

$$\mathcal{L}_{\text{L1}}(\theta) = \mathcal{L}(\theta) + \lambda \sum_{\ell} \|\mathbf{W}^{(\ell)}\|_1 = \mathcal{L}(\theta) + \lambda \sum_{\ell} \sum_{i,j} |W_{ij}^{(\ell)}|. \tag{8.25}$$

The subgradient satisfies

$$\frac{\partial}{\partial W_{ij}^{(\ell)}} |W_{ij}^{(\ell)}| = \begin{cases} \operatorname{sign}(W_{ij}^{(\ell)}) & W_{ij}^{(\ell)} \neq 0, \\ s \in [-1, 1] & W_{ij}^{(\ell)} = 0. \end{cases} \tag{8.26}$$

L1 tends to produce sparse solutions (many parameters exactly zero).

8.4.3 Dropout (inverted dropout)

Let $\mathbf{a}^{(\ell)}$ be activations at layer ℓ. Sample a mask $\mathbf{m}^{(\ell)} \in \{0, 1\}^{n_\ell}$ i.i.d. with

$$m_j^{(\ell)} \sim \text{Bernoulli}(q), \qquad q = 1 - p. \tag{8.27}$$

In inverted dropout, training-time activations are

$$\tilde{\mathbf{a}}^{(\ell)} = \frac{1}{q}\big(\mathbf{m}^{(\ell)} \odot \mathbf{a}^{(\ell)}\big). \tag{8.28}$$

Then

$$\mathbb{E}[\tilde{\mathbf{a}}^{(\ell)} \mid \mathbf{a}^{(\ell)}] = \mathbf{a}^{(\ell)}, \tag{8.29}$$

so no additional scaling is needed at inference time (contrast with the non-inverted convention).

8.4.4 Batch normalization (BN)

Given pre-activations $z_j^{(\ell),(i)}$ for neuron j over a mini-batch $i = 1, \dots, B$, BN computes

$$\mu_{B,j} = \frac{1}{B} \sum_{i=1}^{B} z_j^{(\ell),(i)}, \tag{8.30}$$

$$\sigma_{B,j}^2 = \frac{1}{B} \sum_{i=1}^{B} \left(z_j^{(\ell),(i)} - \mu_{B,j} \right)^2, \tag{8.31}$$

$$\hat{z}_j^{(\ell),(i)} = \frac{z_j^{(\ell),(i)} - \mu_{B,j}}{\sqrt{\sigma_{B,j}^2 + \varepsilon}}, \tag{8.32}$$

$$y_j^{(\ell),(i)} = \gamma_j \hat{z}_j^{(\ell),(i)} + \beta_j. \tag{8.33}$$

Here γ_j, β_j are learnable parameters.

Training vs inference. During training, BN uses mini-batch statistics $\mu_{B,j}, \sigma_{B,j}^2$. During inference, implementations typically use running averages (estimated during training) to avoid dependence on the test-time batch composition.

Why it helps. BN stabilizes the scale of intermediate representations, often enabling larger learning rates and improving gradient flow in deep networks, while also injecting mild stochasticity due to batch statistics.

8.5 Stability Tricks (practical)

8.5.1 Gradient clipping

To prevent exploding gradients, clip by global norm:

$$g_t \leftarrow g_t \cdot \min\left(1, \frac{\tau}{\|g_t\|_2}\right), \tag{8.34}$$

for threshold $\tau > 0$.

8.5.2 Mini-batch size trade-off

Small batches increase gradient noise (sometimes improving exploration and generalization), while large batches reduce variance but can require careful learning-rate scaling and warmup.

Chapter 9

Analysis and Theory

This chapter expands the theoretical part of the text. The goal is not to provide fully detailed proofs, but to state results precisely, clarify assumptions, and give *proof sketches* and intuition.

9.1 Function Approximation

9.1.1 Setting and notation

Let $K \subset \mathbb{R}^d$ be a compact set (e.g., $K = [0,1]^d$). Let $\sigma : \mathbb{R} \to \mathbb{R}$ be an activation function. A one-hidden-layer (two-layer) network with width N is

$$f_N(x) = \sum_{j=1}^{N} a_j \, \sigma(\mathbf{w}_j^\top x + b_j), \qquad x \in \mathbb{R}^d, \tag{9.1}$$

where $(a_j, \mathbf{w}_j, b_j) \in \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}$ are parameters.

9.1.2 Universal Approximation Theorem (UAT)

Theorem (Universal approximation; informal). If σ is a non-polynomial activation (e.g., sigmoid, ReLU, tanh), then for any continuous $f \in C(K)$ and any $\varepsilon > 0$, there exists N and parameters such that

$$\sup_{x \in K} |f_N(x) - f(x)| < \varepsilon. \tag{9.2}$$

This asserts *existence* of an approximating network but does not give an efficient method to find it.

Proof sketch (high level). One common route uses functional analysis:

- Consider the set $\mathcal{F} = \{f_N : N \in \mathbb{N}\}$ and its closure in $(C(K), \|\cdot\|_\infty)$.
- Show that if σ is non-polynomial, then $\mathcal{F}$ is dense in $C(K)$ (often via a Hahn–Banach / Riesz representation argument: any continuous linear functional separating $\mathcal{F}$ from $C(K)$ would correspond to a signed measure μ, and one shows $\int \sigma(\mathbf{w}^\top x + b)\, d\mu(x) = 0$ for all $(\mathbf{w}, b)$ forces $\mu = 0$).
- Density implies any continuous f can be approximated uniformly on K.

Different proofs exist (e.g., Stone–Weierstrass style arguments for specific activations).

9.1.3 Approximation rates (why UAT is not enough)

UAT does not tell how large N must be as a function of ε. A useful theoretical question is: for a function class $\mathcal{G}$ (e.g., Lipschitz or Sobolev functions), what is the best achievable error

$$\inf_{f_N \in \mathcal{F}_N} \|f_N - f\|? \tag{9.3}$$

where $\mathcal{F}_N$ is the class of width-N networks of form (9.1).

Very roughly:

- **Smooth functions** can often be approximated faster as N increases.
- **High dimension** (d large) typically leads to slow worst-case rates ("curse of dimensionality") unless f has exploitable structure (sparsity, compositional form, low effective dimension).

This motivates studying *structured* targets and *deep* architectures.

9.1.4 Why depth helps (compositional structure)

A depth-L network can be viewed as a compositional function class:

$$f(x) = f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(1)}(x), \tag{9.4}$$

where each $f^{(\ell)}$ is an affine map plus nonlinearity.

If the target function itself has a compositional structure (e.g., it is naturally written as nested low-dimensional functions), a deep network can represent/approximate it using far fewer parameters than a shallow one.

9.2 Depth vs. Width

9.2.1 Expressivity measures

Two common notions:

- **Representation power:** Can the network represent a given function exactly?
- **Approximation power:** Can it approximate within error ε?

Depth increases expressivity because repeated composition creates many "regions" of different affine behavior (especially for piecewise-linear activations like ReLU).

9.2.2 Piecewise linear regions (ReLU intuition)

A ReLU network is piecewise linear. The input space is partitioned into regions within which the network is an affine function. Depth can increase the number of such regions dramatically, often exponentially in depth under suitable conditions.

Proof sketch (intuition). Each ReLU introduces a hyperplane boundary where a unit switches between active/inactive. Composing layers leads to new boundaries that are mapped and "folded" by previous layers, creating many linear regions. This creates a combinatorial growth in region count with depth.

9.2.3 Separation results (functions needing depth)

Theorem (informal separation). There exist families of functions that can be represented by a deep network with polynomial size (parameters/units) but require exponential width if restricted to shallow networks.

Example intuition: parity / compositional Boolean functions. A parity-like function has a strong hierarchical structure: it can be computed by composing XORs on pairs, which naturally forms a tree of depth $\log n$. Shallow representations must "memorize" many input patterns, leading to exponential size in the worst case.

9.3 Optimization Landscapes

9.3.1 Nonconvexity and critical points

Training a neural network typically solves

$$\min_{\theta} \ \mathcal{L}(\theta), \tag{9.5}$$

where $\mathcal{L}$ is nonconvex in θ due to composition and nonlinearities.

A critical point satisfies

$$\nabla \mathcal{L}(\theta) = 0. \tag{9.6}$$

Second-order behavior is characterized by the Hessian

$$H(\theta) = \nabla^2 \mathcal{L}(\theta). \tag{9.7}$$

A point can be a local minimum ($H \succeq 0$), local maximum ($H \preceq 0$), or saddle (indefinite Hessian).

9.3.2 Saddle points in high dimension (intuition)

In high-dimensional parameter spaces, saddle points are more common than strict local maxima. SGD noise and mini-batching introduce stochasticity that can help escape saddle regions, which partially explains why first-order methods work well in practice.

9.3.3 Overparameterization and benign landscapes (idea)

Modern networks are often overparameterized (more parameters than samples). Empirically, this regime often allows reaching near-zero training error. A common theoretical theme is that, with sufficient width, gradient-based methods behave almost like convex optimization in a neighborhood of initialization (related to linearization/NTK-type arguments).

Proof sketch (idea only). Linearize the network around initialization θ_0:

$$f_\theta(x) \approx f_{\theta_0}(x) + \nabla_\theta f_{\theta_0}(x)^\top (\theta - \theta_0). \tag{9.8}$$

If this linearization remains accurate during training and the induced kernel matrix is well-conditioned, then gradient descent can be analyzed similarly to kernel regression.

9.4 Generalization

9.4.1 Train vs. test

Let P be the (unknown) data distribution on (X, Y). Define the population risk

$$R(\theta) = \mathbb{E}_{(X,Y)\sim P}\big[\ell(f_\theta(X), Y)\big], \tag{9.9}$$

and empirical risk (training loss)

$$\hat{R}_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(f_\theta(x_i), y_i). \tag{9.10}$$

Generalization asks how close $\hat{R}_n(\theta)$ is to $R(\theta)$, especially for $\hat{\theta}$ produced by training.

9.4.2 Bias–variance decomposition (squared loss)

For squared loss in regression with a learning algorithm producing a predictor $\hat{f}$ from data $\mathcal{D}$, one can decompose the expected test error at a point x:

$$\mathbb{E}_{\mathcal{D}}\big[(\hat{f}(x) - f^\star(x))^2\big] = \underbrace{\big(\mathbb{E}_{\mathcal{D}}[\hat{f}(x)] - f^\star(x)\big)^2}_{\text{Bias}^2} + \underbrace{\mathbb{E}_{\mathcal{D}}\big[(\hat{f}(x) - \mathbb{E}_{\mathcal{D}}[\hat{f}(x)])^2\big]}_{\text{Variance}}. \tag{9.11}$$

(An additional irreducible noise term appears when $Y = f^\star(X) + \varepsilon$ with noise.)

Interpretation. Increasing model complexity often reduces bias but increases variance, motivating regularization and early stopping.

9.4.3 Capacity control and uniform convergence (sketch)

A typical form of learning-theory result bounds the gap between population and empirical risks over a hypothesis class $\mathcal{H}$:

$$\sup_{h\in\mathcal{H}} |R(h) - \hat{R}_n(h)|. \tag{9.12}$$

This can be controlled by complexity measures such as VC dimension or Rademacher complexity (especially for bounded loss classes).

Proof sketch (outline). One uses symmetrization:

$$\mathbb{E}\left[\sup_{h\in\mathcal{H}}(R(h) - \hat{R}_n(h))\right] \le 2\,\mathbb{E}\left[\sup_{h\in\mathcal{H}} \frac{1}{n}\sum_{i=1}^{n} \sigma_i h(x_i)\right], \tag{9.13}$$

where $\sigma_i \in \{-1, +1\}$ are Rademacher variables, then bounds the right-hand side using contraction inequalities and norm constraints.

9.4.4 Implicit regularization (phenomenon)

Even without explicit penalties, gradient-based training can prefer certain solutions among many interpolating ones. For example, in linear regression, gradient descent initialized at zero converges to the minimum ℓ_2-norm solution among those that fit the data. Deep networks exhibit more complex forms of implicit bias, but the theme remains: optimization dynamics can act as a regularizer.

9.5 Interpolation and Double Descent

9.5.1 Classical U-shaped curve

In classical statistics, test error often decreases with model complexity (bias reduction) and then increases (variance increase), yielding a U-shaped curve.

9.5.2 Interpolation threshold

When a model becomes expressive enough to fit the training set perfectly, one reaches the interpolation regime:

$$\hat{R}_n(\hat{\theta}) \approx 0. \tag{9.14}$$

Classically, perfect fit suggests overfitting, but modern deep learning often operates in this regime.

9.5.3 Double descent (empirical phenomenon)

In many modern settings, the test error can decrease again as parameters increase further, producing a "double descent" curve:

- Underparameterized: decreasing test error.
- Near interpolation threshold: peak test error.
- Overparameterized: decreasing test error again.

This is an active research area and not fully explained by classical bias–variance alone.

Sketch of one explanation route. In linear models, one can analyze the minimum-norm interpolating solution explicitly and show that increasing parameters changes the geometry of interpolation and the effective complexity (e.g., via eigenvalues of the data covariance), leading to non-monotone risk. Deep networks are more complex, but similar geometric/effective-dimension ideas appear.

9.6 What theory does *not* yet explain

Even with the above tools, many practical behaviors remain only partially understood:

- Why certain architectures (e.g., residual connections, attention) train reliably at scale.
- Why SGD with specific hyperparameters generalizes well despite extreme overparameterization.
- Predicting performance from data/model/compute scaling in a principled way.

Thus, the theory is best viewed as a set of lenses: approximation, optimization, and generalization, each explaining part of the empirical success.

Chapter 10

Computational Graphs and Automatic Differentiation

The backpropagation algorithm presented in Chapter 5 is written for a specific network structure (layer-by-layer composition). This chapter generalizes it to arbitrary computational graphs, which is essential for modern frameworks (PyTorch, TensorFlow) and complex architectures (RNNs, Transformers, dynamic graphs).

10.1 Computational Graphs: Formal Definition

10.1.1 Directed acyclic graphs (DAGs)

A **computational graph** is a directed acyclic graph (DAG) where:

- Each **node** v represents a variable or operation.
- Each **edge** (u, v) represents data flow: output of u is input to v.
- **Leaf nodes** (sources) hold input data $\mathbf{x}$ or parameters θ.
- **Root nodes** (sinks) represent the loss or output $\mathcal{L}$.

An acyclic structure ensures that a topological ordering exists: we can label nodes $v_1, \ldots, v_n$ such that if (v_i, v_j) is an edge, then $i < j$.

10.1.2 Example: Simple expression graph

Consider computing $y = (x_1 + x_2) \cdot x_1$ where inputs are $x_1, x_2 \in \mathbb{R}$.

Nodes:

- $v_1 = x_1$ (leaf)
- $v_2 = x_2$ (leaf)
- $v_3 = v_1 + v_2$ (addition)
- $v_4 = v_3 \cdot v_1$ (multiplication)
- $v_5 = y = v_4$ (output/root)

Edges: (v_1, v_3), (v_2, v_3), (v_3, v_4), (v_1, v_4), (v_4, v_5).

The topological order is v_1, v_2, v_3, v_4, v_5. Forward evaluation follows this order; backward propagation (differentiation) reverses it.

10.1.3 Forward evaluation

For each node v_j, let $\text{in}(v_j)$ denote the set of immediate predecessors (parents). If v_j is an operation with inputs from parents, compute

$$v_j = \text{op}_j(\{v_i : i \in \text{parents}(j)\}). \tag{10.1}$$

By topological ordering, all parents of v_j have already been computed.

10.2 Automatic Differentiation: Backpropagation in DAGs

10.2.1 Generalized chain rule

For any node v_j in the graph, the gradient w.r.t. parameter (or leaf) v_i is decomposed as a sum over all paths from v_i to the root (loss $\mathcal{L}$):

$$\frac{\partial \mathcal{L}}{\partial v_i} = \sum_{\text{paths } i \to \text{root}} \prod_{\text{edges in path}} \frac{\partial v_{\text{dest}}}{\partial v_{\text{src}}}. \tag{10.2}$$

Equivalently, using dynamic programming on the DAG: define $\bar{v}_j = \frac{\partial \mathcal{L}}{\partial v_j}$ (the adjoint or backprop error). For the root, $\bar{v}_{\text{root}} = 1$ (or $\nabla \mathcal{L}$ if loss is vectorial).

For each non-root node v_j, the adjoint is computed as

$$\bar{v}_j = \sum_{k \in \text{children}(j)} \bar{v}_k \cdot \frac{\partial v_k}{\partial v_j}. \tag{10.3}$$

This recurrence is applied in reverse topological order (from root to leaves).

10.2.2 Example: Computing adjoints for $y = (x_1 + x_2) \cdot x_1$

Continue the graph from Section 10.1.

Forward pass (already computed): Suppose $x_1 = 2, x_2 = 3$. Then $v_3 = 2 + 3 = 5$, $v_4 = 5 \cdot 2 = 10$, $y = 10$.

Backward pass (adjoints): Assume loss is $\mathcal{L} = y^2$, so $\frac{\partial \mathcal{L}}{\partial y} = 2y = 20$.

1. Initialize $\bar{v}_5 = 20$ (root adjoint).
2. $v_4 \to v_5$: edge with $\frac{\partial v_5}{\partial v_4} = 1$, so

$$\bar{v}_4 = \bar{v}_5 \cdot 1 = 20. \tag{10.4}$$

3. $v_3 \to v_4$ and $v_1 \to v_4$: edges with $\frac{\partial v_4}{\partial v_3} = v_1 = 2$ and $\frac{\partial v_4}{\partial v_1} = v_3 = 5$, so

$$\bar{v}_3 = \bar{v}_4 \cdot 2 = 40, \quad \bar{v}_1 \mathrel{+}= \bar{v}_4 \cdot 5 = 100. \tag{10.5}$$

(Note: $\bar{v}_1$ accumulates because it has multiple children.)

4. $v_1 \to v_3$ and $v_2 \to v_3$: edges with $\frac{\partial v_3}{\partial v_1} = 1$ and $\frac{\partial v_3}{\partial v_2} = 1$, so

$$\bar{v}_1 \mathrel{+}= \bar{v}_3 \cdot 1 = 40 \quad \Rightarrow \quad \bar{v}_1 = 100 + 40 = 140, \tag{10.6}$$

$$\bar{v}_2 = \bar{v}_3 \cdot 1 = 40. \tag{10.7}$$

Result: $\frac{\partial \mathcal{L}}{\partial x_1} = 140$, $\frac{\partial \mathcal{L}}{\partial x_2} = 40$.

This can be verified by hand: $\frac{\partial \mathcal{L}}{\partial x_1} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial x_1} = 20 \cdot (2x_1 + x_2) = 20(4 + 3) = 140$.

10.3 Forward Mode vs. Reverse Mode Differentiation

10.3.1 Reverse mode (backpropagation)

As described above, reverse mode (backprop) computes all adjoints via a single backward pass.

Complexity: Each edge is traversed once, and each adjoint accumulation is $O(1)$. Total cost: $O(|V| + |E|)$ where $|V|$ is node count and $|E|$ is edge count. For a feedforward network with L layers of n neurons each, this is roughly $O(L \cdot n)$.

Memory: Must store intermediate activations v_j for all nodes (for use in gradients during backward pass). For deep networks, this can be prohibitive, motivating strategies like gradient checkpointing.

10.3.2 Forward mode (tangent linear)

Forward mode traces gradients *forward* through the graph. For each leaf input x_i, compute the tangent vector $\dot{v}_j = \frac{\partial v_j}{\partial x_i}$ via a forward pass.

Recurrence: Initialize $\dot{x}_i = 1$, $\dot{x}_{i'} = 0$ for $i' \neq i$. For each node in topological order,

$$\dot{v}_j = \sum_{k \in \text{parents}(j)} \frac{\partial v_j}{\partial v_k} \dot{v}_k. \tag{10.8}$$

Complexity: One forward tangent pass computes $\frac{\partial v_j}{\partial x_i}$ for all j and *one* input i. To get all d inputs: requires d forward passes, each costing $O(|V|+|E|)$. Total: $O(d \cdot (|V|+|E|))$.

For $d \gg 1$ outputs and $\ll d$ inputs (as in supervised learning), reverse mode is vastly more efficient.

10.3.3 Comparison table

	Reverse (Backprop)	**Forward (Tangent)**
One gradient pass cost	$O(\|V\| + \|E\|)$	$O(\|V\| + \|E\|)$
For m outputs, n inputs	$O(m \cdot (\|V\| + \|E\|))$	$O(n \cdot (\|V\| + \|E\|))$
Best for	$n \gg m$ (typical learning)	$m \gg n$ (rare in learning)
Memory	$O(n_{\max}^{(\ell)})$ during backward	$O(n_{\max}^{(\ell)})$ during forward

In neural network training, $m = 1$ (scalar loss) and n is number of parameters (millions to billions), so reverse mode is standard.

10.4 Chain Rule in Multivariate Form

For nodes with vector/matrix values, the chain rule uses careful indexing.

10.4.1 Jacobian-vector products

If $v_k : \mathbb{R}^a \to \mathbb{R}^b$ (a node taking a-dimensional input, producing b-dimensional output), and loss is scalar, then

$$\frac{\partial \mathcal{L}}{\partial v_{k,i}} = \sum_{j=1}^{b} \frac{\partial \mathcal{L}}{\partial v_{k,j}^{\text{out}}} \cdot \frac{\partial v_{k,j}^{\text{out}}}{\partial v_{k,i}}. \tag{10.9}$$

In matrix notation, if $v_k^{\text{in}} \in \mathbb{R}^a$, $v_k^{\text{out}} \in \mathbb{R}^b$, and $J_k \in \mathbb{R}^{b \times a}$ is the Jacobian, then

$$\overline{v_k^{\text{in}}} = J_k^\top \, \overline{v_k^{\text{out}}}. \tag{10.10}$$

10.4.2 Example: Softmax backward

Softmax node: input $z \in \mathbb{R}^K$, output $p \in \mathbb{R}^K$ with $p_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$.

The Jacobian (from Chapter 4) is

$$J_{ij} = \frac{\partial p_i}{\partial z_j} = p_i(\delta_{ij} - p_j). \tag{10.11}$$

If the upstream loss adjoint is $\bar{p} \in \mathbb{R}^K$, then

$$\bar{z} = J^\top \bar{p} = \begin{bmatrix} p_1(\bar{p}_1 - \bar{p}^\top p) \\ \vdots \\ p_K(\bar{p}_K - \bar{p}^\top p) \end{bmatrix}. \tag{10.12}$$

In the special case of cross-entropy loss where $\bar{p}_i = p_i - y_i$ (from Chapter 5), we get $\bar{z}_i = p_i - y_i$, matching our earlier result.

Chapter 11

Numerical Stability and Precision

Neural networks require careful numerical handling, especially in loss computation and gradient flow.

11.1 Softmax and the Log-Sum-Exp Trick

11.1.1 Naive softmax (numerically unstable)

Computing $\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$ directly can overflow: if z_i is large (e.g., $z_i = 1000$), then e^{z_i} exceeds floating-point range.

11.1.2 Stable variant (log-sum-exp)

Subtract the max before exponentiating:

$$z_i' = z_i - \max_k z_k, \tag{11.1}$$

then compute

$$p_i = \frac{e^{z_i'}}{\sum_j e^{z_j'}}. \tag{11.2}$$

Now $z_i' \leq 0$ for all i, so $e^{z_i'} \in (0, 1]$, avoiding overflow.

Mathematically:

$$\frac{e^{z_i}}{\sum_j e^{z_j}} = \frac{e^{z_i - \max z}}{\sum_j e^{z_j - \max z}}. \tag{11.3}$$

11.1.3 Log-domain computation

For numerical stability in probability computations, work in log space:

$$\log p_i = z_i' - \log\left(\sum_j e^{z_j'}\right) = z_i' - \text{logsumexp}(z'). \tag{11.4}$$

This is especially useful when computing cross-entropy:

$$\text{CCE} = -\sum_k y_k \log p_k = -\sum_k y_k (z_k' - \text{logsumexp}(z')). \tag{11.5}$$

11.2 Underflow and Overflow in Deep Networks

11.2.1 Activation norms

In very deep networks, activations can grow or shrink exponentially layer by layer. If $|z^{(\ell)}| \to 0$ (underflow), gradients vanish. If $|z^{(\ell)}| \to \infty$ (overflow), parameters become NaN.

11.2.2 Initialization and gradient norms

Careful initialization (e.g., He initialization for ReLU, Xavier for sigmoid) helps maintain reasonable activation magnitudes:

$$\mathbf{W}_{ij}^{(\ell)} \sim \mathcal{N}\left(0, \frac{2}{n_{\ell-1}}\right) \quad \text{(He)} \tag{11.6}$$

ensures that $\mathbb{E}[|z^{(\ell)}|^2] \approx \mathbb{E}[|z^{(\ell-1)}|^2]$.

11.2.3 Gradient clipping

To prevent exploding gradients, clip by norm:

$$g \leftarrow g \cdot \min\left(1, \frac{C}{\|g\|_2}\right), \tag{11.7}$$

where C is a threshold (e.g., $C = 1$). This keeps gradients bounded during backprop, especially important for RNNs.

11.3 Mixed Precision Training

Modern hardware (GPUs, TPUs) supports low-precision floating point (e.g., FP16: 16-bit) at much higher speed than full precision (FP32: 32-bit).

11.3.1 Strategy

1. Perform forward pass in FP16 (fast).
2. Compute loss in FP16 (or reduced precision).
3. *Scale* the loss by a large factor L_{scale} (e.g., 2^{15}):
$$\hat{\mathcal{L}} = L_{\text{scale}} \cdot \mathcal{L}. \tag{11.8}$$
4. Backprop through scaled loss (gradients are also scaled up, reducing underflow risk).
5. Perform weight update in FP32, with gradients scaled down by L_{scale}.

11.3.2 Why scaling helps

FP16 has range roughly $[6 \times 10^{-5}, 6 \times 10^{4}]$. Typical gradients are small ($\sim 10^{-3}$ to 10^{-5}); without scaling, they underflow to zero in FP16. Scaling before backprop keeps gradients in the representable range; then downscaling recovers the true gradient for the update.

Chapter 12

Tensor Operations and Notation

Modern neural networks, especially CNNs and Transformers, manipulate high-dimensional arrays (tensors). This chapter formalizes tensor operations and notation.

12.1 Tensors and Index Notation

12.1.1 Definition

An n-th order tensor $T \in \mathbb{R}^{d_1 \times \cdots \times d_n}$ is a multi-dimensional array.

- Order 0: scalar.
- Order 1: vector.
- Order 2: matrix.
- Order 3+: higher-order tensors.

Element indexing: $T[i_1, i_2, \ldots, i_n]$ or $T_{i_1 i_2 \cdots i_n}$.

12.1.2 Einstein notation (summation convention)

In Einstein notation, repeated indices imply summation:

$$C_{ij} = \sum_k A_{ik} B_{kj} \quad \text{is written as} \quad C_{ij} = A_{ik} B_{kj}. \tag{12.1}$$

Implicit indices (not repeated) are free indices; repeated indices are contracted (summed over).

Example: Matrix-vector product

$$y_i = \sum_j W_{ij} x_j \quad \Rightarrow \quad y_i = W_{ij} x_j. \tag{12.2}$$

Example: Convolution (1D, single sample)

Input sequence x_t (length T), filter w_s (length S), stride 1:

$$y_t = \sum_{s=0}^{S-1} w_s x_{t+s} \quad \Rightarrow \quad y_t = w_s x_{t+s}. \tag{12.3}$$

Here t is the free (output) index, s is contracted.

Example: Batched matrix multiplication

Batch size B, $A \in \mathbb{R}^{B\times M\times K}$, $B \in \mathbb{R}^{B\times K\times N}$:

$$C_{b,m,n} = \sum_k A_{b,m,k} B_{b,k,n} \quad \Rightarrow \quad C_{bmn} = A_{bmk} B_{bkn}. \tag{12.4}$$

12.2 Broadcasting and Element-wise Operations

12.2.1 Broadcasting rules (NumPy/PyTorch convention)

When operating on tensors of different shapes, dimensions are aligned from the right. Missing dimensions are inserted on the left.

Example 1: Shape (M, N) and shape $(N,)$: expand $(N,)$ to $(1, N)$, then broadcast to (M, N).

Example 2: Shape (B, M, N) and shape $(N,)$: expand to $(1, 1, N)$, broadcast to (B, M, N).

Element-wise scaling:

$$Y_{b,m,n} = X_{b,m,n} \cdot \gamma_n \tag{12.5}$$

where γ is shape $(N,)$. This is a common pattern in layer normalization and bias addition.

12.3 Reshape and Transpose

12.3.1 Reshape (view)

Changing shape without reordering data:

$$X \in \mathbb{R}^{B\times C\times H\times W} \to X' \in \mathbb{R}^{BC\times HW}. \tag{12.6}$$

Data layout matters: reshape assumes row-major (C-contiguous) or column-major (Fortran-contiguous) memory order.

12.3.2 Transpose (permutation)

Reorder dimensions:

$$Y_{i,j,k,\ell} = X_{k,i,\ell,j} \quad \text{corresponds to} \quad \text{permute}((0,1,2,3) \to (2,0,3,1)). \tag{12.7}$$

In Einstein notation:

$$Y_{ijkl} = X_{kilj}. \tag{12.8}$$

12.3.3 Flattening (vectorization)

Combining batch and spatial dimensions:

$$X \in \mathbb{R}^{B\times C\times H\times W} \to \vec{X} \in \mathbb{R}^{B\cdot C\cdot H\cdot W}. \tag{12.9}$$

Useful for fully connected layers following convolutional layers.

Chapter 13

Hyperparameter Tuning and Learning Rate Schedules

Training hyperparameters (learning rate, momentum, batch size, etc.) dramatically affect convergence and generalization. This chapter covers principled tuning strategies.

13.1 Learning Rate Selection

13.1.1 Learning rate finder (LRFinder)

A practical heuristic (Fastai, PyTorch Lightning):

1. Start with a small learning rate $\eta_{\min}$ (e.g., 10^{-5}).
2. Train for one epoch, exponentially increasing η at each batch:

$$\eta_t = \eta_{\min} \cdot \left(\frac{\eta_{\max}}{\eta_{\min}} \right)^{t/T}, \tag{13.1}$$

   where T is total batches, $\eta_{\max}$ is max rate.
3. Track loss vs. η.
4. Select η where loss is still decreasing steeply but not yet diverging.

Why it works: Identifies the "sweet spot" where the loss landscape has largest gradient (learning is efficient) without being at risk of divergence.

13.1.2 Learning rate schedules

After choosing a base learning rate, reduce it over time:

Step decay:

$$\eta_t = \eta_0 \cdot \gamma^{\lfloor t/S \rfloor}, \tag{13.2}$$

where $\gamma < 1$ (e.g., 0.1) and S is step size (epochs between drops).

Exponential decay:

$$\eta_t = \eta_0 \cdot e^{-\lambda t}, \tag{13.3}$$

where $\lambda > 0$ is decay rate.

Cosine annealing:

$$\eta_t = \eta_{\min} + \frac{\eta_0 - \eta_{\min}}{2}\left(1 + \cos\frac{\pi t}{T}\right), \tag{13.4}$$

where T is total iterations. Smoothly decreases from η_0 to $\eta_{\min}$.

Cosine with warm restarts (SGDR): Reset the cosine schedule multiple times, with decreasing max learning rate:

$$\eta_t^{(i)} = \eta_{\min} + \frac{\eta_0 \cdot \gamma^i - \eta_{\min}}{2}\left(1 + \cos\frac{\pi(t - t_i)}{T_i}\right), \tag{13.5}$$

where t_i marks the start of restart i, T_i is its period.

13.2 Warmup

13.2.1 Linear warmup

Start with very small learning rate, linearly increase to target:

$$\eta_t = \eta_{\min} + \frac{\eta_0 - \eta_{\min}}{T_{\text{warm}}} \cdot t, \quad t \in [0, T_{\text{warm}}]. \tag{13.6}$$

After T_{warm} iterations, switch to standard schedule.

Why: Prevents extreme parameter updates early in training when initialization is random and gradients are unreliable. Especially important for Transformers and other large models.

13.2.2 Gradient accumulation + warmup

With gradient accumulation (computing loss on mini-batches, accumulating gradients, updating after k mini-batches), warmup typically spans the accumulated steps:

$$\eta_t = \eta_{\min} + \frac{\eta_0 - \eta_{\min}}{k \cdot T_{\text{warm}}} \cdot t. \tag{13.7}$$

13.3 Hyperparameter Search Methods

13.3.1 Grid search

Enumerate all combinations of discrete values:

$$(\eta, \beta, \lambda) \in \{\eta_1, \ldots, \eta_m\} \times \{\beta_1, \ldots, \beta_n\} \times \{\lambda_1, \ldots, \lambda_p\}. \tag{13.8}$$

Train $m \cdot n \cdot p$ models, select the best.

Disadvantage: Combinatorial explosion; many hyperparameters become infeasible.

13.3.2 Random search

Sample hyperparameters uniformly (or from a prior) for N trials:

$$(\eta^{(i)}, \beta^{(i)}, \lambda^{(i)}) \sim p(\eta, \beta, \lambda), \quad i = 1, \ldots, N. \tag{13.9}$$

Train N models, select best.

Advantage: More efficient than grid search in high dimensions; discovers good regions faster.

13.3.3 Bayesian optimization

Model the objective (e.g., validation loss) as a Gaussian process:

$$f(\mathbf{h}) \sim \mathcal{GP}(\mu(\mathbf{h}), k(\mathbf{h}, \mathbf{h}')) \tag{13.10}$$

where $\mathbf{h}$ is the hyperparameter vector.

At each iteration:

1. Fit GP to observed trials.
2. Define an acquisition function (e.g., Expected Improvement) balancing exploration and exploitation.
3. Select next hyperparameters to maximize acquisition.
4. Train and observe outcome.
5. Repeat.

Complexity: Higher computational cost per iteration (fitting GP) but fewer total trials needed.

Chapter 14

Data Preprocessing and Normalization

Before training, data and intermediate activations should be normalized for stability and convergence.

14.1 Input Normalization

14.1.1 Standardization (Z-score)

Center and scale each feature:

$$x_i' = \frac{x_i - \mu_i}{\sigma_i}, \tag{14.1}$$

where $\mu_i = \frac{1}{n}\sum_{j=1}^{n} x_{ij}$, $\sigma_i = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(x_{ij} - \mu_i)^2}$.

Assumption: Features are roughly normally distributed.

14.1.2 Min-Max scaling

Scale to fixed range (e.g., $[0, 1]$):

$$x_i' = \frac{x_i - \min_j x_{ij}}{\max_j x_{ij} - \min_j x_{ij}}. \tag{14.2}$$

Use: When you want bounded values; sensitive to outliers.

14.1.3 Data statistics (train vs. test)

Compute μ, σ (or min, max) on training data. Apply the same transformation to test data. **Never** compute statistics on test data; this leaks test information into the model.

14.2 Batch Normalization (Revisited)

From Chapter 7/8, batch normalization normalizes layer inputs within mini-batches:

$$\hat{z}_i^{(\ell)} = \frac{z_i^{(\ell)} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}, \quad y_i^{(\ell)} = \gamma \hat{z}_i^{(\ell)} + \beta. \tag{14.3}$$

14.2.1 Running mean and variance (inference)

During training, use batch statistics. During inference, use a running average computed across training:

$$\mu_{\text{run}} \leftarrow \alpha \mu_{\text{run}} + (1-\alpha)\mu_B, \quad \sigma^2_{\text{run}} \leftarrow \alpha \sigma^2_{\text{run}} + (1-\alpha)\sigma_B^2, \tag{14.4}$$

where $\alpha \approx 0.9$ or 0.99 (momentum).

14.3 Layer Normalization

Layer normalization (LayerNorm) computes statistics per sample and layer, not per batch. For a layer input $z^{(\ell)} \in \mathbb{R}^{n_\ell}$ (single sample):

$$\mu^{(\ell)} = \frac{1}{n_\ell} \sum_{i=1}^{n_\ell} z_i^{(\ell)}, \quad \sigma^{(\ell),2} = \frac{1}{n_\ell} \sum_{i=1}^{n_\ell} (z_i^{(\ell)} - \mu^{(\ell)})^2, \tag{14.5}$$

$$\hat{z}_i^{(\ell)} = \frac{z_i^{(\ell)} - \mu^{(\ell)}}{\sqrt{\sigma^{(\ell),2} + \varepsilon}}, \quad y_i^{(\ell)} = \gamma_i \hat{z}_i^{(\ell)} + \beta_i. \tag{14.6}$$

Advantage: Not dependent on batch statistics; works well with small batches, RNNs, and Transformers. Learnable scale γ and shift β are usually per-feature (size n_ℓ).

14.4 Group Normalization and Instance Normalization

14.4.1 Group normalization

Divide channels into G groups, normalize within each group:

$$\mu^{(g)} = \frac{1}{S/G} \sum_{s \in \text{group } g} z_s, \quad \sigma^{(g),2} = \frac{1}{S/G} \sum_{s \in \text{group } g} (z_s - \mu^{(g)})^2, \tag{14.7}$$

where $S = C \cdot H \cdot W$ (total features per sample).

Use: Works well when batch size is small (e.g., 1–4).

14.4.2 Instance normalization

Normalize each feature map (channel) independently:

$$\mu^{(c)} = \frac{1}{H \cdot W} \sum_{h,w} z_{c,h,w}, \quad \sigma^{(c),2} = \frac{1}{H \cdot W} \sum_{h,w} (z_{c,h,w} - \mu^{(c)})^2. \tag{14.8}$$

Use: Style transfer, image generation. Normalizes per-instance statistics, removing instance-specific information.

14.5 Comparison of Normalization Methods

Method	Computes	Statistics on	Best for
Batch Norm	μ_B, σ_B	Batch	Large batch, CNNs
Layer Norm	μ, σ per sample	Layer features	RNNs, Transformers
Group Norm	μ, σ per group	Groups of channels	Small batch
Instance Norm	μ, σ per channel	Channel	Style transfer

Chapter 15

Recurrent Neural Networks (RNNs)

15.1 Sequence Data and Mathematical Formulation

15.1.1 Temporal Data Representation

For sequence data, we denote:

- Input sequence: $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T)}$ where each $\mathbf{x}^{(t)} \in \mathbb{R}^{d_x}$
- Target sequence: $\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(T)}$ (for many-to-many tasks)
- Sequence length T may vary across examples

Unlike feedforward networks, RNNs process sequences one timestep at a time, maintaining an internal state.

15.1.2 Notation and Convention

Let:

- $\mathbf{h}^{(t)} \in \mathbb{R}^{d_h}$ be the hidden state at time t
- d_h be the hidden dimension
- $\mathbf{h}^{(0)} = \mathbf{0}$ (zero initialization)

For mini-batch processing with m sequences stacked as columns:

$$\mathbf{H}^{(t)} = [\mathbf{h}^{(t,1)}, \mathbf{h}^{(t,2)}, \ldots, \mathbf{h}^{(t,m)}] \in \mathbb{R}^{d_h \times m} \tag{15.1}$$

15.1.3 Common Task Architectures

Many-to-one (e.g., sentiment classification):

- Input: $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}$
- Output: single $\hat{\mathbf{y}}$ from final hidden state

One-to-many (e.g., image captioning):

- Input: single $\mathbf{x}$

- Output: $\hat{\mathbf{y}}^{(1)}, \ldots, \hat{\mathbf{y}}^{(T')}$

Many-to-many (e.g., machine translation):

- Input sequence: $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T_{\text{in}})}$
- Output sequence: $\hat{\mathbf{y}}^{(1)}, \ldots, \hat{\mathbf{y}}^{(T_{\text{out}})}$

15.2 Vanilla RNN Definition and Forward Propagation

15.2.1 Recurrent Computation

At each timestep $t = 1, 2, \ldots, T$, the Vanilla RNN computes:

$$\mathbf{z}_h^{(t)} = \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{W}_{xh}\mathbf{x}^{(t)} + \mathbf{b}_h \tag{15.2}$$

$$\mathbf{h}^{(t)} = \sigma_h(\mathbf{z}_h^{(t)}) \tag{15.3}$$

$$\mathbf{z}_y^{(t)} = \mathbf{W}_{hy}\mathbf{h}^{(t)} + \mathbf{b}_y \tag{15.4}$$

$$\hat{\mathbf{y}}^{(t)} = \sigma_y(\mathbf{z}_y^{(t)}) \tag{15.5}$$

where:

- $\mathbf{W}_{hh} \in \mathbb{R}^{d_h \times d_h}$ (hidden-to-hidden weights)
- $\mathbf{W}_{xh} \in \mathbb{R}^{d_h \times d_x}$ (input-to-hidden weights)
- $\mathbf{W}_{hy} \in \mathbb{R}^{d_y \times d_h}$ (hidden-to-output weights)
- σ_h is typically tanh or ReLU
- σ_y depends on the task (softmax for classification, sigmoid for binary, linear for regression)

15.2.2 Parameter Sharing Across Time

The key insight of RNNs is **parameter sharing**: the same parameters ($\mathbf{W}_{hh}, \mathbf{W}_{xh}, \mathbf{W}_{hy}, \mathbf{b}_h, \mathbf{b}_y$) are used at every timestep. This is why the model can handle variable-length sequences.

15.2.3 Vectorized Mini-batch Forward Pass

For a mini-batch, stack hidden states and inputs as matrices:

$$\mathbf{Z}_h^{(t)} = \mathbf{W}_{hh}\mathbf{H}^{(t-1)} + \mathbf{W}_{xh}\mathbf{X}^{(t)} + \mathbf{b}_h\mathbf{1}^\top \tag{15.6}$$

$$\mathbf{H}^{(t)} = \sigma_h(\mathbf{Z}_h^{(t)}) \tag{15.7}$$

$$\mathbf{Z}_y^{(t)} = \mathbf{W}_{hy}\mathbf{H}^{(t)} + \mathbf{b}_y\mathbf{1}^\top \tag{15.8}$$

$$\hat{\mathbf{Y}}^{(t)} = \sigma_y(\mathbf{Z}_y^{(t)}) \tag{15.9}$$

where $\mathbf{1} \in \mathbb{R}^m$ is the all-ones vector.

15.3 Computational Graph Unrolling in Time

15.3.1 Unrolled Graph Representation

When we unfold the RNN across T timesteps, we obtain a computational graph that is a directed acyclic graph (DAG):

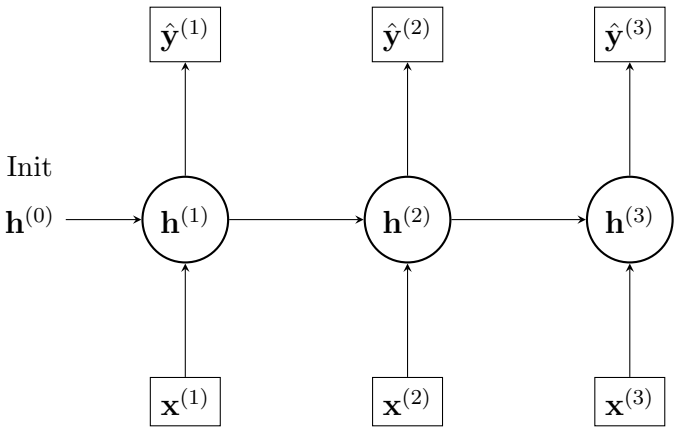

Figure 15.1: Unrolled Recurrent Neural Network. The hidden state $\mathbf{h}^{(t)}$ passes information to the next timestep. (Adapted from Goodfellow et al., 2016)

Each "RNN cell" at time t depends on:

1. Current input $\mathbf{x}^{(t)}$
2. Previous hidden state $\mathbf{h}^{(t-1)}$

15.3.2 Temporal Dependencies

The hidden state $\mathbf{h}^{(t)}$ depends on all previous inputs:

$$\mathbf{h}^{(t)} = f_t(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(t)}) \tag{15.10}$$

This creates a long chain of dependencies, which will be crucial for understanding gradient flow.

15.4 Backpropagation Through Time (BPTT)

15.4.1 Loss Function and Objective

For a sequence, the total loss is:

$$L = \sum_{t=1}^{T} L^{(t)} \tag{15.11}$$

where $L^{(t)} = \ell(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})$ is the loss at timestep t (e.g., cross-entropy for classification).

15.4.2 Backpropagation Through Time Algorithm

The key insight is that the gradient at each timestep comes from two sources:

$$\frac{\partial L}{\partial \mathbf{h}^{(t)}} = \frac{\partial L^{(t)}}{\partial \mathbf{h}^{(t)}} + \frac{\partial L^{(t+1:T)}}{\partial \mathbf{h}^{(t)}} \tag{15.12}$$

Derivation:
By the chain rule and the flow of gradients from the computational graph:

$$\frac{\partial L^{(t+1:T)}}{\partial \mathbf{h}^{(t)}} = \frac{\partial L^{(t+1:T)}}{\partial \mathbf{h}^{(t+1)}} \frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}} \tag{15.13}$$

Let $\boldsymbol{\delta}_h^{(t)} = \frac{\partial L}{\partial \mathbf{z}_h^{(t)}}$ be the delta (error signal) at the hidden layer pre-activation.
From the output layer:

$$\frac{\partial L^{(t)}}{\partial \mathbf{h}^{(t)}} = \mathbf{W}_{hy}^{\top} \frac{\partial L^{(t)}}{\partial \mathbf{z}_y^{(t)}} \tag{15.14}$$

From the next timestep (via the recurrent connection):

$$\frac{\partial L^{(t+1:T)}}{\partial \mathbf{h}^{(t)}} = \mathbf{W}_{hh}^{\top} \boldsymbol{\delta}_h^{(t+1)} \tag{15.15}$$

Therefore:

$$\boldsymbol{\delta}_h^{(t)} = \left(\mathbf{W}_{hy}^{\top} \boldsymbol{\delta}_y^{(t)} + \mathbf{W}_{hh}^{\top} \boldsymbol{\delta}_h^{(t+1)} \right) \odot \sigma_h'(\mathbf{z}_h^{(t)}) \tag{15.16}$$

where $\boldsymbol{\delta}_y^{(t)} = \frac{\partial L^{(t)}}{\partial \mathbf{z}_y^{(t)}}$ is the output layer delta.

15.4.3 Parameter Gradients

The gradients for the weight matrices are computed by summing contributions from all timesteps:

$$\frac{\partial L}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{z}_h^{(t)}} \frac{\partial \mathbf{z}_h^{(t)}}{\partial \mathbf{W}_{hh}} \tag{15.17}$$

$$= \sum_{t=1}^{T} \boldsymbol{\delta}_h^{(t)} (\mathbf{h}^{(t-1)})^{\top} \tag{15.18}$$

Similarly:

$$\frac{\partial L}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \boldsymbol{\delta}_h^{(t)} (\mathbf{x}^{(t)})^\top \tag{15.19}$$

$$\frac{\partial L}{\partial \mathbf{W}_{hy}} = \sum_{t=1}^{T} \boldsymbol{\delta}_y^{(t)} (\mathbf{h}^{(t)})^\top \tag{15.20}$$

For biases:

$$\frac{\partial L}{\partial \mathbf{b}_h} = \sum_{t=1}^{T} \boldsymbol{\delta}_h^{(t)} \tag{15.21}$$

$$\frac{\partial L}{\partial \mathbf{b}_y} = \sum_{t=1}^{T} \boldsymbol{\delta}_y^{(t)} \tag{15.22}$$

15.5 Vanishing and Exploding Gradients: Mathematical Analysis

15.5.1 Gradient Flow Through Hidden States

The gradient of the loss with respect to a distant hidden state $\mathbf{h}^{(k)}$ (where $k < t$) involves a product of Jacobians:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}} \tag{15.23}$$

Derivation:
By the chain rule:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} \frac{\partial \mathbf{h}^{(t-1)}}{\partial \mathbf{h}^{(t-2)}} \cdots \frac{\partial \mathbf{h}^{(k+1)}}{\partial \mathbf{h}^{(k)}} \tag{15.24}$$

Each Jacobian $\frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}}$ has the form:

$$\frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}} = \frac{\partial \sigma_h(\mathbf{z}_h^{(j)})}{\partial \mathbf{z}_h^{(j)}} \frac{\partial \mathbf{z}_h^{(j)}}{\partial \mathbf{h}^{(j-1)}} \tag{15.25}$$

$$= \operatorname{diag}(\sigma_h'(\mathbf{z}_h^{(j)})) \mathbf{W}_{hh} \tag{15.26}$$

15.5.2 Spectral Analysis

For stability, we analyze the spectral properties. The product of Jacobians can be approximated by:

$$\left\| \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}} \right\| \lesssim \|\sigma_h'\|_\infty^{t-k} \|\mathbf{W}_{hh}\|^{t-k} \tag{15.27}$$

For tanh activation, $|\sigma_h'(z)| \leq 1$ for all z, and the maximum is $1/4$ at $z = 0$.
Let $\rho = \lambda_{\max}(\mathbf{W}_{hh})$ be the spectral radius (largest eigenvalue magnitude).

- **Vanishing gradients:** If $\rho < 1$, then $\|\mathbf{W}_{hh}\|^{t-k} \to 0$ as $t - k \to \infty$.

  - Consequence: Gradients for distant timesteps become negligible.
  - Learning long-term dependencies becomes slow.

- **Exploding gradients:** If $\rho > 1$, then $\|\mathbf{W}_{hh}\|^{t-k} \to \infty$ as $t - k \to \infty$.
  - Consequence: Gradients become unbounded; training becomes unstable.
  - Parameter updates may be very large, causing divergence.

15.5.3 Mathematical Condition for Stability

Define the **temporal condition number**:

$$\kappa_T = \rho^T \tag{15.28}$$

For long sequences (T large):

- If $\rho < 1$: exponential decay of $\kappa_T \to 0$ (vanishing)
- If $\rho > 1$: exponential growth of $\kappa_T \to \infty$ (exploding)
- If $\rho = 1$: $\kappa_T = 1$ (critically balanced, unstable in practice)

15.6 Common Questions (RNNs)

15.6.1 Q1: Why do we share $\mathbf{W}_{hh}$?

A: Parameter sharing allows the RNN to apply the same "rule" regardless of the sequence length.

Example:

- **Without sharing**: A sequence of length 100 and length 1000 would require different networks (different parameters).
- **With sharing**: The same $\mathbf{W}_{hh}$ is used from time 1 to 100, and from 100 to 1000.

Mathematical view:

$$\mathbf{h}^{(T)} = \sigma(\mathbf{W}_{hh} \cdots \sigma(\mathbf{W}_{hh}\mathbf{h}^{(1)})) \tag{15.29}$$

By applying $\mathbf{W}_{hh}$ repeatedly, the model can handle **variable-length sequences**.

Trade-off: Gradients become a long product of $\mathbf{W}_{hh}$, making them prone to vanishing/exploding.

15.6.2 Q2: What exactly is "vanishing gradient"?

A: It is a state where parameter updates become nearly zero, stopping the model from learning.

Numerical example:

- Processing a 100-step sentence with Vanilla RNN.
- $|\sigma'| \approx 0.5$ at each step (moderate gradient for tanh).

- Gradient magnitude: $0.5^{100} \approx 10^{-30}$ (nearly zero!)

Practical impact:

- The influence of the 1st word on the 100th output becomes unmeasurable.
- The model relies only on "recent words" and cannot learn long-term dependencies.

Example: Translation task

```
"The quick brown fox jumps over the lazy dogs are ___"
        ^ Subject (long distance)          ^ Predicate (end)
```

The RNN tries to complete "dogs are" using only local context, missing the singular/-plural agreement with "fox".

15.6.3 Q3: What is the computational cost of BPTT?

A: Full BPTT requires storing states for all timesteps, which is memory-inefficient.
Memory usage:

- Sequence length $T = 1000$, Hidden dim $d_h = 512$, Float32 (4 bytes).
- **Memory required**: $1000 \times 512 \times 4 = 2.048$ MB (per sequence).
- **Batch size 32**: 65.5 MB.

Since data must be kept for gradient computation across all steps, it is memory-heavy.
Solution: Truncated BPTT ($\tau \approx 50$), considering only the past 50 steps.

15.6.4 Q4: Why can't RNNs be parallelized?

A: Because $\mathbf{h}^{(t)}$ depends on $\mathbf{h}^{(t-1)}$, calculations must respect chronological order.
Dependency graph:

```
h(1) -> h(2) -> h(3) -> h(4)
 |       |       |       |
x(1)    x(2)    x(3)    x(4)
```

Computation time:

- RNN: $O(T)$ (Sequential).
- Transformer: $O(\log T)$ or $O(1)$ (Parallelizable).

In LLMs, parallel efficiency dictates training speed, giving Transformers a huge advantage.

15.7 Common Questions (Gradient Problems)

15.7.1 Q5: What is the danger of exploding gradients?

A: Updates become massive, causing parameters to overshoot the optimal solution.

Numerical example:

```
# Exploding gradient
gradient = 1e8
learning_rate = 0.001
weight_update = 100000  # Massive update!

# Normal
gradient = 1.0
weight_update = 0.001   # Stable
```

Impact on Loss Curve: Instead of converging, the loss oscillates wildly or diverges to NaN.

Solution:

1. **Gradient Clipping**: $\mathbf{g} \leftarrow \theta \frac{\mathbf{g}}{\|\mathbf{g}\|}$ if $\|\mathbf{g}\| > \theta$.
2. **Weight Initialization**: Keep $\|\mathbf{W}_{hh}\|$ small.

15.7.2 Q6: Why is the spectral radius important?

A: The largest eigenvalue ρ of $\mathbf{W}_{hh}$ determines the rate of gradient growth/decay.

Intuition:

- $\rho = 0.9$: Gradient $\approx 0.9^{100} \approx 0$ (Vanishing).
- $\rho = 1.0$: Gradient ≈ 1 (Critical boundary).
- $\rho = 1.1$: Gradient $\approx 1.1^{100} \approx 14000$ (Exploding).

Implication: Initialize weights such that the spectral radius is reasonable (e.g., Xavier initialization, Orthogonal initialization).

Part I

Embodied Intelligence and Robot Learning

Chapter 16

Embodied AI Fundamentals

16.1 Introduction

This chapter marks a fundamental shift in the mathematical framework from static supervised learning (Chapters 115) to **sequential decision-making in continuous, partially observable environments**. Unlike image classification or language modeling where data is i.i.d. and fully observed, robot control involves:

1. **Partial Observability**: The agent observes only sensor readings $(\mathbf{I}_t, \mathbf{q}_t, \mathbf{f}_t)$, not the true state.

2. **Continuous Actions**: Control signals live in continuous spaces $(\mathbb{R}^D)$, not discrete vocabularies.

3. **Temporal Dependencies**: Optimal actions depend on the history of observations, requiring recurrent or attention-based encoders.

4. **Multimodality**: Many tasks admit multiple correct solutions, violating the single-answer assumption of supervised learning.

This chapter provides the mathematical foundation for Chapters 1720.

16.2 Markov Decision Processes (MDPs)

Markov Decision Processes (MDPs) provide the fundamental mathematical framework for reinforcement learning, modeling decision-making in situations where outcomes are partly random and partly under the control of a decision maker.

16.2.1 Basic Definition

An MDP is a discrete-time stochastic control process. It is characterized by the **Markov property**: the future state depends only on the current state and the action taken, not on the sequence of events that preceded it.

16.2.2 Key Components

An MDP is defined by the following five elements:

- **State Space ($\mathcal{S}$)**: The set of all possible states of the system (e.g., positions in a maze).
- **Action Space ($\mathcal{A}$)**: The set of all actions available to the agent in each state (e.g., move up, down, left, right).
- **Transition Probability ($\mathcal{P}$)**: The probability that the system moves to state s' given that it is in state s and action a is taken, denoted as $P(s' \mid s, a)$.
- **Reward Function ($\mathcal{R}$)**: The immediate reward received after transitioning from state s to state s' due to action a, denoted as $R(s, a)$ or $R(s, a, s')$.
- **Discount Factor (γ)**: A parameter ($0 \leq \gamma < 1$) that determines the present value of future rewards.

16.2.3 Mathematical Representation

Formally, an MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$. The goal of the agent is to find a policy π (a strategy) that maximizes the expected cumulative reward. The value function $V^\pi(s)$ under a policy π is often defined using the Bellman equation:

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s',r} P(s', r \mid s, a) \left[r + \gamma V^\pi(s') \right] \tag{16.1}$$

16.2.4 Transition Probabilities

The transition probabilities $P(s' \mid s, a)$ are determined by the dynamics of the environment.

Determination Methods

They are typically specified by the problem designer based on physical laws, probabilistic models, or observed data.

- **Deterministic Environments**: The next state is determined with certainty ($P(s' \mid s, a) = 1$).
- **Stochastic Environments**: Transition probabilities are distributed (e.g., multinomial distribution), reflecting noise or uncertainty.

In robotics, these probabilities can be:

- **Pre-modeled**: Derived from physical simulations (e.g., using Gazebo) considering motor slip or friction.
- **Data-driven**: Estimated from real-world interaction data (e.g., frequency of transitions observed during trials).

16.2.5 The Value Function (V-function)

The state-value function $V^\pi(s)$ represents the expected return (cumulative discounted reward) starting from state s and following policy π.

Definition

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s \right] \tag{16.2}$$

Intuitively, it tells us "how good" it is to be in a specific state.

Derivation of the Bellman Expectation Equation

We can decompose the value function recursively:

1. Split into immediate and future rewards:

$$V^\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \tag{16.3}$$

2. Express future value in terms of the next state's value:

$$\mathbb{E}_\pi[G_{t+1} \mid S_t = s] = \mathbb{E}_\pi[V^\pi(S_{t+1}) \mid S_t = s] \tag{16.4}$$

3. Expand using policy and transition probabilities: Using the Law of Total Probability:

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s',r} P(s', r \mid s, a) \left[r + \gamma V^\pi(s') \right] \tag{16.5}$$

This is the Bellman Expectation Equation.

16.2.6 Utility of the Bellman Equation

The Bellman equation is crucial because:

1. **Recursive Decomposition**: It allows infinite-horizon problems to be solved using iterative finite computations (Dynamic Programming).
2. **Algorithmic Foundation**: It forms the basis for algorithms like Value Iteration, Policy Iteration, and Q-learning.
3. **Theoretical Guarantees**: It acts as a contraction mapping, guaranteeing the existence and uniqueness of an optimal value function.

16.3 Partially Observable Markov Decision Processes (POMDPs)

16.3.1 Formal Definition

Definition 16.1 (POMDP Tuple). A Partially Observable Markov Decision Process is a 6-tuple

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{R}, \gamma), \tag{16.6}$$

where:

- $\mathcal{S}$: State space (typically $\mathcal{S} \subseteq \mathbb{R}^{n_s}$, often high-dimensional and unknown)
- $\mathcal{A}$: Action space (continuous: $\mathcal{A} \subseteq \mathbb{R}^{n_a}$, or discrete: $\mathcal{A} = \{1, \ldots, K\}$)
- $\mathcal{O}$: Observation space (multimodal: images, joint angles, forces)
- $\mathcal{T}$: State transition model (dynamics)
- $\mathcal{R}$: Reward function
- γ: Discount factor ($0 \leq \gamma < 1$)

16.3.2 Components in Detail

State Space $\mathcal{S}$

The true state $\mathbf{s}_t \in \mathcal{S}$ fully describes the world configuration. For a robotic manipulation task:

$$\mathbf{s}_t = (\mathbf{p}_{\text{obj}}, \mathbf{v}_{\text{obj}}, \mathbf{q}_{\text{robot}}, \dot{\mathbf{q}}_{\text{robot}}, \text{contact flags}, \ldots) \in \mathbb{R}^{n_s}, \tag{16.7}$$

where:

- $\mathbf{p}_{\text{obj}} \in \mathbb{R}^3$: Position of target object
- $\mathbf{v}_{\text{obj}} \in \mathbb{R}^3$: Velocity of object
- $\mathbf{q}_{\text{robot}} \in \mathbb{R}^D$: Joint angles (D-DOF manipulator)
- $\dot{\mathbf{q}}_{\text{robot}} \in \mathbb{R}^D$: Joint velocities
- Contact flags, friction coefficients, etc.: $\mathbb{R}^{n_{\text{other}}}$

For manipulation in the real world, n_s can be 50–100 dimensional. **Crucially, $\mathbf{s}_t$ is unknown to the agent**; we only observe partial projections.

Example (Drawer Opening Task):

$$\mathbf{s}_t = (\underbrace{\mathbf{p}_{\text{handle}}, \mathbf{p}_{\text{drawer\_body}}, \theta_{\text{joint}}}_{\text{object config}}, \underbrace{\mathbf{q}_{\text{arm}}, \dot{\mathbf{q}}_{\text{arm}}}_{\text{arm state}}, \underbrace{\mu_{\text{friction}}, \text{handle stiffness}}_{\text{hidden params}}) \tag{16.8}$$

Observation Space $\mathcal{O}$

The agent receives multimodal observations, typically:

$$\mathbf{o}_t = (\mathbf{I}_t^{(\text{wrist})}, \mathbf{I}_t^{(\text{overhead})}, \mathbf{q}_t, \mathbf{f}_t) \in \mathcal{O}. \tag{16.9}$$

Visual Component:

$$\mathbf{I}_t^{(c)} \in \mathbb{R}^{H \times W \times 3}, \tag{16.10}$$

where $c \in \{\text{wrist}, \text{overhead}, \ldots\}$ indexes camera viewpoints, typically $H = W = 84$ or 224 for deep learning pipelines.

Proprioceptive Component:

$$\mathbf{q}_t \in \mathbb{R}^D \quad \text{(joint angles)} \tag{16.11}$$

$$\dot{\mathbf{q}}_t \in \mathbb{R}^D \quad \text{(joint velocities, optional)} \tag{16.12}$$

Force/Torque Component (if available):

$$\mathbf{f}_t \in \mathbb{R}^6, \quad \mathbf{f}_t = (\mathbf{F}_{\text{linear}}, \boldsymbol{\tau}_{\text{rotational}}) \tag{16.13}$$

Dimension Analysis:

$$\dim(\mathcal{O}) = (\text{num cameras}) \times (H \times W \times 3) + D + D + 6 \tag{16.14}$$

$$= 2 \times (84 \times 84 \times 3) + 7 + 7 + 6 = 42,258 + 20 \approx 42k \tag{16.15}$$

This high dimensionality necessitates representation learning (vision backbones) before action prediction.

Action Space $\mathcal{A}$

For continuous control (this course's focus):

$$\mathcal{A} = \mathbb{R}^{n_a}, \quad \text{e.g., } n_a \in \{3, 6, 7, 14\} \tag{16.16}$$

Interpretation (Position Control):

$$\mathbf{a}_t = (\Delta\mathbf{q}_{\text{arm}}, \delta_{\text{gripper}}) \in \mathbb{R}^D, \tag{16.17}$$

where $\Delta\mathbf{q}_{\text{arm}} \in \mathbb{R}^6$ is a relative joint position change (e.g., in [-0.1, 0.1] radians) and $\delta_{\text{gripper}} \in [-1, 1]$ is the gripper command.

Interpretation (Velocity Control):

$$\mathbf{a}_t = (\dot{\mathbf{q}}_{\text{desired}}, \text{gripper speed}) \in \mathbb{R}^D. \tag{16.18}$$

The key constraint: $\mathcal{A}$ is **bounded** (actuator limits), so we may write

$$\mathcal{A} = [-1, 1]^{n_a} \quad \text{(normalized)} \tag{16.19}$$

or

$$\mathcal{A} = \prod_{i=1}^{n_a} [a_i^{\min}, a_i^{\max}]. \tag{16.20}$$

Transition Model $\mathcal{T}$

The transition model specifies how the world evolves:

$$\mathcal{T}(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S}), \tag{16.21}$$

where $\Delta(\mathcal{S})$ is the space of probability distributions over $\mathcal{S}$.

In deterministic settings (idealized):

$$\mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{a}_t) \quad \text{(physics dynamics)}. \tag{16.22}$$

In stochastic settings (realistic):

$$\mathbf{s}_{t+1} \sim \mathcal{T}(\cdot|\mathbf{s}_t, \mathbf{a}_t) = \mathcal{N}(f(\mathbf{s}_t, \mathbf{a}_t), \Sigma_{\text{dynamics}}), \tag{16.23}$$

where f is a learned or hand-coded physics function and Σ_{dynamics} captures model uncertainty.

Observation Model: The observation is a noisy projection of the state:

$$\mathbf{o}_t \sim \mathcal{P}(\cdot|\mathbf{s}_t), \tag{16.24}$$

where $\mathcal{P}$ is the observation model. In practice, this is often deterministic (perfect sensors):

$$\mathbf{o}_t = h(\mathbf{s}_t) + \boldsymbol{\epsilon}_{\text{sensor}}, \quad \boldsymbol{\epsilon}_{\text{sensor}} \sim \mathcal{N}(\mathbf{0}, \Sigma_{\text{obs}}). \tag{16.25}$$

For computer vision, this is implicit: the camera captures a projection of the 3D world.

Reward Function $\mathcal{R}$

In imitation learning (ACT, Diffusion Policy), the reward is **implicit** (learned from demonstrations). In reinforcement learning, it is explicit:

$$\mathcal{R}(\mathbf{s}_t, \mathbf{a}_t) \in \mathbb{R}, \tag{16.26}$$

measuring task progress.

Example (Reaching Task):

$$\mathcal{R}(\mathbf{s}_t, \mathbf{a}_t) = \begin{cases} 1 & \text{if } \|\mathbf{p}_{\text{gripper}} - \mathbf{p}_{\text{target}}\|_2 < \epsilon \\ -\|\mathbf{p}_{\text{gripper}} - \mathbf{p}_{\text{target}}\|_2^2 - \lambda\|\mathbf{a}_t\|_2^2 & \text{otherwise} \end{cases}. \tag{16.27}$$

Example (Drawer Opening):

$$\mathcal{R}(\mathbf{s}_t) = \begin{cases} +10 & \text{if drawer opened} > 0.2 \text{ m} \\ -0.1 & \text{per timestep (action cost)} \end{cases}. \tag{16.28}$$

Discount Factor γ

The discount factor weighs immediate vs. future rewards:

$$\gamma \in [0, 1), \quad \text{typical: } \gamma \in \{0.95, 0.99, 0.999\}. \tag{16.29}$$

For **finite-horizon** tasks (e.g., pick-and-place in 10 seconds), γ is less critical. For open-ended tasks, γ controls the planning horizon.

16.3.3 Policy Representation

Definition 16.2 (Policy in POMDPs). A policy π is a mapping from **observation histories** to action distributions:

$$\pi : \mathcal{H} \to \Delta(\mathcal{A}), \tag{16.30}$$

where $\mathcal{H} = \mathcal{O}^* = \bigcup_{t=0}^{\infty} \mathcal{O}^t$ is the space of finite observation histories.

Formally, at timestep t:

$$\pi(\mathbf{a}_t|\mathbf{o}_{0:t}) = P(\mathbf{a}_t|\mathbf{o}_0, \mathbf{o}_1, \ldots, \mathbf{o}_t). \tag{16.31}$$

This explicitly captures that actions depend on the **full history**, not just the current observation (unlike MDPs where $\pi(\mathbf{a}_t|\mathbf{s}_t)$ suffices due to the Markov property).

Practical Implementation (Recurrent State): To avoid storing full histories, we use a recurrent encoder ϕ_{history} with hidden state $\mathbf{h}_t$:

$$\mathbf{h}_{t+1} = \phi_{\text{history}}(\mathbf{h}_t, \mathbf{o}_{t+1}; \theta_{\text{enc}}), \tag{16.32}$$

$$\pi(\mathbf{a}_t|\mathbf{h}_t) = P(\mathbf{a}_t|\phi_{\text{context}}(\mathbf{h}_t; \theta_{\text{dec}})). \tag{16.33}$$

This is the approach taken by ACT, Diffusion Policy, and VLA models, where the Transformer encoder maintains a **latent context** that implicitly summarizes relevant history.

16.4 Why POMDPs Are Different from Supervised Learning

16.4.1 Conceptual Differences

Aspect	Supervised Learning (Ch. 115)	POMDP / Robot Learning
Data Distribution	i.i.d. samples from fixed P_{data}	Non-stationary; depends on policy
Feedback	Immediate: compare $\hat{y}$ to ground truth y	Delayed: reward may come after 1
Optimality	Minimize empirical risk $\frac{1}{m}\sum \ell(\hat{y}_i, y_i)$	Maximize cumulative discounted re
Exploration	Not needed (all examples observed once)	Critical: agent must discover good
Multimodality	typically single correct answer	Multiple valid solutions (e.g., grasp

16.4.2 Action Space Topology: Discrete vs. Continuous

This is the most **mathematical** distinction. The output space topology fundamentally changes how we formulate learning objectives.

Discrete (NLP / Language Modeling):

$$P(\mathbf{w}_{t+1}|\mathbf{w}_{<t}) \in \Delta(\mathcal{V}), \quad |\mathcal{V}| \approx 50k, \tag{16.34}$$

where $\mathcal{V} = \{\text{word}_1, \text{word}_2, \ldots\}$ is a finite vocabulary.

Loss: **cross-entropy** on categorical distribution

$$\mathcal{L}_{\text{NLP}} = -\sum_{v=1}^{|\mathcal{V}|} y_v \log \hat{p}_v, \quad y_v \in \{0, 1\}, \quad \sum_v y_v = 1. \tag{16.35}$$

Continuous (Robotics / Control):

$$P(\mathbf{a}_t|\mathbf{o}_{0:t}) = \mathcal{N}(\boldsymbol{\mu}(\mathbf{o}_{0:t}), \Sigma(\mathbf{o}_{0:t})), \tag{16.36}$$

where $\boldsymbol{\mu} \in \mathbb{R}^{n_a}$ and $\Sigma \in \mathbb{R}_+^{n_a \times n_a}$ (positive definite).

Loss: **negative log-likelihood** of Gaussian

$$\mathcal{L}_{\text{robot}} = -\log \mathcal{N}(\mathbf{a}_t; \boldsymbol{\mu}, \Sigma) = \frac{1}{2}\log|2\pi\Sigma| + \frac{1}{2}(\mathbf{a}_t - \boldsymbol{\mu})^\top \Sigma^{-1}(\mathbf{a}_t - \boldsymbol{\mu}). \tag{16.37}$$

For diagonal covariance $\Sigma = \text{diag}(\sigma_1^2, \ldots, \sigma_{n_a}^2)$:

$$\mathcal{L}_{\text{robot}} = \frac{n_a}{2}\log(2\pi) + \sum_{i=1}^{n_a} \log \sigma_i + \frac{1}{2\sigma_i^2}(a_i - \mu_i)^2. \tag{16.38}$$

16.4.3 The Multimodality Problem

Definition 16.3 (Task Multimodality). A task exhibits multimodality if the conditional distribution $P(\mathbf{a}_{t:t+k}|\mathbf{o}_t)$ of actions has **multiple modes** (local maxima) with significant probability mass.

Concrete Example: Grasping a Cup

The task "grasp the cup" admits infinitely many solutions:

- **Mode 1 (Left Grasp):** Approach from the left, angle $\theta_1 = 30°$

$$\mathbf{a}^{(1)}_{t:t+k} = (x = -5, y = 0, z = -2, \theta = 30°, \text{gripper} = \text{close}) \tag{16.39}$$

- **Mode 2 (Right Grasp):** Approach from the right, angle $\theta_2 = -30°$

$$\mathbf{a}^{(2)}_{t:t+k} = (x = +5, y = 0, z = -2, \theta = -30°, \text{gripper} = \text{close}) \tag{16.40}$$

- **Mode 3 (Top Grasp):** Approach from above

$$\mathbf{a}^{(3)}_{t:t+k} = (x = 0, y = 0, z = -5, \text{approach vertical}, \text{gripper} = \text{close}) \tag{16.41}$$

Averaging Modes is Catastrophic: If we naively average multiple demonstrations:

$$\bar{\mathbf{a}} = \frac{1}{3}(\mathbf{a}^{(1)} + \mathbf{a}^{(2)} + \mathbf{a}^{(3)}) = (x = 0, y = 0, z = -3, \theta = 0°, \text{gripper} = \text{partially close}), \tag{16.42}$$

this trajectory **crashes into the cup** from directly above with insufficient gripper closure. None of the original solutions' physics is preserved.

Mathematical Formulation: Let $\mathbf{A}^*$ be the set of valid action sequences. For multimodal tasks:

$$P(\mathbf{a}_{t:t+k}|\mathbf{o}_t) = \sum_{m=1}^{M} p_m \mathcal{N}(\mathbf{a}_{t:t+k}; \boldsymbol{\mu}_m, \Sigma_m), \quad M > 1, \quad p_m > 0. \tag{16.43}$$

Naive Behavioral Cloning (fails):

$$\mathcal{L}_{\text{BC}} = \mathbb{E}_{(\mathbf{o},\mathbf{a})\sim\mathcal{D}} \left[\|\mathbf{a} - \mu_{\text{network}}(\mathbf{o})\|_2^2 \right]. \tag{16.44}$$

By regression to the mean, $\mu_{\text{network}}(\mathbf{o}) \to \frac{1}{N_{\text{demos}}} \sum_i \mathbf{a}_i^{(demo)}$, which lies in the **gap between modes**.

Solution (Latent Variable Models): ACT and Diffusion Policy address this by introducing a **latent variable z** that selects among modes:

$$P(\mathbf{a}_{t:t+k}|\mathbf{o}_t) = \int P(\mathbf{a}_{t:t+k}|\mathbf{z}, \mathbf{o}_t) P(\mathbf{z}|\mathbf{o}_t) d\mathbf{z}. \tag{16.45}$$

Each latent value $\mathbf{z}$ corresponds to a different mode, allowing the policy to generate coherent, physically realistic action sequences.

16.5 Observation Encoding: From High-Dimensional Images to Features

16.5.1 Vision Backbone Architecture

Robot policies receive images as observations. The first processing stage is a **vision backbone**: a pre-trained or learned convolutional encoder that reduces $\mathbb{R}^{H\times W\times 3}$ to feature vectors.

Definition 16.4 (Vision Backbone). A vision backbone is a function

$$\Phi_{\text{vision}} : \mathbb{R}^{C_{\text{in}}\times H\times W} \to \mathbb{R}^{C_{\text{out}}\times H'\times W'}, \tag{16.46}$$

where $C_{\text{in}} = 3$ (RGB), and (C_{out}, H', W') are architecture-dependent.

Common Architectures:

Model	Output Dim	Spatial Size	Comp. Cost (MACs)
ResNet-18 (no avg pool)	512	7×7 (84px in)	1.8 B
ResNet-34	512	7×7	3.6 B
EfficientNet-B0	1280	3×3	0.4 B
ViT-Base	768	7×7 (patches)	8 B

For ACT and Diffusion Policy, we use **ResNet-18 without the final global average pooling**, preserving spatial structure for Transformer processing.

Forward Pass (ResNet-18 on 8484 input):

Layer	Output Spatial Size	Output Channels
Input	84×84	3
conv1 (stride=2)	42×42	64
maxpool (stride=2)	21×21	64
layer1 (4 blocks, stride=1)	21×21	64
layer2 (4 blocks, stride=2)	11×11	128
layer3 (4 blocks, stride=2)	6×6	256
layer4 (4 blocks, stride=2)	3×3	512

Feature Extraction (Concrete Dimensions): For a batch of 2 camera images at 84×84:

$$\mathbf{I} \in \mathbb{R}^{B\times 2\times 3\times 84\times 84}, \tag{16.47}$$

after ResNet-18:

$$\mathbf{F} = \Phi_{\text{vision}}(\mathbf{I}) \in \mathbb{R}^{B\times 2\times 512\times 3\times 3}. \tag{16.48}$$

16.5.2 From Feature Maps to Tokens

To integrate vision features with a Transformer encoder, we **tokenize** the spatial feature map.

Definition 16.5 (Vision Tokenization). Given a feature map $\mathbf{F} \in \mathbb{R}^{C_{\text{out}}\times H'\times W'}$, we flatten spatial dimensions to create a sequence of **visual tokens**:

$$\text{Tokens}_{\text{vis}} = \text{Reshape}(\mathbf{F}, (H'W', C_{\text{out}})) \in \mathbb{R}^{H'W'\times C_{\text{out}}}. \tag{16.49}$$

Example (Two Cameras):

$$\mathbf{F}_{\text{wrist}} \in \mathbb{R}^{512\times3\times3} \to \text{Tokens}_{\text{wrist}} \in \mathbb{R}^{9\times512} \tag{16.50}$$

$$\mathbf{F}_{\text{overhead}} \in \mathbb{R}^{512\times3\times3} \to \text{Tokens}_{\text{overhead}} \in \mathbb{R}^{9\times512} \tag{16.51}$$

$$\text{Tokens}_{\text{vis}} = [\text{Tokens}_{\text{wrist}}; \text{Tokens}_{\text{overhead}}] \in \mathbb{R}^{18\times512}. \tag{16.52}$$

Positional Embeddings (Optional but Recommended): To preserve spatial information, add learnable positional embeddings:

$$\text{Tokens}_{\text{vis}}^{(i)} \leftarrow \text{Tokens}_{\text{vis}}^{(i)} + \mathbf{e}_i, \quad \mathbf{e}_i \in \mathbb{R}^{512}, \tag{16.53}$$

where $\mathbf{e}_i$ is a learnable vector for position i.

Alternatively, use **2D positional encodings** that encode both spatial coordinates:

$$\mathbf{e}_{(h,w)} = \sin\left(\frac{h}{10000^{0/512}}\right) + \cos\left(\frac{h}{10000^{1/512}}\right) + \dots \tag{16.54}$$

16.6 Proprioceptive State Integration

16.6.1 Joint Angle Encoding

Robot joint angles $\mathbf{q}_t \in \mathbb{R}^D$ (e.g., $D = 7$ for 7-DOF arms) are typically normalized to $[-1, 1]$ or $[-\pi, \pi]$.

Definition 16.6 (Proprioceptive Features). The proprioceptive input includes:

$$\mathbf{q}_t \in \mathbb{R}^D, \quad \dot{\mathbf{q}}_t \in \mathbb{R}^D, \quad \mathbf{f}_t \in \mathbb{R}^6 \quad \text{(if available)}. \tag{16.55}$$

Total proprioceptive dimension: $D_{\text{prop}} = D + D + 6 = 2D + 6$ (or D if only positions).

Encoding as Token:

$$\text{Token}_{\text{prop}} = \text{Linear}_{D_{\text{prop}} \to d_{\text{model}}}(\mathbf{q}_t, \dot{\mathbf{q}}_t, \mathbf{f}_t) \in \mathbb{R}^{d_{\text{model}}}, \tag{16.56}$$

where $d_{\text{model}} = 512$ (Transformer hidden dimension).

This projects the proprioceptive state into the same embedding space as vision tokens, enabling **cross-modal fusion** in the Transformer.

16.6.2 Temporal Context

Beyond the current observation, the Transformer encoder can access **recency information** via:

1. **Stacked frames:** Concatenate T_{stack} consecutive observations: $[\mathbf{o}_{t-T_{\text{stack}}}, \dots, \mathbf{o}_{t-1}, \mathbf{o}_t]$
2. **Temporal embeddings:** Add time-based positional encodings to distinguish past observations
3. **Recurrent hidden states:** Use a recurrent encoder (LSTM/GRU) to maintain temporal context

For ACT, approach (1) is common: stack 2 recent frames for context.

16.7 Action Space: Control Parameterizations

16.7.1 End-Effector vs. Joint Space

Actions can be specified in different spaces:

Joint Space (Workspace):

$$\mathbf{a}_t^{(\text{joint})} = \Delta\mathbf{q}_t \in \mathbb{R}^D, \quad \text{e.g., } \Delta q_i \in [-0.1, 0.1] \text{ rad.} \tag{16.57}$$

Advantages: Direct control, interpretable, avoids IK problems. Disadvantages: Not invariant to robot kinematics changes; difficult to compare across robots.

Task Space (Operational Space):

$$\mathbf{a}_t^{(\text{task})} = (\Delta\mathbf{p}, \Delta\mathbf{R}) \in \mathbb{R}^3 \times SO(3), \tag{16.58}$$

where $\mathbf{p}$ is end-effector position and $\mathbf{R}$ is orientation. Advantages: Task-centric; transferable across robots. Disadvantages: Requires inverse kinematics; may have singularities.

Rotation Representation: Rotations in task space require careful handling:

- **Euler angles** (ϕ, θ, ψ): Simple but suffers gimbal lock
- **Rotation matrices** $\mathbf{R} \in SO(3)$: Overcomplete (9 parameters for 3 DOF)
- **Quaternions** $\mathbf{q} \in \mathbb{H}$: Compact and smooth (4 parameters, constraint $\|\mathbf{q}\| = 1$)
- **6D rotation representation** (Zhou et al., 2019): 6 parameters, Gram-Schmidt orthogonalization. Preferred for learning.

16.7.2 Gripper Commands

Gripper control is typically 1-dimensional:

$$a_{\text{gripper}} \in [-1, 1], \quad \text{where} \quad a_{\text{gripper}} < 0 \Rightarrow \text{open}, \quad a_{\text{gripper}} > 0 \Rightarrow \text{close}. \tag{16.59}$$

Or binary: $a_{\text{gripper}} \in \{0, 1\}$ (open/close).

Some systems use **continuous grasp force**:

$$F_{\text{grasp}} = c_1 \cdot a_{\text{gripper}} + c_2, \quad \text{clamped to } [F_{\min}, F_{\max}]. \tag{16.60}$$

16.8 Distribution Over Trajectories: The Core Challenge

16.8.1 Trajectory Distribution

Rather than a single action distribution $P(\mathbf{a}_t|\mathbf{o}_t)$, we often condition on longer observation histories and predict **action chunks** (trajectory segments):

Definition 16.7 (Trajectory Distribution). Let the action chunk be

$$\mathbf{A}_{t:t+k} = [\mathbf{a}_t, \mathbf{a}_{t+1}, \ldots, \mathbf{a}_{t+k-1}] \in \mathbb{R}^{k \times D}. \tag{16.61}$$

The policy models:

$$P(\mathbf{A}_{t:t+k}|\mathbf{o}_{0:t}) : \text{probability of a coherent sequence of } k \text{ actions.} \tag{16.62}$$

For multimodal distributions, this can be a mixture:

$$P(\mathbf{A}_{t:t+k}|\mathbf{o}_{0:t}) = \sum_{m=1}^{M} w_m \mathcal{N}(\mathbf{A}_{t:t+k}; \boldsymbol{\mu}_m(\mathbf{o}_{0:t}), \Sigma_m(\mathbf{o}_{0:t})), \tag{16.63}$$

where each mode m represents a distinct manipulation strategy (left-grasp, right-grasp, etc.).

16.8.2 Temporal Coherence and Smoothness Constraints

Real robot movements must satisfy **smoothness constraints** due to:

1. **Actuator bandwidth:** Motors have finite speed/acceleration limits
2. **Stability:** Jerky motions cause vibrations, tracking errors
3. **Naturalness:** Smooth motions are easier to learn and replicate

Definition 16.8 (Smoothness Regularization). A common regularizer penalizes **action differences**:

$$\mathcal{R}_{\text{smooth}} = \lambda \sum_{t=0}^{k-2} \|\mathbf{a}_{t+1} - \mathbf{a}_t\|_2^2. \tag{16.64}$$

This encourages the learned policy to predict gradually changing actions rather than abrupt switches. Implicitly, it biases the learned distribution toward **unimodal** components for individual timesteps, while still allowing **multimodality at the trajectory level** (via latent variables).

16.9 Imitation Learning: A Preview

As detailed in Chapter 17, we will focus on learning policies from demonstrations. The core challenge is that a robot's own actions induce a distribution shift from the training data, requiring advanced generative modeling techniques (Chapters 1720) rather than simple supervised regression.

16.10 Key Differences from Supervised Learning: Summary Table

16.11 Bridging to Chapters 1720

The mathematical framework established in this chapterPOMDPs, partial observability, multimodal trajectory distributions, and latent variable modelsdirectly motivates:

- **Chapter 17 (ACT):** Action chunks + CVAE for tractable multimodal distribution learning

Aspect	**Supervised Learning (Vision/NLP)**	**Robot Learning (POM**
Environment	Static dataset	Dynamic, interactive world
Observation Space	Fixed-dimensional (images, text)	Multimodal (vision + prop
Action Space	Discrete (class labels)	Continuous ($\mathbb{R}^D$), bounded
State Observability	Fully observable (labels given)	**Partially observable** (tr
Policy Horizon	Single-step prediction	Multi-step trajectory gener
Multimodality	Single correct answer (usually)	**Multiple valid solutions**
Distribution Learning	Often unimodal (regression)	**Inherently multimodal**
Loss Function	Cross-entropy (discrete) or MSE (regression)	Negative log-likelihood (Ga
Temporal Structure	I.i.d. samples	**Temporal dependencies**
Evaluation	Accuracy/perplexity on test set	Success rate in sim/real; ze

- **Chapter 18 (Diffusion Policy):** Score-based models as an alternative to explicit latent variables
- **Chapter 19 (VLA):** Unified token spaces for vision, language, and action
- **Chapter 20 (Numerical Walkthrough):** Concrete tensor dimensions and gradient computation for ACT

Each method addresses the core challenge: **learning a rich conditional distribution $P(\mathbf{a}_{t:t+k}|\mathbf{o}_{0:t})$ that captures multimodality while maintaining temporal smoothness and physical plausibility.**

Chapter 17

Imitation Learning: Mathematical Formulation and Challenges

In this chapter, we formally define the problem of **Imitation Learning (IL)**, focusing on learning from demonstrations without explicit reward functions. We will explore the baseline approach, **Behavioral Cloning (BC)**, and mathematically derive why it often fails in practice due to the **distribution shift problem**. This motivates the need for advanced generative policies (like ACT and Diffusion Policy) discussed in subsequent chapters.

17.1 Problem Setup: Learning from Demonstrations

Let $\mathcal{M}$ be a Markov Decision Process (MDP) or POMDP with state space $\mathcal{S}$, action space $\mathcal{A}$, and transition dynamics $P(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$.

We are provided with a dataset of **expert demonstrations** $\mathcal{D} = \{\tau_1, \tau_2, \ldots, \tau_N\}$, where each trajectory τ_i consists of state-action pairs generated by an expert policy π^*:

$$\tau = (\mathbf{s}_0, \mathbf{a}_0, \mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T). \tag{17.1}$$

The goal is to learn a policy $\pi_\theta(\mathbf{a}|\mathbf{s})$ that mimics the expert's behavior, such that the performance $J(\pi_\theta)$ approximates $J(\pi^*)$.

17.1.1 The i.i.d. Assumption Violation

A critical distinction from standard supervised learning is that data in IL is **not independent and identically distributed (i.i.d.)**. In standard supervised learning (e.g., image classification), the training data $\{\mathbf{x}_i, y_i\}$ comes from a fixed distribution P_{data}.

In robotics, the policy π_θ interacts with the environment. The actions chosen by π_θ at time t determine the state $\mathbf{s}_{t+1}$ at time $t+1$. Thus, the policy **induces its own state distribution** $P_{\pi_\theta}(\mathbf{s})$.

$$\mathbf{a}_t \sim \pi_\theta(\cdot|\mathbf{s}_t) \implies \mathbf{s}_{t+1} \sim P(\cdot|\mathbf{s}_t, \mathbf{a}_t). \tag{17.2}$$

This feedback loop is the source of the distribution shift problem.

17.2 Behavioral Cloning (BC)

17.2.1 Formulation

Behavioral Cloning treats imitation learning as a standard supervised learning problem. We assume the expert demonstrations are samples from a fixed joint distribution $P_{\text{expert}}(\mathbf{s}, \mathbf{a})$.

The objective is to maximize the log-likelihood of the expert actions under the learned policy:

$$\theta^* = \arg\max_\theta \mathbb{E}_{(\mathbf{s},\mathbf{a})\sim\mathcal{D}}[\log \pi_\theta(\mathbf{a}|\mathbf{s})]. \tag{17.3}$$

For a deterministic policy $\pi_\theta(\mathbf{s}) \approx \mu_\theta(\mathbf{s})$ with Gaussian noise, this is equivalent to minimizing the Mean Squared Error (MSE):

$$\mathcal{L}_{\text{BC}}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{s},\mathbf{a})\in\mathcal{D}} \|\mathbf{a} - \mu_\theta(\mathbf{s})\|_2^2. \tag{17.4}$$

17.2.2 Intuition

BC asks: "Given that I am in state $\mathbf{s}$ (which is in the expert's support), what action would the expert take?" It ignores the consequences of the action on future states. Ideally, if the training error is low ($\epsilon \approx 0$), the policy should perform well. However, in sequential tasks, small errors accumulate.

17.3 The Covariate Shift Problem

Why does BC often fail in practice? The fundamental issue is **Covariate Shift** (or Distribution Shift).

17.3.1 Distribution Mismatch

- **Training Distribution:** $P_{\text{train}}(\mathbf{s}) = P_{\pi^*}(\mathbf{s})$ (states visited by the expert).
- **Test Distribution:** $P_{\text{test}}(\mathbf{s}) = P_{\pi_\theta}(\mathbf{s})$ (states visited by the learned policy).

Since $\pi_\theta \neq \pi^*$ (learning is never perfect), $P_{\pi_\theta} \neq P_{\pi^*}$. The policy inevitably drifts into states it has never seen during training. In these out-of-distribution (OOD) states, the policy behavior is undefined (or arbitrary), leading to further errors.

17.3.2 Error Accumulation Analysis ($O(T^2)$ Error)

Ross et al. (2011) provided a theoretical bound on the error accumulation.

Let ϵ be the probability that π_θ chooses a different action than π^* in any state drawn from P_{π^*}:

$$\mathbb{E}_{\mathbf{s}\sim P_{\pi^*}}[\mathbb{I}(\pi_\theta(\mathbf{s}) \neq \pi^*(\mathbf{s}))] \leq \epsilon. \tag{17.5}$$

Over a horizon of T steps:

1. At $t = 0$, the robot starts in $\mathbf{s}_0$ (Expert state). Error probability is ϵ.

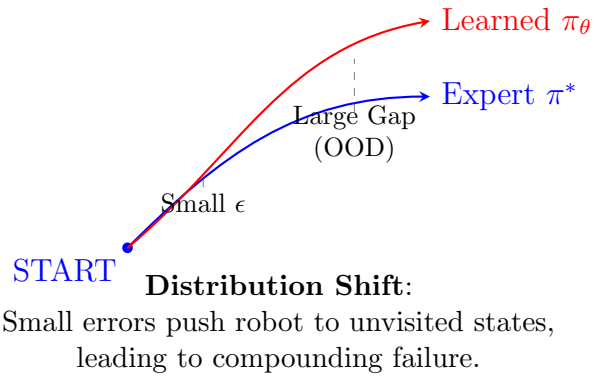

Figure 17.1: Visualizing Compounding Errors. A small error ϵ at $t = 0$ puts the robot in a slightly different state. Since the policy was trained on the blue trajectory distributions, this new state is out-of-distribution (OOD), leading to larger errors and quadratic divergence $O(T^2)$.

2. Once an error occurs, the robot enters a state $\mathbf{s}'$ OOD.

3. In OOD states, the error bound ϵ (based on expert distribution) no longer applies. The error could be as large as 1 (complete failure).

The total expected error (loss) over T steps accumulates as **$O(T^2)$** in the worst case (unlike $O(T)$ in environments where the policy can recover).

$$J(\pi_\theta) \leq J(\pi^*) + T^2\epsilon. \tag{17.6}$$

This quadratic scaling implies that for long-horizon tasks, pure BC is extremely fragile. A small perturbation at the beginning can lead to a completely failed trajectory.

17.4 Addressing the Shift

To overcome the fragility of BC, several approaches exist:

17.4.1 Online Approaches (e.g., DAgger)

Dataset Aggregation (DAgger) iteratively collects data *from the robot's own induced distribution*. 1. Train π_θ on $\mathcal{D}$. 2. Run π_θ to collect new observations $\mathcal{D}_\pi$. 3. Ask the expert to label these new observations (provide correct actions). 4. $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_\pi$. Repeat. This forces the policy to learn how to **recover** from its own mistakes. However, it requires an interactive expert (human-in-the-loop), which is often impractical.

17.4.2 Explicit Modal Modeling

If the expert is multimodal (e.g., sometimes goes left, sometimes right), a unimodal BC policy (MSE loss) learns the **average**, which might be an invalid action (going straight into an obstacle). Explicit mixture models (e.g., Mixture Density Networks, GMMs) can model $P(\mathbf{a}|\mathbf{s})$ as a mixture of Gaussians.

17.4.3 Implicit Generative Modeling (ACT, Diffusion)

Modern imitation learning focuses on **high-fidelity distribution matching**. Instead of just minimizing MSE, we use powerful generative models to capture the full conditional distribution $P(\mathbf{a}|\mathbf{s})$.

- **ACT (Chapter 17):** Uses a CVAE to model the distribution via latent variables.
- **Diffusion Policy (Chapter 19):** Uses diffusion processes to gradually denoise actions.
- **Action Chunking:** Predicting sequences of actions helps maintain temporal consistency and reduces the effective horizon T, mitigating the T^2 drift.

This shift from simple regression to generative modeling is the key theme of the following chapters.

Chapter 18

Generative Modeling Foundations: VAE and CVAE

Before diving into specific robotic policies like ACT, we must establish the mathematical framework for modeling complex, multimodal distributions. This chapter introduces Variational Autoencoders (VAEs) and their conditional variant (CVAE), which provide the theoretical basis for learning latent variable models.

18.1 Latent Variable Models

In many real-world tasks, the observable data $\mathbf{x}$ (e.g., robot trajectories) is complex and multimodal. We assume this complexity arises from unobserved *latent variables* $\mathbf{z}$ (e.g., the "strategy" or "intention" of the motion).

The generative process is modeled as:

1. Sample a latent code from a prior distribution: $\mathbf{z} \sim P(\mathbf{z})$.
2. Generate data from a conditional distribution: $\mathbf{x} \sim P_\theta(\mathbf{x}|\mathbf{z})$.

Here, P_θ is typically parameterized by a neural network (the *decoder*). The marginal likelihood of the data is:

$$P_\theta(\mathbf{x}) = \int P_\theta(\mathbf{x}|\mathbf{z})P(\mathbf{z})d\mathbf{z}. \tag{18.1}$$

Computing this integral directly is intractable because for any given $\mathbf{x}$, we do not know which $\mathbf{z}$ produced it.

18.2 Variational Autoencoder (VAE)

18.2.1 Evidence Lower Bound (ELBO)

To solve the intractability, we introduce an approximate posterior $Q_\phi(\mathbf{z}|\mathbf{x})$, parameterized by another neural network (the *encoder*). We want $Q_\phi(\mathbf{z}|\mathbf{x})$ to be close to the true posterior $P(\mathbf{z}|\mathbf{x})$.

The objective is to maximize the log-likelihood of the data, $\log P_\theta(\mathbf{x})$. Using Jensen's inequality, we can derive a tractable lower bound:

$$\log P_\theta(\mathbf{x}) = \log \int P_\theta(\mathbf{x}|\mathbf{z}) \frac{Q_\phi(\mathbf{z}|\mathbf{x})}{Q_\phi(\mathbf{z}|\mathbf{x})} P(\mathbf{z}) d\mathbf{z} \tag{18.2}$$

$$= \log \mathbb{E}_{\mathbf{z}\sim Q_\phi}\left[\frac{P_\theta(\mathbf{x}|\mathbf{z})P(\mathbf{z})}{Q_\phi(\mathbf{z}|\mathbf{x})}\right] \tag{18.3}$$

$$\geq \mathbb{E}_{\mathbf{z}\sim Q_\phi}\left[\log \frac{P_\theta(\mathbf{x}|\mathbf{z})P(\mathbf{z})}{Q_\phi(\mathbf{z}|\mathbf{x})}\right] \quad \text{(Jensen's Inequality)} \tag{18.4}$$

$$= \mathbb{E}_{\mathbf{z}\sim Q_\phi}[\log P_\theta(\mathbf{x}|\mathbf{z})] - \mathrm{KL}(Q_\phi(\mathbf{z}|\mathbf{x})||P(\mathbf{z})). \tag{18.5}$$

The right-hand side of (18.5) is the **Evidence Lower Bound (ELBO)**. Maximizing the ELBO is equivalent to maximizing the log-likelihood while minimizing the divergence between the approximate and true posteriors.

The loss function to minimize is typically the negative ELBO:

$$\mathcal{L}_{\text{VAE}} = \underbrace{-\mathbb{E}_{\mathbf{z}\sim Q_\phi}[\log P_\theta(\mathbf{x}|\mathbf{z})]}_{\text{Reconstruction Loss}} + \beta \cdot \underbrace{\mathrm{KL}(Q_\phi(\mathbf{z}|\mathbf{x})||P(\mathbf{z}))}_{\text{Regularization Loss}}, \tag{18.6}$$

where β is a hyperparameter (often annealed during training) to balance reconstruction quality and latent space regularization.

18.2.2 Reparameterization Trick

To train the encoder Q_ϕ using gradient descent, we need to differentiate through the sampling operation $\mathbf{z} \sim Q_\phi(\mathbf{z}|\mathbf{x})$. If we sample directly, the gradient flow is broken.

The **Reparameterization Trick** expresses the random variable $\mathbf{z}$ as a deterministic function of the encoder outputs and an independent noise source. Assuming a Gaussian posterior $Q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}_\phi(\mathbf{x}), \mathrm{diag}(\boldsymbol{\sigma}^2_\phi(\mathbf{x})))$:

$$\mathbf{z} = \boldsymbol{\mu}_\phi(\mathbf{x}) + \boldsymbol{\sigma}_\phi(\mathbf{x}) \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{18.7}$$

where $\odot$ denotes element-wise multiplication. Gradients can now flow back to $\boldsymbol{\mu}_\phi$ and $\boldsymbol{\sigma}_\phi$:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\sigma}_\phi} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}} \odot \boldsymbol{\epsilon}. \tag{18.8}$$

18.3 Conditional VAE (CVAE)

In robotics, we rarely generate data from scratch. Instead, we generate an action sequence $\mathbf{a}$ *conditioned* on the current observation $\mathbf{o}$. The CVAE extends the VAE framework by conditioning both the encoder and decoder on $\mathbf{o}$.

18.3.1 Architecture Modification

- **Latent Prior:** Conditioned on observation, $P(\mathbf{z}|\mathbf{o})$. Often simplified to $\mathcal{N}(\mathbf{0}, \mathbf{I})$ independent of $\mathbf{o}$.
- **Encoder (Training only):** Approximates $P(\mathbf{z}|\mathbf{a}, \mathbf{o})$. Input is the pair $(\mathbf{a}, \mathbf{o})$.
- **Decoder (Policy):** Models $P(\mathbf{a}|\mathbf{z}, \mathbf{o})$. Input is $(\mathbf{z}, \mathbf{o})$.

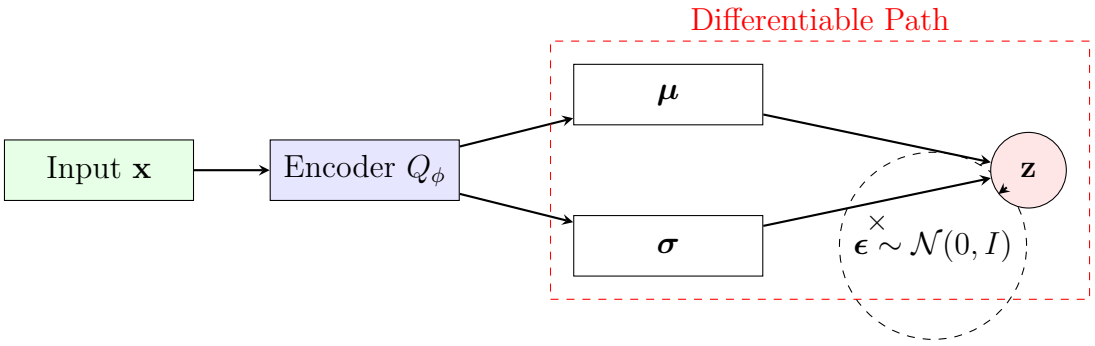

Figure 18.1: The Reparameterization Trick. Instead of sampling $\mathbf{z}$ directly (which blocks gradients), we sample noise $\boldsymbol{\epsilon}$ and transform it deterministically: $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$. This allows gradients to backpropagate through $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$.

18.3.2 CVAE Loss Function

The CVAE objective maximizes the conditional log-likelihood $\log P_\theta(\mathbf{a}|\mathbf{o})$. The resulting loss is:

$$\mathcal{L}_{\text{CVAE}} = \underbrace{-\mathbb{E}_{\mathbf{z}\sim Q_\phi(\cdot|\mathbf{a},\mathbf{o})}[\log P_\theta(\mathbf{a}|\mathbf{z},\mathbf{o})]}_{\text{Reconstruction}} + \beta \cdot \underbrace{\text{KL}(Q_\phi(\mathbf{z}|\mathbf{a},\mathbf{o})||P(\mathbf{z}|\mathbf{o}))}_{\text{Regularization}}. \tag{18.9}$$

18.3.3 Inference (Test Time)

During inference (e.g., robot deployment), we do not have the ground-truth action $\mathbf{a}$.

1. Sample $\mathbf{z}$ from the prior: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.
2. Generate action using the decoder: $\hat{\mathbf{a}} = \text{Decoder}_\theta(\mathbf{z}, \mathbf{o})$.

This formulation allows the model to capture *multimodal* action distributions. Different samples of $\mathbf{z}$ will result in different valid trajectories (modes), solving the averaging problem inherent in deterministic regression.

Chapter 19

Action Chunking Transformers (ACT)

19.1 Overview and Core Motivation

Action Chunking Transformers (ACT) is a method for learning multimodal robot policies from demonstrations, proposed by Zhao et al. (2023) for the ACT paper and related work. It combines three key ingredients:

1. **Action Chunking**: Predict sequences of k actions (chunks) rather than single actions
2. **Conditional Variational Autoencoder (CVAE)**: Model the multimodal distribution of action chunks via latent variables
3. **Transformer Architecture**: Use attention mechanisms to fuse visual and proprioceptive information and condition on observation history

The core insight is that many robot manipulation tasks are **multimodal** (multiple valid ways to accomplish the goal), and naive averaging of demonstrations fails catastrophically. ACT uses a latent code $\mathbf{z}$ to **select among modes**, enabling the model to generate coherent, physically plausible action sequences.

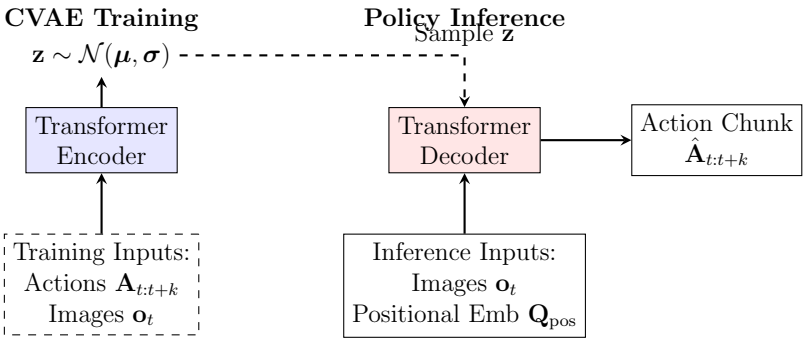

Figure 19.1: ACT Architecture. Left: During training, the CVAE encoder compresses future actions and observations into a latent $\mathbf{z}$. Right: During inference, the decoder uses sampled $\mathbf{z}$ and current observations to generate the action chunk.

19.2 Action Chunking: Problem Formulation

19.2.1 Why Chunking?

Definition 17.1 (Action Chunk). At timestep t, instead of predicting a single action $\mathbf{a}_t$, the policy predicts a sequence of k future actions:

$$\mathbf{A}_t = [\hat{\mathbf{a}}_t, \hat{\mathbf{a}}_{t+1}, \ldots, \hat{\mathbf{a}}_{t+k-1}] \in \mathbb{R}^{k \times D}, \tag{19.1}$$

where k is the chunk size (typical: $k \in [10, 100]$) and D is the action dimension.

Motivation 1: Temporal Coherence A single latent code $\mathbf{z}$ now determines an entire trajectory segment, not just the next action. This enforces **temporal smoothness**the trajectory is coherent and physically realistic, rather than oscillating between modes on a per-timestep basis.

Motivation 2: Multimodality at Trajectory Level The chunk-level multimodality is more natural. Consider:

- **Mode 1:** "Approach the cup from the left, then grasp"a coherent 2-second sequence
- **Mode 2:** "Approach from the right, then grasp"another coherent sequence

Predicting these at the chunk level avoids averaging modes.

Motivation 3: Reduced Inference Latency Control loops typically run at 2050 Hz. If the policy outputs 100 actions at once, inference happens at 0.5 Hz (every 2 seconds), amortizing the computational cost of a forward pass.

19.2.2 Temporal Overlap and Execution

At inference time, the policy makes predictions at **every timestep**, leading to overlapping chunks:

- **t=0**: Predict $\mathbf{A}_0 = [\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_{99}]$
- **t=1**: Predict $\mathbf{A}_1 = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_{100}]$
- **t=2**: Predict $\mathbf{A}_2 = [\mathbf{a}_2, \mathbf{a}_3, \ldots, \mathbf{a}_{101}]$

Temporal Ensembling (discussed in 17.5) resolves conflicts by averaging overlapping predictions with exponential decay weights, favoring recent predictions.

19.3 Conditional Variational Autoencoder (CVAE)

As detailed in Chapter 18, we use a Conditional Variational Autoencoder (CVAE) to model the multimodal distribution of action chunks:

$$P_\theta(\mathbf{A}|\mathbf{o}) = \int P_\theta(\mathbf{A}|\mathbf{z}, \mathbf{o}) P(\mathbf{z}) d\mathbf{z}. \tag{19.2}$$

By conditioning on a latent variable $\mathbf{z}$, the policy can represent multiple valid action sequences (modes). The training objective is the **Evidence Lower Bound (ELBO)** derived in Equation (18.5).

19.4 Encoder: Inferring the Posterior

19.4.1 Architecture: Transformer Encoder (BERT-like)

The encoder processes **both** the true action chunk and the current observation to infer latent parameters.

Definition 17.3 (Encoder Architecture). The encoder is a Transformer encoder (bidirectional self-attention) that:

1. Tokenizes and embeds the action chunk
2. Concatenates observation features (visual + proprioceptive)
3. Passes through L Transformer blocks
4. Pools the representation to output $\boldsymbol{\mu}$ and $\log \boldsymbol{\sigma}^2$

Detailed Forward Pass:

Step 1: Action Embedding Input: Action chunk $\mathbf{A} \in \mathbb{R}^{B\times k\times D}$ (batch size B, chunk length k, action dim D)

Linear projection to embedding space:

$$\mathbf{A}_{\text{embed}} = \mathbf{A}\mathbf{W}_{\text{act}} + \mathbf{b}_{\text{act}}, \quad \mathbf{A}_{\text{embed}} \in \mathbb{R}^{B\times k\times d_{\text{model}}}, \tag{19.3}$$

where $\mathbf{W}_{\text{act}} \in \mathbb{R}^{D\times d_{\text{model}}}$ and d_{model} is the Transformer hidden dimension (typically 512).

Add positional embeddings to distinguish temporal positions:

$$\mathbf{A}^{(i)}_{\text{embed}} \leftarrow \mathbf{A}^{(i)}_{\text{embed}} + \mathbf{e}^{\text{pos}}_i, \quad \mathbf{e}^{\text{pos}}_i \in \mathbb{R}^{d_{\text{model}}}. \tag{19.4}$$

Step 2: Observation Encoding Observation features $\mathbf{o}$ are processed to produce:

- Vision tokens: $\text{Tokens}_{\text{vis}} \in \mathbb{R}^{B\times N_{\text{vis}}\times d_{\text{model}}}$ (e.g., 18 tokens from ResNet-18)
- Proprioception: $\text{Token}_{\text{prop}} \in \mathbb{R}^{B\times 1\times d_{\text{model}}}$
- CLS token: $\text{Token}_{[\text{CLS}]} \in \mathbb{R}^{B\times 1\times d_{\text{model}}}$ (learnable, for pooling)

Step 3: Token Concatenation

$$\mathbf{H}_{\text{input}} = [\text{Token}_{[\text{CLS}]}; \text{Token}_{\text{prop}}; \text{Tokens}_{\text{vis}}; \mathbf{A}_{\text{embed}}] \in \mathbb{R}^{B\times(1+1+N_{\text{vis}}+k)\times d_{\text{model}}}. \tag{19.5}$$

Example dimensions:

$$\mathbf{H}_{\text{input}} \in \mathbb{R}^{8\times 128\times 512} \quad (\text{for } B=8, k=100, N_{\text{vis}}=18, d_{\text{model}}=512). \tag{19.6}$$

Step 4: Transformer Encoder Pass through L self-attention blocks:

$$\mathbf{H}^{(0)} = \mathbf{H}_{\text{input}} \tag{19.7}$$

$$\mathbf{H}^{(\ell)} = \text{TransformerBlock}_\ell(\mathbf{H}^{(\ell-1)}), \quad \ell = 1,\dots,L \tag{19.8}$$

$$\mathbf{H}_{\text{output}} = \mathbf{H}^{(L)} \in \mathbb{R}^{B\times 128\times 512}. \tag{19.9}$$

Step 5: Pooling to Latent Parameters Extract the [CLS] token output (first position):

$$\mathbf{h}_{\text{cls}} = \mathbf{H}_{\text{output}}[:, 0, :] \in \mathbb{R}^{B\times d_{\text{model}}} = \mathbb{R}^{B\times 512}. \tag{19.10}$$

Or use mean pooling over all tokens:

$$\mathbf{h}_{\text{pool}} = \frac{1}{128}\sum_{i=0}^{127}\mathbf{H}_{\text{output}}[:, i, :] \in \mathbb{R}^{B\times 512}. \tag{19.11}$$

Step 6: Output Linear Layers

$$\boldsymbol{\mu}_\phi = \text{Linear}_{d_{\text{model}}\to d_z}(\mathbf{h}_{\text{pool}}) \in \mathbb{R}^{B\times d_z} = \mathbb{R}^{B\times 32}, \tag{19.12}$$

$$\log \boldsymbol{\sigma}^2_\phi = \text{Linear}_{d_{\text{model}}\to d_z}(\mathbf{h}_{\text{pool}}) \in \mathbb{R}^{B\times 32}. \tag{19.13}$$

19.4.2 Self-Attention Mechanics in the Encoder

Query, Key, Value Projections:

$$\mathbf{Q} = \mathbf{H}^{(\ell-1)}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{H}^{(\ell-1)}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{H}^{(\ell-1)}\mathbf{W}_V, \tag{19.14}$$

where $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d_{\text{model}}\times d_k}$ (for single-head; multi-head splits).

Attention Map:

$$\text{Attn} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right) \in \mathbb{R}^{B\times 128\times 128}. \tag{19.15}$$

This allows each action token to "look at" all other actions, proprioception, and visionenabling the encoder to capture **global structure** of the action sequence.

Attention Output:

$$\mathbf{H}^{(\ell)} = \text{Attn}\cdot\mathbf{V} + \text{FFN}(\cdot), \tag{19.16}$$

where FFN is a feed-forward network (position-wise MLP).

19.5 Decoder: Generating Action Chunks from Latent Code

19.5.1 Architecture: Transformer Decoder with Cross-Attention

At **inference time** (when ground-truth actions are unavailable), the decoder generates actions using the latent code $\mathbf{z}$ and observation features as conditioning.

Definition 17.4 (Decoder Architecture). The decoder is a Transformer decoder with:

- **Self-attention** on the action sequence being generated
- **Cross-attention** to observation features (visual + latent code)
- Position-wise feed-forward networks

Detailed Forward Pass (Inference):

Step 1: Latent Code Processing Sample from the prior (at inference):

$$\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \quad \mathbf{z} \in \mathbb{R}^{B\times d_z} = \mathbb{R}^{B\times 32}. \tag{19.17}$$

Project to embedding space:

$$\mathbf{z}_{\text{embed}} = \text{Linear}_{d_z \to d_{\text{model}}}(\mathbf{z}) \in \mathbb{R}^{B \times d_{\text{model}}} = \mathbb{R}^{B \times 512}. \tag{19.18}$$

Expand to match action sequence length (by tiling or concatenation):

$$\mathbf{z}_{\text{tiled}} = \text{Repeat}(\mathbf{z}_{\text{embed}}, k) \in \mathbb{R}^{B \times k \times d_{\text{model}}} = \mathbb{R}^{B \times 100 \times 512}. \tag{19.19}$$

Step 2: Query (Action Positions) Learnable position embeddings for the k actions:

$$\mathbf{Q}_{\text{pos}} \in \mathbb{R}^{B \times k \times d_{\text{model}}} = \mathbb{R}^{B \times 100 \times 512}. \tag{19.20}$$

These are shared across all samples in the batch.

Step 3: Key/Value from Observation Concatenate vision tokens and latent code:

$$\mathbf{K}_{\text{cond}} = \text{Concat}([\text{Tokens}_{\text{vis}}, \mathbf{z}_{\text{embed}}]) \in \mathbb{R}^{B \times (N_{\text{vis}}+1) \times d_{\text{model}}} = \mathbb{R}^{B \times 19 \times 512}. \tag{19.21}$$

$$\mathbf{V}_{\text{cond}} = \mathbf{K}_{\text{cond}} \quad \text{(same, or learned projection)}. \tag{19.22}$$

Step 4: Self-Attention (over action sequence) Allows each action position to "see" previous/future actions (causal masking if autoregressive):

$$\text{Attn}_{\text{self}} = \text{softmax}\left(\frac{\mathbf{Q}_{\text{pos}}\mathbf{Q}_{\text{pos}}^\top}{\sqrt{d_k}} + \text{Mask}\right)\mathbf{V}_{\text{pos}}. \tag{19.23}$$

For typical ACT, bidirectional attention is used (not causal), since we can see the entire chunk:

$$\text{Attn}_{\text{self}} = \text{softmax}\left(\frac{\mathbf{Q}_{\text{pos}}\mathbf{Q}_{\text{pos}}^\top}{\sqrt{d_k}}\right)\mathbf{Q}_{\text{pos}}. \tag{19.24}$$

Step 5: Cross-Attention (to observation) Each action attends to observation features:

$$\text{Attn}_{\text{cross}} = \text{softmax}\left(\frac{\mathbf{Q}_{\text{pos}}\mathbf{K}_{\text{cond}}^\top}{\sqrt{d_k}}\right)\mathbf{V}_{\text{cond}} \in \mathbb{R}^{B \times k \times d_{\text{model}}}. \tag{19.25}$$

Dimension check:

$$\frac{\mathbf{Q}_{\text{pos}}\mathbf{K}_{\text{cond}}^\top}{\sqrt{d_k}} \in \mathbb{R}^{B \times k \times (N_{\text{vis}}+1)} = \mathbb{R}^{B \times 100 \times 19}. \tag{19.26}$$

After softmax and multiplication by $\mathbf{V}_{\text{cond}} \in \mathbb{R}^{B \times 19 \times 512}$:

$$\text{Attn}_{\text{cross}} \in \mathbb{R}^{B \times 100 \times 512}. \tag{19.27}$$

Step 6: Output Projection

$$\hat{\mathbf{A}} = \text{MLP}_{\text{out}}(\text{Attn}_{\text{cross}}) \in \mathbb{R}^{B \times k \times D} = \mathbb{R}^{B \times 100 \times 7}. \tag{19.28}$$

The MLP projects from $d_{\text{model}} = 512$ to $D = 7$ (action dimension).

19.5.2 Multi-Head Attention in the Decoder

In practice, Transformer decoders use **multi-head attention** to enable the model to attend to different aspects of the input simultaneously.

Definition 17.5 (Multi-Head Attention). For H heads:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H)\mathbf{W}_O, \tag{19.29}$$

where each head computes:

$$\text{head}_h = \text{softmax}\left(\frac{\mathbf{Q}_h \mathbf{K}_h^\top}{\sqrt{d_k}}\right)\mathbf{V}_h, \tag{19.30}$$

$$\mathbf{Q}_h = \mathbf{Q}\mathbf{W}_Q^{(h)}, \quad d_k = \frac{d_{\text{model}}}{H}. \tag{19.31}$$

For $H = 8$ heads and $d_{\text{model}} = 512$: $d_k = 64$.

Benefit: Different heads learn different attention patterns:

- Head 1 might attend to latent code (global context)
- Head 2 might attend to left camera (left-side obstacles)
- Head 3 might attend to proprioception (self-collision avoidance)

19.6 Gradient Computation

19.6.1 Reparameterization Trick

As explained in Chapter 18, the reparameterization trick allows gradients to flow through the stochastic sampling layer:

$$\mathbf{z} = \boldsymbol{\mu}_\phi(\mathbf{A}, \mathbf{o}) + \boldsymbol{\sigma}_\phi(\mathbf{A}, \mathbf{o}) \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}). \tag{19.32}$$

This makes $\mathbf{z}$ a deterministic function of the encoder outputs and random noise, enabling standard backpropagation.

19.6.2 Parameter Update (SGD/Adam)

Parameter groups:

1. **Encoder parameters** ϕ: Updated via $\mathcal{L}_{\text{rec}} + \beta\mathcal{L}_{\text{KL}}$
2. **Decoder parameters** θ: Updated via $\mathcal{L}_{\text{rec}}$ only (KL doesn't depend on θ)
3. **Vision backbone**: Fixed (pre-trained) or fine-tuned

Update rule (Adam):

$$\phi_{t+1} = \phi_t - \alpha_{\text{enc}} m_t^{(\phi)} / (\sqrt{v_t^{(\phi)}} + \epsilon), \tag{19.33}$$

$$\theta_{t+1} = \theta_t - \alpha_{\text{dec}} m_t^{(\theta)} / (\sqrt{v_t^{(\theta)}} + \epsilon), \tag{19.34}$$

where $\alpha_{\text{enc}}, \alpha_{\text{dec}}$ are learning rates (potentially different).

19.7 Temporal Ensembling and Smooth Action Execution

19.7.1 Multiple Predictions for the Same Action

At inference, the policy produces predictions at high frequency. Consider a chunk size $k = 100$ at 50 Hz control:

- $\mathbf{t = 0}$ (0.0 s): Predict $\hat{\mathbf{a}}_0, \hat{\mathbf{a}}_1, \ldots, \hat{\mathbf{a}}_{99}$
- $\mathbf{t = 1}$ (0.02 s): Predict $\hat{\mathbf{a}}_1, \hat{\mathbf{a}}_2, \ldots, \hat{\mathbf{a}}_{100}$
- $\mathbf{t = 2}$ (0.04 s): Predict $\hat{\mathbf{a}}_2, \hat{\mathbf{a}}_3, \ldots, \hat{\mathbf{a}}_{101}$

The action $\hat{\mathbf{a}}_1$ is predicted twice (by chunks from $t = 0$ and $t = 1$), with potentially different values.

19.7.2 Exponential Moving Average (EMA) Ensembling

Definition 17.7 (Temporal Ensembling). For each action index i, collect all predictions and form a weighted average:

$$\mathbf{a}_i^{\text{final}} = \frac{\sum_{\tau=0}^{k-1} w_\tau \hat{\mathbf{a}}_i^{(\text{chunk at } i-\tau)}}{\sum_{\tau=0}^{k-1} w_\tau}, \tag{19.35}$$

where the weights decay exponentially:

$$w_\tau = \exp(-\lambda\tau), \quad \lambda > 0 \text{ (decay constant, e.g., } \lambda = 0.1). \tag{19.36}$$

Interpretation:

- $\tau = 0$: Most recent prediction (weight $w_0 = 1$)
- $\tau = 1$: Older prediction (weight $w_1 = e^{-0.1} \approx 0.905$)
- $\tau = 99$: Very old prediction (weight $w_{99} \approx 0$)

Result: Recent predictions dominate, ensuring the executed action is close to the most current policy estimate, while benefiting from averaging away noise.

19.7.3 Smoothness Guarantee

Temporal ensembling provides **implicit smoothness regularization**. The executed actions are:

$$\mathbf{a}_i^{\text{final}} = \frac{1}{Z} \sum_\tau w_\tau \hat{\mathbf{a}}_i^{(\tau)}, \quad Z = \sum_\tau w_\tau. \tag{19.37}$$

If the policy predicts $\hat{\mathbf{a}}_i \approx \hat{\mathbf{a}}_{i+1}$ (smooth predictions), then:

$$\mathbf{a}_{i+1}^{\text{final}} - \mathbf{a}_i^{\text{final}} \approx \text{small}. \tag{19.38}$$

Even if individual chunks are slightly jerky, ensemble averaging smooths out discontinuities.

19.8 Handling Multimodality: How ACT Solves the Averaging Problem

19.8.1 Revisiting the Problem

Recall from Chapter 16: averaging demonstrations from multiple modes produces **invalid trajectories**.

Example: Cup grasping with 3 modes

Mode	Approach	Success
Left	$x = -5, z = -2, \theta = 30°$	Grasps
Right	$x = +5, z = -2, \theta = -30°$	Grasps
Naive Mean	$x = 0, z = -2, \theta = 0°$	Crashes

Why ACT Works:

The CVAE introduces a **discrete choice variable** (the latent code) that selects among modes. During training:

$$\mathbf{z}^{(1)} \to \text{Mode 1 (Left)}, \quad \mathbf{z}^{(2)} \to \text{Mode 2 (Right)}, \quad \ldots \tag{19.39}$$

The encoder learns to infer which mode the demonstration follows, and the decoder learns to reconstruct each mode faithfully.

19.8.2 Mathematical Characterization

Definition 17.8 (Mixture Interpretation). The learned conditional distribution can be written as an (implicit) mixture:

$$P_\theta(\mathbf{A}|\mathbf{o}) = \int P_\theta(\mathbf{A}|\mathbf{z}, \mathbf{o}) P(\mathbf{z}) d\mathbf{z} \tag{19.40}$$

$$\approx \sum_{m=1}^{M} P(\mathbf{z} = \mathbf{z}_m|\mathbf{o}) \cdot \mathcal{N}(\mathbf{A}; \boldsymbol{\mu}_m(\mathbf{o}), \Sigma_m), \tag{19.41}$$

where $\mathbf{z}_m$ are "mode centers" (learned implicitly) and $\mathcal{N}(\mathbf{A}; \boldsymbol{\mu}_m, \Sigma_m)$ is the reconstructed distribution for mode m.

Key Property: Unlike explicit mixture models, the mode structure is **implicit** in the continuous latent space. This allows:

1. **Smooth interpolation** between modes (latent interpolation)
2. **Arbitrary number of modes** (continuous latent space, not discrete clusters)
3. **Generalization** (the decoder can reconstruct unseen latent codes)

19.8.3 Inference: Mode Selection

At inference time, we sample $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ (prior), which implicitly selects a mode:

- Different samples $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \ldots$ typically produce different action sequences

- Multiple samples from the prior allow us to generate diverse rollouts

Concrete Example (Bimanual Task):
During training, the encoder might learn:

- Demonstrations where both arms approach a cup from the left → inferred posterior concentrates around $\mathbf{z}_L$
- Demonstrations where both arms approach from the right → inferred posterior concentrates around $\mathbf{z}_R$

At inference, sampling different latent codes produces different coordinated arm strategies:

$$\mathbf{z}_L \sim \mathcal{N}(\mu_L, \sigma_L) \Rightarrow \text{Left-side coordination} \tag{19.42}$$

$$\mathbf{z}_R \sim \mathcal{N}(\mu_R, \sigma_R) \Rightarrow \text{Right-side coordination} \tag{19.43}$$

19.9 Practical Considerations: Hyperparameter Selection

19.9.1 Latent Dimension d_z

Choice: $d_z \in [16, 64]$ depending on task complexity.

Task	**Typical d_z**	**Reasoning**
Reaching (3-DOF)	816	Few modes (left/right)
Grasping (6-DOF)	1632	Multiple grasp points
Complex manipulation	3264	Many coordination strategies

Too small ($d_z < 8$): Cannot capture task multimodality; poor performance. **Too large** ($d_z > 256$): Overfitting; posterior becomes difficult to regularize.

19.9.2 Chunk Size k

Choice: $k \in [10, 150]$ depending on task horizon and control frequency.

Setting	k	**Duration**	**Benefit**
Fast control (100 Hz)	1020	0.10.2 s	Low latency
Standard (50 Hz)	50100	12 s	Balance
Slow manipulation	100200	1020 s	Long horizons

Too small ($k < 5$): Loses temporal structure; chunk-level multimodality is weak. **Too large** ($k > 200$): Expensive inference; harder to learn temporal coherence.

19.9.3 β (KL Weight)

Recommendation: Start with $\beta = 0.1$ and **anneal** to $\beta_{\text{target}} \in [1, 10]$ over the first 2030% of training.

Annealing Schedule:

$$\beta(e) = \min\left(1, \frac{e}{e_{\text{warmup}}}\right) \cdot \beta_{\text{target}}, \quad e \in \{1, 2, \ldots, E\}. \tag{19.44}$$

Example: $e_{\text{warmup}} = 20, \beta_{\text{target}} = 10, E = 100$:

- Epoch 120: β increases linearly from 0.1 to 10
- Epoch 21100: $\beta = 10$ (constant)

19.9.4 Vision Backbone Freezing vs. Fine-tuning

Frozen backbone (default):

- Use pre-trained ResNet-18 (ImageNet weights)
- Faster training, stable gradients
- Best for small datasets (< 1000 demonstrations)

Fine-tuned backbone:

- Unfreeze backbone weights (typically from epoch 1020)
- Allows adaptation to robot-specific viewpoints
- Better for large datasets (> 10000 demonstrations)

Learning rate strategy (if fine-tuning):

$$\alpha_{\text{backbone}} = 0.1 \times \alpha_{\text{decoder}}. \tag{19.45}$$

19.10 Training Procedure: Full Algorithm

19.10.1 Pseudocode

```
Algorithm 1: ACT Training

Input: Dataset D = {(o_i, A_i*)}, hyperparameters (beta, d_z, k, alpha_enc, alpha_dec

Initialize:
  - Encoder phi (random or pre-trained)
  - Decoder theta (random)
  - Vision backbone Phi (pre-trained, frozen)
  - Optimizer: Adam(lr_enc=alpha_enc, lr_dec=alpha_dec)

for epoch e = 1 to E:
  for batch B = {(o_j, A_j*)} in D:
```

```
    # Forward pass (encoder)
    z_embed <- Phi(o_j)  # Extract features
    h_cls <- Transformer_encoder(z_embed, A_j*, phi)  # Bidirectional attention
    mu_phi, log_sigma^2_phi <- Linear(h_cls)  # Posterior parameters

    # Reparameterization
    epsilon ~ N(0, I)
    z <- mu_phi + sqrt(exp(log_sigma^2_phi)) * epsilon

    # Forward pass (decoder)
    A_pred <- Transformer_decoder(z, z_embed, theta)  # Cross-attention

    # Compute losses
    L_rec <- ||A_pred - A_j*||^2_2
    L_KL <- (1/2) Sum_j (mu^2_phi,j + sigma^2_phi,j - log sigma^2_phi,j - 1)
    beta(e) <- min(1, e / e_warmup) * beta_target

    L_total <- L_rec + beta(e) * L_KL

    # Backward pass
    grad_theta L_total, grad_phi L_total

    # Update parameters
    theta <- Adam_update(theta, grad_theta L_total, alpha_dec)
    phi <- Adam_update(phi, grad_phi L_total, alpha_enc)

return theta, phi
```

19.10.2 Computational Complexity

Per-batch forward pass:

- Vision backbone: $\approx 1.8 \times 10^9$ MACs (ResNet-18)
- Encoder: $2 \times (k + N_{\text{vis}}) \times d_{\text{model}}^2 + 2 \times (k + N_{\text{vis}})^2 \times d_{\text{model}}$ (Transformer)
- Decoder: Same as encoder
- Total per batch (B=8): $\approx 1 - 2$ billion MACs $\Rightarrow$ **1050 ms on V100 GPU**

Training time:

- Dataset size: $N = 10,000$ demonstrations
- Batch size: $B = 8$
- Epochs: $E = 100$
- Time per epoch: ≈ 10 s (on V100)
- **Total training time:** ≈ 1520 **minutes**

19.11 Summary: From Multimodality to Coherent Action Sequences

Core Insight: By combining **action chunking** (predicts coherent trajectory segments), **CVAE** (models multimodal distributions), and **Transformers** (fuses vision + proprioception + history), ACT solves the fundamental problem in robot imitation learning: capturing diverse, physically realistic behaviors without averaging modes.

Key Equations to Remember:

$$\mathcal{L}_{\text{total}} = \underbrace{\|\mathbf{A} - \hat{\mathbf{A}}\|_2^2}_{\text{Reconstruction}} + \beta \underbrace{\frac{1}{2}\sum_j (\mu_j^2 + \sigma_j^2 - \log \sigma_j^2 - 1)}_{\text{KL Regularization}}, \tag{19.46}$$

$$\mathbf{z} = \boldsymbol{\mu}_\phi + \boldsymbol{\sigma}_\phi \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{19.47}$$

$$\hat{\mathbf{A}} = \text{Decoder}_\theta(\mathbf{z}, \text{Vision Features}, \mathbf{o}). \tag{19.48}$$

This framework is the foundation for later chapters: Diffusion Policy (alternative to CVAE), VLA (unified token space), and the numerical walkthrough (Chapter 20).

Chapter 20

Diffusion Policy

Diffusion Policy represents a paradigm shift in robot learning, moving away from explicit deterministic policies ($\mathbf{a} = \pi(\mathbf{o})$) or simple unimodal Gaussian policies ($\mathbf{a} \sim \mathcal{N}(\mu, \sigma)$) towards **score-based generative modeling**. It treats robot action generation as a conditional denoising process, learning to gradually refine a sequence of actions from random Gaussian noise into a coherent trajectory.

This approach effectively handles **multimodal action distributions** (e.g., passing a specific obstacle on the left or right, but never the average) and provides disjoint, high-quality solutions where standard regression methods fail.

20.1 Mathematical Formulation of Diffusion for Control

The core idea is to model the conditional distribution of action sequences $p(\mathbf{A}|\mathbf{O})$ using a Denoising Diffusion Probabilistic Model (DDPM).

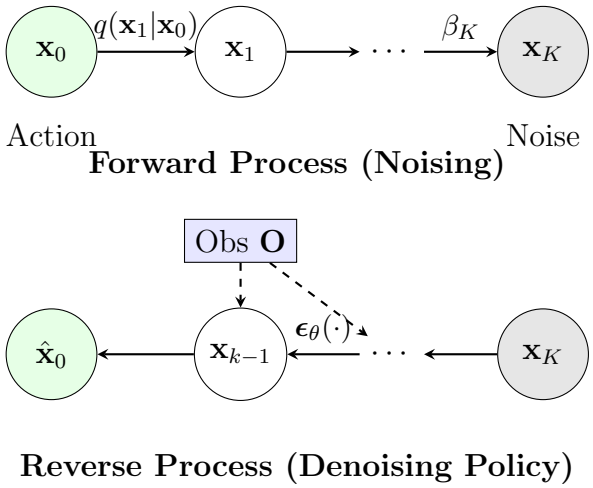

Figure 20.1: Diffusion Policy. Top: The forward process gradually destroys structure by adding noise. Bottom: The reverse process learns to remove noise, conditioned on observations $\mathbf{O}$, to recover the action $\mathbf{x}_0$.

20.2 Forward Process (Diffusion)

Definition 18.1 (Action Trajectory Space). Let $\mathbf{x}_0 \in \mathbb{R}^{T_a \times D}$ be a sequence of action vectors (an action chunk) of length T_a with action dimension D. The forward diffusion process is a fixed Markov chain that gradually adds Gaussian noise to the data $\mathbf{x}_0$ over K steps.

For steps $k = 1, \ldots, K$:

$$q(\mathbf{x}_k|\mathbf{x}_{k-1}) = \mathcal{N}\left(\mathbf{x}_k; \sqrt{1-\beta_k}\mathbf{x}_{k-1}, \beta_k\mathbf{I}\right) \tag{20.1}$$

where $\{\beta_k\}_{k=1}^K$ defines a variance schedule (typically linear or cosine).

Closed-Form Sampling: We can sample $\mathbf{x}_k$ at any step k directly from $\mathbf{x}_0$ using the reparameterization trick. Let $\alpha_k = 1 - \beta_k$ and $\bar{\alpha}_k = \prod_{s=1}^k \alpha_s$.

$$q(\mathbf{x}_k|\mathbf{x}_0) = \mathcal{N}\left(\mathbf{x}_k; \sqrt{\bar{\alpha}_k}\mathbf{x}_0, (1-\bar{\alpha}_k)\mathbf{I}\right) \tag{20.2}$$

Thus:

$$\mathbf{x}_k = \sqrt{\bar{\alpha}_k}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_k}\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{20.3}$$

As $k \to K$, $\bar{\alpha}_K \approx 0$, so $\mathbf{x}_K \approx \mathcal{N}(\mathbf{0}, \mathbf{I})$. This means the final state is pure Gaussian noise, devoid of the original action information.

20.3 Reverse Process (Denoising)

The goal of learning is to invert this process: starting from pure noise $\mathbf{x}_K \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and iteratively "denoising" it to recover a realistic action sequence $\mathbf{x}_0$, conditioned on an observation $\mathbf{O}$.

Definition 18.2 (Conditional Noise Prediction Network). We approximate the reverse transition $p_\theta(\mathbf{x}_{k-1}|\mathbf{x}_k, \mathbf{O})$ using a neural network $\boldsymbol{\epsilon}_\theta$. Ideally, the reverse mean is:

$$\boldsymbol{\mu}_\theta(\mathbf{x}_k, k, \mathbf{O}) = \frac{1}{\sqrt{\alpha_k}}\left(\mathbf{x}_k - \frac{\beta_k}{\sqrt{1-\bar{\alpha}_k}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_k, k, \mathbf{O})\right) \tag{20.4}$$

Instead of predicting the mean directly, we train the network to predict the **noise** $\boldsymbol{\epsilon}$ that was added.

20.4 Training Objective

The loss function is the Mean Squared Error (MSE) between the actual noise $\boldsymbol{\epsilon}$ and the predicted noise $\boldsymbol{\epsilon}_\theta$:

$$\mathcal{L}_{\text{diff}} = \mathbb{E}_{k\sim\mathcal{U}(1,K),\mathbf{x}_0\sim\mathcal{D},\boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}\left[\left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\underbrace{\sqrt{\bar{\alpha}_k}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_k}\boldsymbol{\epsilon}}_{\mathbf{x}_k}, k, \mathbf{O})\right\|^2\right] \tag{20.5}$$

Interpretation:

- Sample a random timestep k.
- Corrupt a clean action chunk $\mathbf{x}_0$ to get $\mathbf{x}_k$.
- The network $\boldsymbol{\epsilon}_\theta$ sees the noisy $\mathbf{x}_k$, the timestep k, and the visual observation $\mathbf{O}$.
- It tries to guess the noise $\boldsymbol{\epsilon}$ to "clean" the signal.

20.5 Network Architectures

Two main architectures are dominant in Diffusion Policy: **CNN-based (1D U-Net)** and **Transformer-based**.

20.5.1 1. CNN-Based (1D Temporal U-Net)

Used for shorter action horizons or when computational efficiency is key.

- **Input:** Noisy action sequence $\mathbf{x}_k \in \mathbb{R}^{T_a \times D}$.
- **Conditioning:** Observation embedding $\mathbf{e}_{\text{obs}}$ is injected via **FiLM (Feature-wise Linear Modulation)** layers at each residual block.

$$\text{FiLM}(\mathbf{h}; \gamma, \beta) = \gamma(\mathbf{e}_{\text{obs}}) \odot \mathbf{h} + \beta(\mathbf{e}_{\text{obs}}) \tag{20.6}$$

- **Structure:** Downsamples temporal resolution (Conv1D $\rightarrow$ Pooling) to capture high-level structure, then upsamples (ConvTranspose1D) to reconstruct details.

20.5.2 2. Transformer-Based (DiT)

Used for complex, multimodal distributions and long horizons.

- **Input:** Action tokens + Observation tokens.
- **Conditioning:** Cross-attention between noisy action tokens (Queries) and observation tokens (Keys/Values).
- **Positional Encoding:** Sinusoidal embeddings are added to action tokens to denote time index t and diffusion step k.

20.6 Inference: DDIM Sampling

Standard DDPM sampling requires traversing all K steps (e.g., $K = 1000$), which is too slow for real-time robotic control (often requiring >10Hz). **Denoising Diffusion Implicit Models (DDIM)** allow us to skip steps, solving the reverse ODE with fewer iterations (e.g., 10–20 steps) without retraining.

DDIM Update Rule: To go from step k to k' (where $k' < k$):

1. **Predict $\mathbf{x}_0$:**

$$\hat{\mathbf{x}}_0 = \frac{\mathbf{x}_k - \sqrt{1 - \bar{\alpha}_k}\boldsymbol{\epsilon}_\theta(\mathbf{x}_k, k, \mathbf{O})}{\sqrt{\bar{\alpha}_k}} \tag{20.7}$$

2. **Point to next state $\mathbf{x}_{k'}$:**

$$\mathbf{x}_{k'} = \sqrt{\bar{\alpha}_{k'}}\hat{\mathbf{x}}_0 + \sqrt{1 - \bar{\alpha}_{k'} - \sigma_{k'}^2}\boldsymbol{\epsilon}_\theta(\mathbf{x}_k, k, \mathbf{O}) + \sigma_{k'}\boldsymbol{\epsilon}' \tag{20.8}$$

Setting $\sigma_{k'} = 0$ yields a deterministic sampling process (ODE flow), which is often preferred for stability in robotics.

20.7 Algorithm: Implementation Summary

Algorithm 1 Diffusion Policy Training Loop

Input: Dataset $\mathcal{D} = \{(\mathbf{O}, \mathbf{x}_0)\}$, Diffusion steps K
repeat
  Sample batch $(\mathbf{O}, \mathbf{x}_0) \sim \mathcal{D}$
  Sample $k \sim \text{Uniform}(\{1, \dots, K\})$
  Sample noise $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
  Compute noisy action $\mathbf{x}_k = \sqrt{\bar{\alpha}_k}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_k}\boldsymbol{\epsilon}$
  Compute network output $\hat{\boldsymbol{\epsilon}} = \boldsymbol{\epsilon}_\theta(\mathbf{x}_k, k, \mathbf{O})$
  Update parameters θ to minimize $\|\boldsymbol{\epsilon} - \hat{\boldsymbol{\epsilon}}\|^2$
until converged

Algorithm 2 Diffusion Policy Inference (DDIM)

Input: Observation $\mathbf{O}$, Horizon T_a, Sampling steps S
Initialize $\mathbf{x}_K \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ ($T_a \times D$ Gaussian noise)
for $i = S, S-1, \dots, 1$ **do**
  Map i to effective diffusion step k_i
  Predict noise $\hat{\boldsymbol{\epsilon}} = \boldsymbol{\epsilon}_\theta(\mathbf{x}_{k_i}, k_i, \mathbf{O})$
  Compute $\mathbf{x}_{k_{i-1}}$ using DDIM update rule
end for
Output: Clean action sequence $\mathbf{x}_0$ (execute first M steps)

20.8 Why Diffusion for Robotics?

20.8.1 Multimodality

In the "avoid obstacle" scenario, a standard MSE policy learns the average path (crashing into the obstacle). A Gaussian policy (CVAE or MDN) might collapse to a single mode. Diffusion models naturally represent the full distribution, allowing the samples to bifrucate into valid left or right trajectories.

20.8.2 Stability

Unlike GANs (min-max instability) or Energy-Based Models (MCMC convergence issues), specific Diffusion objectives are strictly convex (MSE) and training is highly stable.

20.8.3 Action Consistency

The denoising process operates on the *entire trajectory* at once. This ensures that the generated $\mathbf{a}_t, \mathbf{a}_{t+1}, \dots$ are kinematically consistent and smooth, reducing the need for post-hoc filtering (like One-Euro filters).

Chapter 21

Vision-Language-Action (VLA) Models

Vision-Language-Action (VLA) models (e.g., RT-2) represent a unification of perception, reasoning, and control into a single transformer-based generative framework. This chapter provides a rigorous mathematical formulation of transforming robotic control into a token generation stability problem.

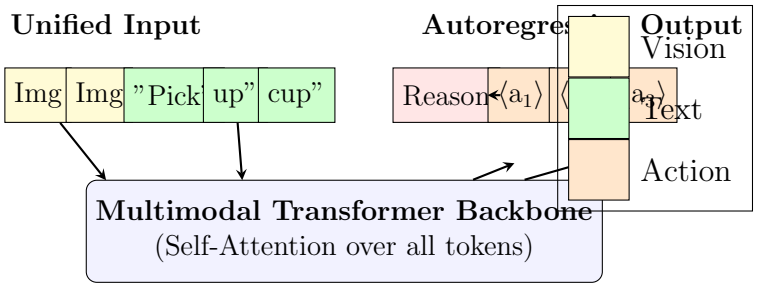

Figure 21.1: VLA Architecture (RT-2). Vision, Text, and Actions are treated as a unified sequence of tokens. The model generates reasoning (Chain-of-Thought) followed by discretized action tokens.

21.1 Unified Token Representation

The core premise of VLA is that robot actions are just another language. We explicitly define the unified vocabulary.

Definition 19.1 (Unified Vocabulary Space). Let $\mathcal{V}$ be the total discrete vocabulary used by the model. It is composed of the disjoint union of three modalities:

$$\mathcal{V} = \mathcal{V}_{\text{text}} \cup \mathcal{V}_{\text{image}} \cup \mathcal{V}_{\text{action}}, \tag{21.1}$$

where:

- $\mathcal{V}_{\text{text}}$: Standard LLM tokens (e.g., SentencePiece, size $\approx 30,000$–$50,000$).
- $\mathcal{V}_{\text{image}}$: Discrete visual patch tokens or learned soft-embeddings.
- $\mathcal{V}_{\text{action}}$: Discretized control commands.

The total vocabulary size is $|\mathcal{V}| = |\mathcal{V}_{\text{text}}| + |\mathcal{V}_{\text{image}}| + |\mathcal{V}_{\text{action}}|$.

21.2 Action Tokenization Details

To treat continuous motor signals as discrete tokens, we must quantize the action space.

Definition 19.2 (Action Discretization). Consider a continuous action vector $\mathbf{a} \in \mathbb{R}^D$, where each dimension $d \in \{1, \dots, D\}$ represents a degree of freedom (e.g., joint angle) bounded by $[a_{\min}^{(d)}, a_{\max}^{(d)}]$. We divide each dimension into N_{bins} uniform intervals. The function $\phi : \mathbb{R} \to \{0, \dots, N_{\text{bins}} - 1\}$ maps a continuous value x to a discrete bin index:

$$\phi(x) = \text{clip}\left(\left\lfloor \frac{x - a_{\min}}{a_{\max} - a_{\min}} \times N_{\text{bins}} \right\rfloor, 0, N_{\text{bins}} - 1 \right). \tag{21.2}$$

For a 7-DOF arm ($D = 7$) with $N_{\text{bins}} = 256$, the total action vocabulary size contribution is:

$$|\mathcal{V}_{\text{action}}| = D \times N_{\text{bins}} = 7 \times 256 = 1792 \text{ tokens}. \tag{21.3}$$

This linear scaling avoids the "curse of dimensionality" that would arise if we defined a unique token for every combination of joint angles (256^7 states).

Quantization Error Analysis: For a standard Panda arm joint range of $[-\pi, \pi]$ (2π rad):

$$\Delta\theta = \frac{2\pi}{256} \approx 0.0245 \text{ rad} \approx 1.4^\circ. \tag{21.4}$$

The expected quantization error (uniform distribution) is half the bin width: $\approx 0.7^\circ$. This provides sufficient precision for most manipulation tasks.

21.3 Image Tokenization (Vision Transformer)

Images are ingested as sequences of patch embeddings.

Definition 19.3 (Patch Embedding). Given an image $\mathbf{I} \in \mathbb{R}^{H \times W \times C}$, we divide it into patches of size $P \times P$. The number of patches is $N_p = (H/P) \times (W/P)$. For $H = W = 224$ and $P = 16$:

$$N_p = \frac{224}{16} \times \frac{224}{16} = 14 \times 14 = 196 \text{ patches}. \tag{21.5}$$

Each patch $\mathbf{x}_p$ is flattened and projected linearly to dimension d_{model}:

$$\mathbf{z}_p = \mathbf{x}_p \mathbf{W}_{\text{proj}} + \mathbf{e}_{\text{pos}}^{(p)}, \tag{21.6}$$

where $\mathbf{e}_{\text{pos}}$ is a learnable position embedding.

21.4 Unified Embedding Space

All tokens, regardless of modality, are mapped to the same vector space.

Definition 19.4 (Shared Embedding Matrix). Let $\mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d_{\text{model}}}$ be the master embedding matrix. Any token index $s \in \mathcal{V}$ is converted to a vector $\mathbf{h} \in \mathbb{R}^{d_{\text{model}}}$ via lookup:

$$\mathbf{h} = \mathbf{E}[s]. \tag{21.7}$$

Theorem 19.1 (Alignment Necessity). *For a single Transformer backbone to process multimodal data, the semantic spaces of text, vision, and action must be linearly aligned in* $\mathbb{R}^{d_{model}}$.

Proof. Let $\mathbf{h}_{\text{img}}$ be a visual feature and $\mathbf{h}_{\text{txt}}$ be a text instruction. Adaptation layers (like PaLM-E's projector) train a mapping $f(\mathbf{h}_{\text{img}}) \approx \mathbf{h}_{\text{txt}}$ such that the self-attention mechanism $\text{Attention}(Q, K, V)$ yields high compatibility scores $\mathbf{Q}_{\text{txt}}\mathbf{K}_{\text{img}}^{\top}$. Without this alignment, the attention weights would effectively ignore cross-modal dependencies. □

Definition 19.5 (Input Sequence Composition). A typical input sequence for a task like "Grasp the cup" is structured as:

$$\mathbf{S}_{\text{in}} = [\underbrace{\mathbf{v}_1, \ldots, \mathbf{v}_{196}}_{\text{Image Patches}}, \quad \underbrace{t_1, t_2, t_3}_{\text{"Grasp the cup"}} \quad]. \tag{21.8}$$

21.5 Mathematical Formulation of OpenVLA

21.5.1 Unified Token Space

OpenVLA unifies vision, language, and action into a single autoregressive generation framework. The core mathematical transformation is the discretization of the continuous action space $\mathcal{A}_{\text{cont}} \subseteq \mathbb{R}^{d_a}$ into a discrete vocabulary $\mathcal{V}_{\text{action}}$.

21.5.2 Action Quantization

Let a continuous action vector be $\mathbf{a} \in [a_{\min}, a_{\max}]^{d_a}$. We discretize each dimension i into $B = 256$ bins uniformly. The quantization function $\Phi : \mathbb{R} \to \{0, \ldots, B-1\}$ is defined as:

$$\Phi(x_i) = \text{round}\left((B-1) \times \frac{x_i - a_{\min}}{a_{\max} - a_{\min}} \right). \tag{21.9}$$

The continuous action $\mathbf{a}$ is thus converted into a sequence of discrete tokens:

$$\text{tokens}_{\text{action}} = [\Phi(a_1), \Phi(a_2), \ldots, \Phi(a_{d_a})]. \tag{21.10}$$

These tokens are added to the LLM's vocabulary, extending it to $\mathcal{V}_{\text{total}} = \mathcal{V}_{\text{text}} \cup \mathcal{V}_{\text{action}}$.

21.5.3 Autoregressive Objective

The model is trained as a standard Decoder-only Transformer. Given a sequence of inputs including the image patches $\mathbf{I}$, text instruction $\mathbf{T}$, and history, the probability of generating the action sequence is modeled via the chain rule:

$$P(\mathbf{a}|\mathbf{I}, \mathbf{T}) = \prod_{j=1}^{d_a} P(token_j | \mathbf{I}, \mathbf{T}, token_{<j}). \tag{21.11}$$

The training loss is the standard Cross-Entropy loss over the discrete action tokens:

$$\mathcal{L}_{\text{VLA}} = -\sum_{j=1}^{d_a} \log P_\theta(token_j^{\text{GT}} | \mathbf{I}, \mathbf{T}, token_{<j}^{\text{GT}}). \tag{21.12}$$

21.5.4 De-quantization (Inference)

During inference, the model outputs discrete token indices $\hat{k}_i \in \{0, \dots, B-1\}$. These are converted back to continuous control signals via the inverse function Φ^{-1}:

$$\hat{a}_i = a_{\min} + \frac{\hat{k}_i + 0.5}{B}(a_{\max} - a_{\min}). \tag{21.13}$$

Using the center of the bin (+0.5) minimizes the expected quantization error. This approach allows leveraging the massive pre-trained knowledge of LLMs (like Llama) for robotic control.

21.6 Transformer-Based Action Generation

The model operates as a standard causal decoder.

Definition 19.6 (Forward Pass). Let $\mathbf{H}^{(0)} = [\mathbf{h}_1, \dots, \mathbf{h}_T]$ comprise the embeddings of the input sequence. The sequence flows through L layers:

$$\mathbf{H}^{(\ell)} = \text{TransformerBlock}(\mathbf{H}^{(\ell-1)}), \quad \ell = 1, \dots, L. \tag{21.14}$$

The final layer $\mathbf{H}^{(L)}$ is projected back to the vocabulary size to produce logits:

$$\mathbf{L} = \mathbf{H}^{(L)}\mathbf{W}_{\text{unembed}}, \quad \mathbf{L} \in \mathbb{R}^{T \times |\mathcal{V}|}. \tag{21.15}$$

The probability of the next token w_{t+1} given history $w_{1:t}$ is:

$$P(w_{t+1}|w_{1:t}) = \text{softmax}(\mathbf{L}_{t,:}). \tag{21.16}$$

Definition 19.7 (Causal Masking). To ensure autoregressive generation, we apply a mask $\mathbf{M} \in \mathbb{R}^{T \times T}$ to the attention scores:

$$M_{i,j} = \begin{cases} 0 & \text{if } j \le i \\ -\infty & \text{otherwise} \end{cases}. \tag{21.17}$$

This enforces that position i can only attend to positions $j \le i$.

Definition 19.8 (Language Modeling Loss). The objective function minimizes the negative log-likelihood of the ground truth sequence:

$$\mathcal{L}_{\text{VLA}} = -\sum_{t=1}^{T-1} \log P(w^*_{t+1}|w_1, \dots, w_t), \tag{21.18}$$

where only the loss on $\mathcal{V}_{\text{action}}$ tokens (and optionally reasoning text) is typically back-propagated during fine-tuning on robot data.

21.7 Chain-of-Thought (CoT) Reasoning

VLA models can "think" before they act.

Definition 19.9 (Reasoning Tokens). Let $\mathbf{R} = [r_1, \dots, r_m] \in \mathcal{V}^m_{\text{text}}$ be intermediate reasoning steps generated by the model before specifying actions.

Definition 19.10 (Reasoning-Action Sequence). The target output sequence becomes:

$$\mathbf{S}_{\text{out}} = [\underbrace{\text{"(1) Locate handle..."}}_{\text{Reasoning } \mathbf{R}}, \underbrace{\langle \text{act}_0 \rangle, \ldots, \langle \text{act}_{D-1} \rangle}_{\text{Action } \mathbf{A}}]. \tag{21.19}$$

Definition 19.11 (Conditional Action Loss). The probability of correct action depends explicitly on the reasoning trace:

$$P(\mathbf{A}|\text{Obs}) = P(\mathbf{A}|\mathbf{R}, \text{Obs}) \cdot P(\mathbf{R}|\text{Obs}). \tag{21.20}$$

Maximizing $P(\mathbf{R}|\text{Obs})$ forces the model to ground its plan in the observation before committing to motor commands.

21.8 Action Reconstruction

After generating tokens, we must revert to continuous physical control.

Definition 19.11 (Inverse Discretization). Given a predicted token index $k \in \{0, \ldots, N_{\text{bins}} - 1\}$ for dimension d, the continuous value $\hat{a}^{(d)}$ is recovered by the center of the bin:

$$\hat{a}^{(d)} = a_{\min}^{(d)} + (k + 0.5) \times \frac{a_{\max}^{(d)} - a_{\min}^{(d)}}{N_{\text{bins}}}. \tag{21.21}$$

Decoding Strategies:

- **Greedy Decoding:** Select $\arg\max P(w)$. Lowest latency, but deterministic.
- **Temperature Sampling:** Sample from $P(w)^{1/T_{\text{temp}}}$. Adds diversity for exploration.
- **Constrained Decoding:** Set $P(w) = 0$ for all $w \notin \mathcal{V}_{\text{action}}$ when an action is expected. This guarantees syntax validity.

21.9 Multimodality in VLA

Definition 19.12 (Multimodal Probabilities via Sampling). Unlike Diffusion (which models the score function of the distribution), VLA models the categorical distribution via the LLM logits. A multimodal distribution (e.g., "go left or right") appears as two distinct peaks in the softmax probability distribution over the first action token. Sampling draws from one peak, collapsing the mode for subsequent autoregressive steps.

Table 21.1: Handling Multimodality: Comparison

Method	**Mechanism**	**Pros/Cons**
ACT	CVAE (Latent z)	Fast, but assumes Gaussian modes.
Diffusion	Score Denoising	High fidelity, slow inference.
VLA	Categorical Sampling	Flexible, limited by bin resolution.

21.10 Computational Complexity and Optimization

Theorem 19.2 (Attention Complexity). *The computational cost of the self-attention mechanism scales quadratically with sequence length N.*

$$\text{Cost} \propto O(N^2 d_{\text{model}}). \tag{21.22}$$

Proof. For a sequence length N, computing $\mathbf{QK}^\top$ involves an $N \times N$ matrix multiplication. Specifically:

- QKT: $N^2 d$ ops.
- Softmax: N^2 ops.
- Attention $\times$ V: $N^2 d$ ops.

Total dominant term is $O(N^2 d)$. □

FLOP Estimation (V100): For a VLA with 1B parameters and sequence length 1024, a single forward pass takes roughly 160 ms on a V100 GPU (without optimization).

Definition 19.14 (KV Cache). To speed up autoregressive generation, we cache keys and values of past tokens:

$$\text{Cache}_t = \{(\mathbf{k}_i, \mathbf{v}_i) \mid i = 1 \dots t-1\}. \tag{21.23}$$

At step t, we only compute $\mathbf{q}_t$, $\mathbf{k}_t$, $\mathbf{v}_t$ and attend to $\text{Cache}_t \cup \{(\mathbf{k}_t, \mathbf{v}_t)\}$. This reduces complexity from $O(t^2)$ to $O(t)$ per step.

Memory Requirements (A100): A 7B parameter model (float16) requires ≈ 14 GB for weights. The KV cache for long sequences can add 10-30 GB, often necessitating a 40GB or 80GB A100 for inference.

21.11 Comparison and Summary

Table 21.2: Comprehensive Comparison: ACT vs Diffusion vs VLA

Feature	**ACT**	**Diffusion**	**VLA (RT-2)**
Core Math	CVAE + Transformer	Score Matching (SDE)	Categorical Cross-Ent.
Action Space	Continuous	Continuous	Discrete (Tokenized)
Outputs	Action Chunk ($k = 100$)	Action Chunk ($k = 100$)	single step (usually)
Inference Speed	Very Fast (>50Hz)	Slow (10-20Hz)	Slow (3-10Hz)
Multimodality	Via Latent z	Via Noise Refinement	Via Sampling
Pretraining	No (Scratch)	No (Scratch)	Yes (Internet Data)
Generalization	Low (In-domain)	Low (In-domain)	High (Semantic)

21.12 Mathematical Summary

The transformation from observation to action in VLA is a chain of probabilities:

$$\mathbf{a}^* = \text{decode}\left(\arg\max_{w \in \mathcal{V}_{\text{action}}} P(w|\text{Encoder}(\mathbf{I}), \text{Tokenizer}(\text{Cmd}))\right). \tag{21.24}$$

By leveraging the pre-trained world knowledge of LLMs, VLA models achieve unprecedented generalization, albeit at the cost of higher inference latency and hardware requirements.

Chapter 22

Full Numerical Walkthrough: ACT Implementation

This chapter provides a **complete tensor size trace** for ACT, following the same detailed methodology as Chapter 6 of the main text. We assume the standard configuration used in ALOHA/ACT implementations.

22.1 Configuration

- **Input images:** 2 cameras (wrist + overhead), each $480 \times 640 \times 3$ RGB, resized to 224×224 (or 84×84 for smaller nets). Let's assume 224×224 for ResNet-18.
- **Joint degrees of freedom:** $D = 14$ (two 7-DOF arms) or $D = 7$ (one arm). We assume **bimanual** ($D = 14$).
- **Action chunk size:** $k = 100$ steps.
- **Batch size:** $B = 8$.
- **Latent dimension:** $d_z = 32$.
- **Transformer hidden dimension:** $d_{\text{model}} = 512$.
- **Feedforward dimension:** $d_{\text{ffn}} = 3200$.
- **Heads:** $H = 8$.

22.2 Forward Pass: Complete Tensor Tracking

22.2.1 Vision Backbone (ResNet-18)

Input

Two images per sample, batch of 8.

$$\mathbf{I} \in \mathbb{R}^{B \times 2 \times 3 \times 224 \times 224} \tag{22.1}$$

Feature Exploration

The ResNet-18 removes the pooling layer and the final fully connected layer. Output of the last convolutional layer (layer4):

$$\text{Shape: } [B \cdot 2, 512, 7, 7] \tag{22.2}$$

We flatten the spatial dimensions to treat them as a sequence.

$$\text{Tokens}_{\text{vis}} \in \mathbb{R}^{(B\cdot 2)\times 49\times 512} \tag{22.3}$$

Reshape back to batch dimension:

$$\mathbf{F}_{\text{vis}} \in \mathbb{R}^{B\times(2\times 49)\times 512} = \mathbb{R}^{8\times 98\times 512} \tag{22.4}$$

We add a learnable 2D sinusoidal position embedding to these 98 tokens.

22.2.2 Transformer Encoder (Perception)

Input Composition

The encoder fuses joint state (proprioception) and visual tokens.

- Proprioception: $\mathbf{q}_t \in \mathbb{R}^{B\times 14}$. Projected to $\mathbb{R}^{B\times 1\times 512}$.
- CLS token: $\mathbb{R}^{B\times 1\times 512}$.
- Sequence: $[\text{CLS}, \text{Prop}, \text{Vis}_1, \ldots, \text{Vis}_{98}]$. Total length $1 + 1 + 98 = 100$ tokens.

$$\mathbf{X}_{\text{enc}} \in \mathbb{R}^{8\times 100\times 512} \tag{22.5}$$

Self-Attention

Standard Transformer Encoder steps (Self-Attention + MLP) for $L = 4$ layers. Output:

$$\mathbf{H}_{\text{enc}} \in \mathbb{R}^{8\times 100\times 512} \tag{22.6}$$

22.2.3 CVAE Encoder (Training Only)

Goal: Compress future action sequence into latent $\mathbf{z}$. Input: True actions $\mathbf{a}_{t:t+k} \in \mathbb{R}^{B\times 100\times 14}$.

1. Flatten actions or embed time: $\mathbf{a}_{\text{seq}} \in \mathbb{R}^{B\times 100\cdot 14}$ (if MLP) or keep temporal (if Transformer). ACT uses a [CLS] token style encoding over the sequence.
2. Let's assume a simple BERT-like encoder over actions.
3. Output: $\boldsymbol{\mu}, \boldsymbol{\sigma} \in \mathbb{R}^{B\times 32}$.

Sampling $\mathbf{z}$:

$$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}, \quad \mathbf{z} \in \mathbb{R}^{8\times 32} \tag{22.7}$$

22.2.4 Transformer Decoder (Policy)

Inputs

- **Query (Q):** Fixed positional embeddings representing the action chunk time steps $0 \dots 99$.
$$\mathbf{Q}_{\text{dec}} \in \mathbb{R}^{B \times 100 \times 512} \tag{22.8}$$
- **Key/Value (K, V):** From Encoder output $\mathbf{H}_{\text{enc}}$. We also append the latent $\mathbf{z}$ (projected) to the input sequence of the encoder? Actually, ACT usually prepends $\mathbf{z}$ to the input of the *Encoder*, or adds it as a style variable. In standard DETR-like decoding, it's often:
$$\mathbf{K}, \mathbf{V} = \text{Concat}(\mathbf{H}_{\text{enc}}, \text{Project}(\mathbf{z})) \in \mathbb{R}^{8 \times 101 \times 512} \tag{22.9}$$

Cross-Attention Layers

The decoder attends to the perception features ($\mathbf{H}_{\text{enc}}$) conditioned on the "time queries".

$$\text{Attn}(\mathbf{Q}_{\text{dec}}, \mathbf{K}, \mathbf{V}) \to \mathbb{R}^{8 \times 100 \times 512} \tag{22.10}$$

Prediction Heads

A simple MLP projects the 512-dim tokens to action dimension 14.

$$\hat{\mathbf{A}} = \text{MLP}(\mathbf{H}_{\text{dec}}) \in \mathbb{R}^{8 \times 100 \times 14} \tag{22.11}$$

22.3 Computational Cost Analysis

22.3.1 FLOPs Breakdown

- **ResNet-18:** ≈ 1.8 GFLOPs per image. 2 images $\to 3.6$ GFLOPs.
- **Transformer Encoder ($N = 100, d = 512$):** Self-Attention: $4N^2d = 4 \cdot 10000 \cdot 512 \approx 20$ MFLOPs. MLP: $8Nd^2 = 8 \cdot 100 \cdot 262144 \approx 200$ MFLOPs. Total per layer: ≈ 0.22 GFLOPs. 4 layers $\to 0.9$ GFLOPs.
- **Total:** dominated by Vision Backbone ($\approx 80\%$).

22.3.2 Memory

Batch size 8, 100-step chunk. Activations are manageable. The main VRAM usage comes from the stored feature maps of the ResNet for backpropagation. Inference can run easily on a consumer GPU (e.g., RTX 3080) at > 50 Hz.

22.4 Summary of Tensor Shapes

Stage	**Tensor**	**Shape**
Input Images	$\mathbf{I}$	$(8, 2, 3, 224, 224)$
CNN Feats	$\mathbf{F}_{\text{vis}}$	$(8, 98, 512)$
Joint State	$\mathbf{q}$	$(8, 14)$
Encoder Input	$\mathbf{X}_{\text{enc}}$	$(8, 100, 512)$
Latent	$\mathbf{z}$	$(8, 32)$
Decoder Query	$\mathbf{Q}_{\text{dec}}$	$(8, 100, 512)$
Output Actions	$\hat{\mathbf{A}}$	$(8, 100, 14)$

Chapter 23

LeRobot: Open-Source Ecosystem for Embodied AI

While theoretical formulations like ACT and Diffusion Policy provide the "brain" for robots, applying them to the real world requires solving significant engineering challenges: heterogeneous data formats, hardware interfaces, and reproducible training loops. **LeRobot** is an open-source library developed by Hugging Face to address these issues, aiming to do for robotics what the `transformers` library did for NLP.

23.1 Core Philosophy: The Data Unification Problem

Historically, robotic datasets used custom formats (HDF5, pickle, ROS bags) with varying schemas for joint angles, velocities, and images. This fragmentation made multi-robot training impossible.

LeRobot solves this by standardizing data into the **Hugging Face Datasets** format (Apache Parquet), treating robot trajectories as time-series data with a unified schema.

23.1.1 Mathematical Data Abstraction

LeRobot abstracts a robotic dataset $\mathcal{D}$ as a collection of temporal frames. For a trajectory τ of length T:

$$\tau = \{(\mathbf{o}_t, \mathbf{a}_t)\}_{t=0}^{T-1} \tag{23.1}$$

The library enforces a strict schema for observations $\mathbf{o}_t$ and actions $\mathbf{a}_t$:

- **Observations**: $\mathbf{o}_t = \{\text{image}_1, \dots, \text{image}_N, \mathbf{q}_{\text{pos}}, \mathbf{q}_{\text{vel}}\}$. Images are stored as encoded bytes (PNG/WebP) to save space and decoded on-the-fly.
- **Actions**: $\mathbf{a}_t \in \mathbb{R}^D$. LeRobot typically records absolute joint positions, but supports delta-actions via normalization transforms.

23.2 Policy Abstraction and Training Loop

LeRobot provides a unified API for different policies (ACT, Diffusion, VQ-Bet). The training loop abstracts the complex input/output processing described in Chapters 19

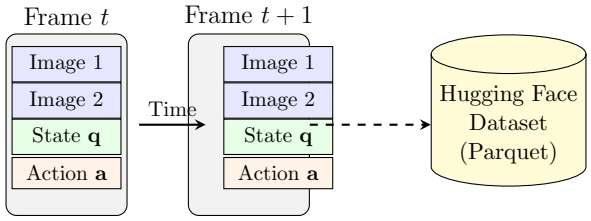

Figure 23.1: LeRobot Data Abstraction. Robot trajectories are flattened into a temporal sequence of frames containing observations (Images, State) and Actions. This unified schema (saved as Parquet) allows mixing datasets from different robots.

and 20.

23.2.1 Input Normalization (The Delta Problem)

Different robots have different joint limits. LeRobot handles normalization mathematically using pre-computed dataset statistics (mean $\boldsymbol{\mu}$ and std $\boldsymbol{\sigma}$). For a raw action $\mathbf{a}_{\text{raw}}$, the normalized input to the neural network $\mathbf{a}_{\text{norm}}$ is:

$$\mathbf{a}_{\text{norm}} = \text{clip}\left(\frac{\mathbf{a}_{\text{raw}} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, -C, C\right) \tag{23.2}$$

where C is a clipping threshold (usually 5.0 or 10.0) to handle outliers. This normalization is baked into the model's forward pass during inference, ensuring safety.

23.2.2 Unified Loss Calculation

The library abstracts the specific loss functions of different architectures into a common `compute_loss` interface:

$$\mathcal{L}(\theta) = \text{Policy}_\theta.\text{compute\_loss}(\text{batch}) \tag{23.3}$$

- For **ACT**: It automatically handles the VAE reconstruction loss and KL divergence, including the temporal ensembling logic during validation.
- For **Diffusion**: It handles the noise sampling $\epsilon \sim \mathcal{N}(0, I)$, the forward diffusion process $q(x_k|x_0)$, and the denoising loss calculation.

23.3 Hardware Abstraction: So-100 and WidowX

A key contribution of LeRobot is the software-hardware integration. It provides a control loop running at fixed frequency (e.g., 50Hz) that handles:

1. **Sensor Reading**: Synchronizing cameras and joint encoders.
2. **Inference**: Running the policy $\pi(a_t|o_t)$ (often with chunking $k = 100$).
3. **Actuation**: Sending commands via standard protocols (Dynamixel SDK, etc.).

23.3.1 The Chunking Buffer

To implement the **Temporal Ensembling** derived in the ACT chapter, LeRobot maintains a rolling buffer of predictions. Let $\hat{\mathbf{A}}_t$ be the chunk predicted at time t. The executable action is computed as:

$$\mathbf{a}_t^{\text{exec}} = \sum_{j=t-k+1}^{t} w_{t-j} \cdot \hat{\mathbf{A}}_j[t-j] \tag{23.4}$$

LeRobot handles this buffer management transparently, allowing researchers to focus on architecture design rather than real-time thread synchronization.

23.4 Summary: From Theory to Practice

LeRobot bridges the gap between mathematical formulation and physical deployment.

- **Theory**: We defined policies as conditional distributions $P(A|O)$.
- **LeRobot**: Implements $P(A|O)$ as a PyTorch Module, standardizes O and A via Parquet datasets, and connects the output to physical motors via a standardized control loop.

This standardization is crucial for the "Scaling Laws" of robotics, enabling training on mixed datasets from diverse robots (e.g., training a policy on combined ALOHA and WidowX data).

Part II

Vision and Image Recognition Architectures

Chapter 24

Convolutional Neural Networks: Foundations

The fully connected networks discussed in earlier chapters treat inputs as vectors, ignoring spatial structure. For image data, a 224×224 RGB image becomes a vector of $224 \cdot 224 \cdot 3 = 150,528$ elements, requiring millions of parameters per layer. Convolutional Neural Networks (CNNs) exploit local spatial structure and weight sharing to dramatically reduce parameters while increasing interpretability.

24.1 Convolution Operation

24.1.1 1D convolution (discrete)

For a 1D signal $x[t]$ (length T) and filter (kernel) $w[s]$ (length S), the discrete convolution is:

$$y[t] = \sum_{s=0}^{S-1} w[s] \cdot x[t+s] = (x * w)[t]. \tag{24.1}$$

With stride stride $= 1$ and valid padding (no zero-padding), output length is $T_{\text{out}} = T - S + 1$.

Padding and stride: With zero-padding p and stride stride:

$$T_{\text{out}} = \left\lfloor \frac{T + 2p - S}{\text{stride}} \right\rfloor + 1. \tag{24.2}$$

24.1.2 2D convolution (image)

For a 2D input $X[i, j]$ (height H, width W) and filter $W[u, v]$ (height K_h, width K_w):

$$Y[i, j] = \sum_{u=0}^{K_h-1} \sum_{v=0}^{K_w-1} W[u, v] \cdot X[i+u, j+v]. \tag{24.3}$$

Output dimensions:

$$H_{\text{out}} = \left\lfloor \frac{H + 2p_h - K_h}{s_h} \right\rfloor + 1, \quad W_{\text{out}} = \left\lfloor \frac{W + 2p_w - K_w}{s_w} \right\rfloor + 1, \tag{24.4}$$

where p_h, p_w are padding and s_h, s_w are strides.

24.1.3 Multi-channel convolution

For an input with C_{in} channels and output with C_{out} filters:

$$Y[i, j, c_{\text{out}}] = \sum_{c_{\text{in}}=0}^{C_{\text{in}}-1} \sum_{u=0}^{K_h-1} \sum_{v=0}^{K_w-1} W[u, v, c_{\text{in}}, c_{\text{out}}] \cdot X[i+u, j+v, c_{\text{in}}]. \tag{24.5}$$

Parameter count:

$$\#\text{params} = K_h \cdot K_w \cdot C_{\text{in}} \cdot C_{\text{out}} + C_{\text{out}}, \tag{24.6}$$

where the last term is biases.

Computational cost (FLOPs):

$$\text{FLOPs} \approx 2 \cdot K_h \cdot K_w \cdot C_{\text{in}} \cdot H_{\text{out}} \cdot W_{\text{out}} \cdot C_{\text{out}}. \tag{24.7}$$

24.1.4 Group convolution (depthwise separation)

Divide input channels into G groups; each group is convolved independently:

$$Y[i, j, c_{\text{out}}] = \sum_{u,v} W[u, v, c_{\text{in}}, c_{\text{out}}] \cdot X[i+u, j+v, c_{\text{in}}], \tag{24.8}$$

where c_{in} and c_{out} are constrained to groups.

Depthwise convolution ($G = C_{\text{in}}$): Each input channel has its own filter. Parameter reduction: factor $C_{\text{out}}/C_{\text{in}}$.

Depthwise-separable: Combine depthwise (per-channel) + pointwise (1×1) convolutions:

$$\text{FLOPs}_{\text{depthwise}} + \text{FLOPs}_{\text{pointwise}} \ll \text{FLOPs}_{\text{standard}}. \tag{24.9}$$

24.2 ResNet 50: Detailed Breakdown

ResNet-50 is a 50-layer deep residual network with skip connections. It processes a 224×224 RGB image through multiple stages.

Figure 24.1: ResNet-50 Architecture: Progressive downsampling with residual blocks. Each stage has skip connections that bypass non-linearities.

24.2.1 Residual block definition

A residual block with skip connection:

$$\mathbf{y} = \text{ReLU}(\mathbf{F}(\mathbf{x}) + \mathbf{x}), \tag{24.10}$$

where $\mathbf{F}(\mathbf{x})$ is the residual mapping (e.g., conv layers). If dimensions don't match, apply a linear projection:

$$\mathbf{y} = \text{ReLU}(\mathbf{F}(\mathbf{x}) + \mathbf{W}_s\mathbf{x}). \tag{24.11}$$

24.2.2 Bottleneck block

ResNet-50 uses "bottleneck" residual blocks to reduce computation:

$$\mathbf{F}(\mathbf{x}) = \text{conv}(1 \times 1, C/4) \to \text{conv}(3 \times 3, C/4) \to \text{conv}(1 \times 1, C), \tag{24.12}$$

where the middle 3×3 conv operates on reduced channels $C/4$, saving computation.

24.2.3 ResNet-50 architecture

Stage	Layer	Output Size	Blocks	Channels
Conv1	7×7, stride 2	112×112	1	64
Conv2	3×3, stride 1	56×56	3	256
Conv3	3×3, stride 2	28×28	4	512
Conv4	3×3, stride 2	14×14	6	1024
Conv5	3×3, stride 2	7×7	3	2048
Average Pool	-	1×1	1	2048
FC	-	-	1	1000 (ImageNet)

24.2.4 Forward pass: 224224 input step-by-step

Input: $X \in \mathbb{R}^{224\times224\times3}$.

Stage Conv1 (7×7 conv, stride 2):

$$H_{\text{out}} = \left\lfloor \frac{224 + 2 \cdot 3 - 7}{2} \right\rfloor + 1 = \left\lfloor \frac{223}{2} \right\rfloor + 1 = 112. \tag{24.13}$$

Output: $F_1 \in \mathbb{R}^{112\times112\times64}$.

Stage Conv2 (3 bottleneck blocks, stride 1): After max pooling (3×3, stride 2):

$$H = \left\lfloor \frac{112 - 3}{2} \right\rfloor + 1 = 56. \tag{24.14}$$

Output: $F_2 \in \mathbb{R}^{56\times56\times256}$.

Stage Conv3 (4 bottleneck blocks, stride 2):

$$H = \left\lfloor \frac{56 - 1}{2} \right\rfloor + 1 = 28. \tag{24.15}$$

Output: $F_3 \in \mathbb{R}^{28\times28\times512}$.

Stage Conv4 (6 bottleneck blocks, stride 2):

$$H = \left\lfloor \frac{28-1}{2} \right\rfloor + 1 = 14. \tag{24.16}$$

Output: $F_4 \in \mathbb{R}^{14\times14\times1024}$.

Stage Conv5 (3 bottleneck blocks, stride 2):

$$H = \left\lfloor \frac{14-1}{2} \right\rfloor + 1 = 7. \tag{24.17}$$

Output: $F_5 \in \mathbb{R}^{7\times7\times2048}$.

Global Average Pooling:

$$\mathbf{f} = \frac{1}{7\times7} \sum_{i,j} F_5[i,j,:] \in \mathbb{R}^{2048}. \tag{24.18}$$

Classification head (1000-way ImageNet):

$$\textbf{logits} = \mathbf{W}_c\mathbf{f} + \mathbf{b}_c \in \mathbb{R}^{1000}, \tag{24.19}$$

where $\mathbf{W}_c \in \mathbb{R}^{1000\times2048}$.

24.3 Feature Pyramid Network (FPN)

Object detection often requires features at multiple scales. FPN constructs a multi-scale feature pyramid from a backbone (e.g., ResNet-50).

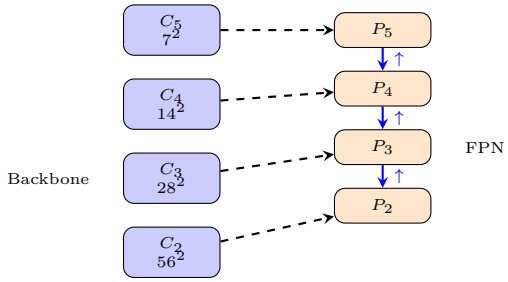

Figure 24.2: Feature Pyramid Network (FPN): Bottom-up backbone features are fused with top-down pathway via lateral connections (1×1 conv) and upsampling.

24.3.1 Bottom-up pathway

Backbone (e.g., ResNet) produces multi-scale features:

$$\{F_2, F_3, F_4, F_5\} \quad \text{with shapes } \{56\times56, 28\times28, 14\times14, 7\times7\} \times C. \tag{24.20}$$

24.3.2 Top-down pathway

Starting from the smallest feature map, upsample and fuse with lateral connections:

$$M_i = \text{Upsample}(M_{i+1}) + L_i, \tag{24.21}$$

where $L_i = \text{conv}(1\times1)(F_i)$ is the lateral connection, and Upsample is $2\times$ nearest-neighbor or bilinear.

Example FPN construction:

$$C_5 = F_5 \in \mathbb{R}^{7\times7\times2048}, \tag{24.22}$$
$$M_5 = \text{conv}(1 \times 1)(C_5) \in \mathbb{R}^{7\times7\times256}, \tag{24.23}$$
$$M_4 = \text{Upsample}(M_5) + \text{conv}(1 \times 1)(C_4) \in \mathbb{R}^{14\times14\times256}, \tag{24.24}$$
$$M_3 = \text{Upsample}(M_4) + \text{conv}(1 \times 1)(C_3) \in \mathbb{R}^{28\times28\times256}, \tag{24.25}$$
$$M_2 = \text{Upsample}(M_3) + \text{conv}(1 \times 1)(C_2) \in \mathbb{R}^{56\times56\times256}. \tag{24.26}$$

Output: Multi-scale features $\{M_2, M_3, M_4, M_5, M_6\}$ (where M_6 is stride-2 downsampling of M_5).

24.3.3 Use in detection

Each level M_i is fed to separate detection heads (bounding box and class prediction), enabling detection at multiple scales.

Chapter 25

Object Detection Fundamentals

Object detection extends classification: for each object in an image, predict its class and location (bounding box).

25.1 Anchor Boxes

25.1.1 Why anchors?

Anchor boxes are predefined reference boxes at each spatial location. The detector predicts offsets and class scores *relative to anchors*.

25.1.2 Anchor definition

For a feature map of size $H \times W$ with stride s relative to image, anchors are defined by:

$$\text{Anchor}[i, j, k] = \{\text{center}_x, \text{center}_y, \text{width}, \text{height}\}, \tag{25.1}$$

where:

$$\text{center}_x = (j + 0.5) \cdot s, \tag{25.2}$$
$$\text{center}_y = (i + 0.5) \cdot s, \tag{25.3}$$
$$\text{width}, \text{height from a set of aspect ratios and scales.} \tag{25.4}$$

Example (1313 feature map, stride 32): For cell $[i = 5, j = 7]$:

$$\text{center} = ((7 + 0.5) \cdot 32, (5 + 0.5) \cdot 32) = (240, 176). \tag{25.5}$$

25.1.3 Multiple anchors per cell

Typically, k anchors per cell with different aspect ratios and scales:

$$\text{Aspect ratios} = \{0.5, 1, 2\}, \quad \text{Scales} = \{1, 2^{1/3}, 2^{2/3}\}. \tag{25.6}$$

Total anchors: $13 \times 13 \times 9 = 1521$ (for 1313 feature map).

25.2 Bounding Box Representation and IoU

25.2.1 Formats

Center format: (c_x, c_y, w, h) center and size.

Corner format (XYXY): (x_1, y_1, x_2, y_2) top-left and bottom-right corners.

25.2.2 Conversion

Center to corner:

$$x_1 = c_x - w/2, \qquad y_1 = c_y - h/2, \tag{25.7}$$
$$x_2 = c_x + w/2, \qquad y_2 = c_y + h/2. \tag{25.8}$$

25.2.3 Intersection over Union (IoU)

$$\text{IoU}(\mathcal{B}_1, \mathcal{B}_2) = \frac{|\mathcal{B}_1 \cap \mathcal{B}_2|}{|\mathcal{B}_1 \cup \mathcal{B}_2|}. \tag{25.9}$$

Computation: Intersection area:

$$\text{Area}_\cap = \max(0, \min(x_2^{(1)}, x_2^{(2)}) - \max(x_1^{(1)}, x_1^{(2)})) \times \max(0, \ldots). \tag{25.10}$$

Union area:

$$\text{Area}_\cup = \text{Area}_1 + \text{Area}_2 - \text{Area}_\cap. \tag{25.11}$$

Example: Box1 $= [100, 100, 200, 200]$, Box2 $= [150, 100, 250, 200]$.
Intersection:

$$\text{dx} = \min(200, 250) - \max(100, 150) = 50, \quad \text{dy} = \min(200, 200) - \max(100, 100) = 100, \tag{25.12}$$

$$\text{Area}_\cap = 50 \times 100 = 5000. \tag{25.13}$$

Areas:

$$\text{Area}_1 = 100 \times 100 = 10000, \quad \text{Area}_2 = 100 \times 100 = 10000, \tag{25.14}$$

$$\text{Area}_\cup = 10000 + 10000 - 5000 = 15000. \tag{25.15}$$

IoU:

$$\text{IoU} = \frac{5000}{15000} = 0.333. \tag{25.16}$$

25.3 Non-Maximum Suppression (NMS)

25.3.1 Motivation

Multiple overlapping predictions for the same object; NMS removes duplicates by keeping high-confidence boxes and suppressing low-confidence overlaps.

Algorithm 3 Non-Maximum Suppression

```
Input: List of boxes with scores, IoU threshold τ
keep ← []
Sort boxes by score in descending order
while boxes not empty do
    B_max ← box with highest score
    keep ← keep ∪ {B_max}
    Remove B_max from boxes
    for each remaining box B do
        if IoU(B_max, B) > τ then
            Remove B from boxes
        end if
    end for
end while
Return keep
```

25.3.2 Algorithm

25.3.3 Numerical example

Input: 5 boxes with scores.

Box	Coords (XYXY)	Score	IoU w/ B1	Keep?
B1	[100, 100, 200, 200]	0.95		
B2	[105, 105, 205, 205]	0.90	0.815	
B3	[400, 400, 500, 500]	0.88		
B4	[405, 405, 505, 505]	0.85		
B5	[600, 600, 700, 700]	0.75		

Trace (IoU threshold = 0.5):

1. Sort by score: [B1: 0.95, B2: 0.90, B3: 0.88, B4: 0.85, B5: 0.75].
2. Keep B1, compute IoU with B2: $\text{IoU}(B1, B2) = \frac{10000}{(100\times100)+(100\times100)-10000+(5\times100\times2)} \approx 0.815 > 0.5$ Remove B2.
3. Keep B3, compute IoU with B4: $\text{IoU}(B3, B4) \approx 0.815 > 0.5$ Remove B4.
4. Keep B5.

Output: [B1, B3, B5].

25.4 Evaluation Metrics: mAP and COCO

25.4.1 Average Precision (AP)

Precision-Recall curve: vary confidence threshold, compute:

$$\text{Precision}(t) = \frac{\text{TP}(t)}{\text{TP}(t) + \text{FP}(t)}, \quad \text{Recall}(t) = \frac{\text{TP}(t)}{\text{TP}(t) + \text{FN}}. \tag{25.17}$$

Average Precision (at IoU threshold, e.g., 0.5):

$$\mathrm{AP}_{0.5} = \int_0^1 \mathrm{Precision}(r)\, dr, \tag{25.18}$$

computed numerically by summing areas under PR curve.

25.4.2 Mean AP (mAP)

$$\mathrm{mAP}_{0.5} = \frac{1}{C} \sum_{c=1}^{C} \mathrm{AP}_{0.5}^{(c)}, \tag{25.19}$$

where C is number of classes.

COCO metric averages over IoU thresholds:

$$\mathrm{mAP} = \frac{1}{10} \sum_{t \in \{0.5, 0.55, \ldots, 0.95\}} \mathrm{mAP}_t. \tag{25.20}$$

25.4.3 COCO dataset

Common Objects in Context (COCO):

- $\sim 330K$ images, 80 object classes.
- ~ 5 objects per image on average.
- Pixel-level masks for segmentation.
- Standard benchmark: mAP, mAP_{50}, mAP_{75}, etc.

Chapter 26

YOLO: Real-Time Object Detection

YOLO (You Only Look Once) reformulates detection as a single-shot regression problem: predict bounding boxes and class probabilities directly from full images in one forward pass.

26.1 Historical Evolution of Mathematical Formulation

The YOLO architecture has evolved not just in terms of layer depth, but fundamentally in how the detection problem is parameterized and how the loss landscape is constructed. We analyze these changes through their mathematical formulations.

26.1.1 YOLOv1: Direct Regression on Grids (2016)

The original YOLO formulation treats detection as a regression problem from a fixed grid to bounding box coordinates, without anchor priors.

Let the input image be divided into an $S \times S$ grid. Each grid cell i predicts B bounding boxes and C class probabilities. The output tensor $Y \in \mathbb{R}^{S \times S \times (B \cdot 5 + C)}$ is defined such that for each box $j \in \{1, \ldots, B\}$ in cell i:

$$\mathbf{y}_{i,j} = [x, y, w, h, \text{conf}]^T .$$

Coordinate parameterization in v1:

- $x, y \in [0, 1]$: Relative to the bounds of the grid cell.
- $w, h \in [0, 1]$: Relative to the **entire image size**.

The loss function for width and height attempts to stabilize learning for small objects by predicting square roots:

$$\mathcal{L}_{\text{coord}}^{(v1)} \propto \sum_{i,j} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right]. \quad (26.1)$$

Limitations:

- Direct prediction of (w, h) proved unstable, as the variance of object scales is large.
- Coarse grid resolution (7×7) led to poor localization of small objects.
- Strong coupling between classification and localization (per-cell class distribution).

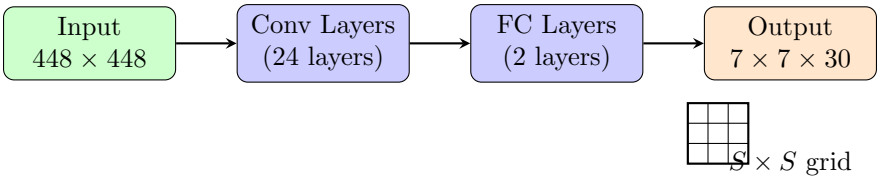

Figure 26.1: YOLOv1 Architecture: Single-scale, fully-connected detection head. Output is $7 \times 7 \times (B \cdot 5 + C)$ where $B = 2$ boxes and $C = 20$ classes (Pascal VOC).

26.1.2 YOLOv2: Introduction of Anchor Priors (2016–2017)

YOLOv2 (and YOLO9000) shifted to an **anchor-based** approach to decouple the scale of the object from the network's output magnitude.

Let $\{(p_{w,k}, p_{h,k})\}_{k=1}^{K}$ be a set of pre-defined anchor box dimensions (priors) derived via K-means clustering on the training set.

The network now predicts offsets (t_x, t_y, t_w, t_h) rather than direct coordinates. The transformation to obtain the bounding box (b_x, b_y, b_w, b_h) is:

$$b_x = \sigma(t_x) + c_x, \tag{26.2}$$
$$b_y = \sigma(t_y) + c_y, \tag{26.3}$$
$$b_w = p_{w,k} \cdot e^{t_w}, \tag{26.4}$$
$$b_h = p_{h,k} \cdot e^{t_h}, \tag{26.5}$$

where (c_x, c_y) is the top-left corner of the grid cell, and $\sigma(\cdot)$ is the sigmoid function.

Mathematical implication: The network now learns log-space scaling factors $t_w = \ln(b_w/p_w)$. This bounds the gradients and prevents the "unstable gradient" problem of v1, since e^{t_w} is strictly positive.

Additional innovations:

- **Batch normalization** on all convolutional layers for stable optimization.
- Higher-resolution input (416×416) and better backbone (Darknet-19).
- Multi-scale training: randomly vary input size every few iterations.

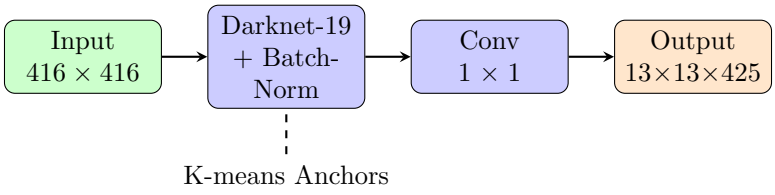

Figure 26.2: YOLOv2 Architecture: Anchor-based predictions with convolutional head (no FC layers). Uses 5 anchors per cell, output $13 \times 13 \times (5 \cdot (5 + 80)) = 13 \times 13 \times 425$ for COCO.

26.1.3 YOLOv3: Multi-Scale Logistic Regression (2018)

YOLOv3 addressed the problem of detecting small objects by making predictions at three different scales (feature pyramid). Mathematically, the prediction tensor exists at scales $s \in \{8, 16, 32\}$ (strides).

Multi-scale predictions:

- Three detection heads on feature maps of sizes 13×13, 26×26, 52×52 (for 416×416 input).
- Each head predicts B anchors per cell (typically $B = 3$).
- Top-down pathway similar to FPN: coarse semantic features are upsampled and fused with finer-resolution features.

A key change in v3 is the **classification formulation**. Instead of a softmax across classes (which assumes mutual exclusivity), v3 uses independent logistic classifiers:

$$P(\text{Class}_c \mid \text{Object}) = \sigma(z_c) = \frac{1}{1 + e^{-z_c}}. \tag{26.6}$$

The classification loss changes from Categorical Cross-Entropy (CCE) to a sum of Binary Cross-Entropies (BCE):

$$\mathcal{L}_{\text{cls}}^{(v3)} = -\sum_{c=1}^{C} \left[y_c \log(\hat{y}_c) + (1 - y_c) \log(1 - \hat{y}_c) \right]. \tag{26.7}$$

Mathematical significance: This allows for multi-label classification (e.g., an object can be both "Woman" and "Person"), altering the probabilistic assumption of the model.

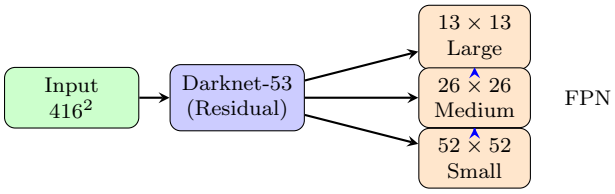

Figure 26.3: YOLOv3 Architecture: Multi-scale detection with FPN-like feature pyramid. Predictions at 3 scales detect objects of different sizes.

26.1.4 YOLOv4/v5: CSPNet and Gradient Flow Optimization (2020–)

While keeping the anchor formulation of v2/v3, modern variants (v4, v5, and beyond) fundamentally changed the backbone architecture to optimize gradient flow using **Cross-Stage Partial (CSP)** connections.

Let $x \in \mathbb{R}^{H\times W\times C}$ be the input feature map to a dense block. CSPNet splits x into two parts along the channel dimension, $x = [x_1, x_2]$:

- x_1 goes directly to the end of the block (skip connection).
- x_2 goes through the computational block $F(\cdot)$.

The output is formed by transition T:

$$y = T\left(\text{Concat}(x_1, F(x_2))\right). \tag{26.8}$$

Mathematical implication: This structure ensures that the gradient $\frac{\partial \mathcal{L}}{\partial x_1}$ does not pass through the non-linearities of F, preserving feature reuse and reducing the number of FLOPs by roughly 50% while maintaining receptive field size. This backbone design (CSPDarknet) is detailed in Section 26.3.

Additional innovations include:

- **PAN (Path Aggregation Network)**: bottom-up and top-down feature aggregation.
- **CIoU/DIoU losses**: improved localization based on complete IoU metrics.
- **Mosaic augmentation**: training with 4 images simultaneously.

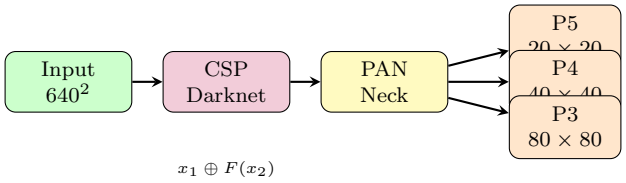

Figure 26.4: YOLOv4/v5 Architecture: CSPDarknet backbone with PAN neck for bidirectional feature aggregation. Three detection heads at different scales.

26.1.5 YOLOv7–v9 and YOLOX: Modern Variants

Recent YOLO-family models introduce further improvements along three axes:

1. **Architecture tweaks:** More efficient backbones and necks (e.g., E-ELAN in YOLOv7, decoupled heads, improved PAN/FPN variants).
2. **Label assignment:** Dynamic label assignment (e.g., SimOTA in YOLOX), advanced IoU-based losses.
3. **Deployment efficiency:** Nano/Tiny variants, quantization-aware training, TensorRT-friendly designs.

From a **mathematical viewpoint**, these remain within the same general framework:

- Convolutional backbone $\rightarrow$ multi-scale feature maps.
- Per-cell, per-anchor predictions of (t_x, t_y, t_w, t_h), objectness, and per-class scores.

This is exactly the formulation formalized in the following sections of this chapter.

26.1.6 Summary: Mathematical Evolution of YOLO

Version	Key Mathematical Change	Impact
v1 $\rightarrow$ v2	Direct (w, h) $\rightarrow$ log-offset e^{t_w}	Stable gradients, positive-only
v2 $\rightarrow$ v3	Softmax $\rightarrow$ Sigmoid per class	Multi-label classification
v3 $\rightarrow$ v4/v5	Dense $\rightarrow$ CSP backbone	Better gradient flow

26.2 Single-Shot vs. Two-Stage Detection

Two-stage (e.g., Faster R-CNN):

1. Region Proposal Network (RPN): generate candidate boxes.
2. Classification head: refine and classify each proposal.

Accurate but slower; not suitable for real-time applications.

Single-shot (YOLO, SSD):

1. Single forward pass: predict boxes and classes.
2. Trade some accuracy for speed.

Enables real-time inference on CPU/mobile.

26.3 YOLO Architecture

26.3.1 Backbone: CSPDarknet

YOLO uses a variant of Darknet backbone with Cross-Stage Partial (CSP) connections:

- 5 stages, progressive downsampling (stride 2 at each stage).
- Skip connections within stages reduce computation.
- Output: feature maps at multiple scales.

26.3.2 Neck: Path Aggregation Network (PAN)

Multi-scale feature fusion:

$$\text{FPN construction} \rightarrow \text{bottom-up pathway} \rightarrow \text{PAN output}. \tag{26.9}$$

26.3.3 Head: Detection Predictions

For each spatial location in the feature map, predict:

$$\mathbf{p} = [t_x, t_y, t_w, t_h, \text{objectness}, c_1, \ldots, c_K], \tag{26.10}$$

where:

$$t_x, t_y : \text{offset to anchor center}, \tag{26.11}$$
$$t_w, t_h : \text{log-scale adjustment to anchor size}, \tag{26.12}$$
$$\text{objectness} : P(\text{object exists}), \tag{26.13}$$
$$c_k : P(\text{class } k | \text{object}). \tag{26.14}$$

26.4 Grid-Cell Prediction Scheme

YOLO divides the image into an $S \times S$ grid. Each cell predicts B bounding boxes.

26.4.1 Example: 1313 grid, 416416 input

Grid cell size:

$$\text{cell\_size} = \frac{416}{13} \approx 32 \text{ pixels}. \tag{26.15}$$

Predictions per cell: Assume $K = 80$ classes, $B = 3$ boxes:

$$\text{predictions per cell} = B \times (5 + K) = 3 \times 85 = 255. \tag{26.16}$$

Total output:

$$\text{output shape} = (13, 13, 255) = (13, 13, 3 \times 85). \tag{26.17}$$

Total predictions:

$$13 \times 13 \times 3 = 507 \text{ boxes} \times 85 \text{ values} = 43,095 \text{ predictions}. \tag{26.18}$$

26.4.2 Coordinate transformation

Raw network outputs (logits) are transformed to bounding boxes. For a cell at (i, j) with box index b:

$$b_x = \sigma(t_x) + j, \tag{26.19}$$
$$b_y = \sigma(t_y) + i, \tag{26.20}$$
$$b_w = p_w \exp(t_w), \tag{26.21}$$
$$b_h = p_h \exp(t_h), \tag{26.22}$$

where p_w, p_h are anchor dimensions, and σ is sigmoid. Multiply by grid cell size s to get image coordinates:

$$\text{box\_coords} = (b_x \cdot s, b_y \cdot s, b_w, b_h). \tag{26.23}$$

26.5 YOLO Loss Function

26.5.1 Components

The loss combines localization, objectness, and classification:

$$\mathcal{L} = \lambda_{\text{box}}\mathcal{L}_{\text{box}} + \lambda_{\text{obj}}\mathcal{L}_{\text{obj}} + \lambda_{\text{cls}}\mathcal{L}_{\text{cls}}. \tag{26.24}$$

26.5.2 Box regression loss

Only compute for boxes with IoU $> \tau_{\text{thresh}}$ with ground truth:

$$\mathcal{L}_{\text{box}} = \frac{1}{N_{\text{obj}}} \sum_{i,j,b} \mathbb{1}_{ij}^{\text{obj}} \left[(\sqrt{b_w} - \sqrt{w^*})^2 + (\sqrt{b_h} - \sqrt{h^*})^2 \right]. \tag{26.25}$$

Square root scaling gives equal weight to small and large boxes.

26.5.3 Objectness loss

$$\mathcal{L}_{\text{obj}} = \frac{1}{N_{\text{bg}}} \sum_{i,j,b} \left[\mathbb{1}_{ij}^{\text{obj}} (o_{ij,b} - 1)^2 + \lambda_{\text{noobj}} \mathbb{1}_{ij}^{\text{noobj}} (o_{ij,b} - 0)^2 \right], \tag{26.26}$$

where $\lambda_{\text{noobj}} \approx 0.5$ down-weights background boxes (most cells contain no object).

26.5.4 Classification loss

Only for cells with objects:

$$\mathcal{L}_{\text{cls}} = \frac{1}{N_{\text{obj}}} \sum_{i,j} \mathbb{1}_{ij}^{\text{obj}} \sum_{k} (c_k - c_k^*)^2, \tag{26.27}$$

where c_k^* is ground truth (one-hot).

26.6 Forward Pass: 416416 Image

26.6.1 Step-by-step

1. Input: $X \in \mathbb{R}^{416 \times 416 \times 3}$.

2. Backbone (CSPDarknet):

$$\text{Stage } 1 : 416 \to 208 \text{ (stride 2)}, \quad 32 \text{ channels}, \tag{26.28}$$
$$\text{Stage } 2 : 208 \to 104 \text{ (stride 2)}, \quad 64 \text{ channels}, \tag{26.29}$$
$$\text{Stage } 3 : 104 \to 52 \text{ (stride 2)}, \quad 128 \text{ channels}, \tag{26.30}$$
$$\text{Stage } 4 : 52 \to 26 \text{ (stride 2)}, \quad 256 \text{ channels}, \tag{26.31}$$
$$\text{Stage } 5 : 26 \to 13 \text{ (stride 2)}, \quad 512 \text{ channels}. \tag{26.32}$$

Output: $F_5 \in \mathbb{R}^{13 \times 13 \times 512}$.

3. Neck (PAN): Fuse features from multiple scales. Output: $\{P_3, P_4, P_5\}$ at scales $\{52 \times 52, 26 \times 26, 13 \times 13\}$.

4. Heads (Detection): For each scale, apply detection head:

$$\text{Head at } P_5 : (13 \times 13 \times 512) \to (13 \times 13 \times 255), \tag{26.33}$$
$$\text{Head at } P_4 : (26 \times 26 \times 256) \to (26 \times 26 \times 255), \tag{26.34}$$
$$\text{Head at } P_3 : (52 \times 52 \times 128) \to (52 \times 52 \times 255). \tag{26.35}$$

26.6.2 Post-processing

1. Flatten predictions: $(13 \times 13 + 26 \times 26 + 52 \times 52) \times 3 = 10,647 + 2,028 + 507 = 13,182$ boxes.
2. Filter by objectness confidence (e.g., > 0.5).
3. Apply NMS with IoU threshold (e.g., 0.5).
4. Return top N detections (e.g., 100).

Result: Typically ~ 100 final predictions after post-processing.

Chapter 27

DETR: Detection with Transformers

DETR (DEtection TRansformer) introduces a paradigm shift: reformulate detection as a set prediction problem using Transformers with bipartite matching.

27.1 Paradigm Shift: Grids to Queries

Traditional (YOLO): Grid cells multiple anchors per cell NMS final boxes.

DETR: Transformer queries direct box + class prediction bipartite matching final boxes.

Key difference: no NMS; unique predicted boxes via query mechanism.

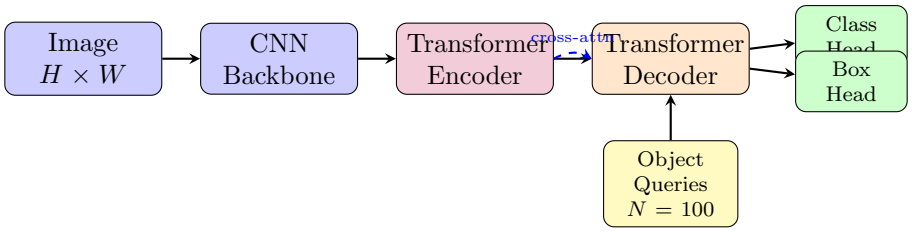

Figure 27.1: DETR Architecture: End-to-end detection with Transformer. Learnable object queries attend to CNN features via cross-attention. No NMS required.

27.2 DETR Architecture

27.2.1 Component 1: CNN Backbone

Extract feature map from image (e.g., ResNet-50):

$$\mathbf{f} \in \mathbb{R}^{H' \times W' \times C}, \quad (H', W') = \frac{1}{32}(H, W). \tag{27.1}$$

Example: For 480640 input:

$$\mathbf{f} \in \mathbb{R}^{15 \times 20 \times 2048}. \tag{27.2}$$

27.2.2 Component 2: Transformer Encoder

Flatten and embed feature map:

$$\mathbf{f}_{\text{flat}} \in \mathbb{R}^{300\times 2048}, \quad (300 = 15 \times 20). \tag{27.3}$$

Project to embedding dimension d:

$$\mathbf{e} = \mathbf{f}_{\text{flat}}\mathbf{W}_e + \mathbf{b}_e \in \mathbb{R}^{300\times d}. \tag{27.4}$$

Add positional encodings:

$$\mathbf{x}_{\text{enc}} = \mathbf{e} + \text{PE}(\text{positions}) \in \mathbb{R}^{300\times d}. \tag{27.5}$$

Pass through 6-layer Transformer encoder:

$$\mathbf{h}_{\text{enc}} = \text{TransformerEncoder}(\mathbf{x}_{\text{enc}}) \in \mathbb{R}^{300\times d}. \tag{27.6}$$

27.2.3 Component 3: Transformer Decoder

Initialize N learnable **object queries**:

$$\mathbf{q}_0, \ldots, \mathbf{q}_{N-1} \in \mathbb{R}^d, \quad N = 100. \tag{27.7}$$

Apply 6-layer Transformer decoder with cross-attention to encoder output:

$$\mathbf{h}_{\text{dec}} = \text{TransformerDecoder}(\mathbf{q}, \mathbf{h}_{\text{enc}}) \in \mathbb{R}^{N\times d}. \tag{27.8}$$

27.2.4 Component 4: Prediction Heads

Class head: Predict class logits for each query:

$$\mathbf{c} = \mathbf{h}_{\text{dec}}\mathbf{W}_c + \mathbf{b}_c \in \mathbb{R}^{N\times(K+1)}, \tag{27.9}$$

where K is number of object classes, plus 1 for "no object" (background).

Box head: Predict bounding box coordinates:

$$\mathbf{b} = \text{MLP}(\mathbf{h}_{\text{dec}}) \in \mathbb{R}^{N\times 4}. \tag{27.10}$$

Output: $(N, K+1)$ class scores and $(N, 4)$ box coordinates.

27.3 Bipartite Matching (Hungarian Algorithm)

27.3.1 Problem formulation

Given N predictions and M ground truth boxes ($M \leq N$), find optimal assignment:

$$\pi^* = \arg\min_{\pi\in\text{Perm}(N)} \sum_{i=1}^{M} \text{Cost}(y_i, \hat{y}_{\pi(i)}), \tag{27.11}$$

where Cost combines classification and box regression losses.

27.3.2 Cost function

$$\text{Cost}(y, \hat{y}) = -P_{\text{correct}}(\hat{y}) + \lambda \cdot L_{\text{box}}(y, \hat{y}), \tag{27.12}$$

where:

$$P_{\text{correct}}(\hat{y}) = \hat{p}_c \text{ if } c \text{ is correct class, else } 1 - \hat{p}_c, \tag{27.13}$$

and L_{box} combines L1 and GIoU losses.

27.3.3 Hungarian algorithm (simplified)

Algorithm 4 Hungarian Algorithm (Simplified)

Input: Cost matrix $C \in \mathbb{R}^{N \times M}$
Initialize matched pairs $\mathcal{M} = \emptyset$
Repeat until all ground truth matched or no improvement:
Find minimum element C_{ij} not yet in $\mathcal{M}$
Add (i, j) to $\mathcal{M}$ (predict i matches ground truth j)
Remove row i and column j
Return $\mathcal{M}$

27.3.4 Numerical example

Setup: 3 predictions, 2 ground truths.
Cost matrix (lower is better):

$$C = \begin{bmatrix} 0.5 & 1.2 \\ 0.8 & 0.3 \\ 1.0 & 0.6 \end{bmatrix} \tag{27.14}$$

Trace:

1. Min cost: $C_{21} = 0.3$ (pred 2, GT 1). Assign: $(2, 1)$.
2. Remove row 2, col 1. Remaining:

$$C' = \begin{bmatrix} 0.5 \\ 1.0 \end{bmatrix} \tag{27.15}$$

Min: $C'_{10} = 0.5$ (pred 1, GT 0). Assign: $(1, 0)$.

3. Pred 3 unmatched. Can be assigned to "no object."

Result: $(1, 0)$ and $(2, 1)$.

27.4 DETR Forward Pass: 480640 Image

27.4.1 Step-by-step

1. Input: $X \in \mathbb{R}^{480 \times 640 \times 3}$.

2. ResNet-50 backbone: Stride 32 downsampling:

$$H' = 480/32 = 15, \quad W' = 640/32 = 20. \tag{27.16}$$

Output: $\mathbf{f} \in \mathbb{R}^{15\times20\times2048}$.

3. Feature flattening:

$$\mathbf{f}_{\text{flat}} \in \mathbb{R}^{300\times2048}. \tag{27.17}$$

4. Embedding and positional encoding: Let $d = 256$:

$$\mathbf{e} = \mathbf{f}_{\text{flat}}\mathbf{W}_e \in \mathbb{R}^{300\times256}, \quad \mathbf{x}_{\text{enc}} = \mathbf{e} + \text{PE} \in \mathbb{R}^{300\times256}. \tag{27.18}$$

5. Encoder:

$$\mathbf{h}_{\text{enc}} = \text{TransformerEncoder}(\mathbf{x}_{\text{enc}}) \in \mathbb{R}^{300\times256}. \tag{27.19}$$

6. Decoder (100 queries):

$$\mathbf{q} = [\mathbf{q}_1, \ldots, \mathbf{q}_{100}]^T \in \mathbb{R}^{100\times256}. \tag{27.20}$$

$$\mathbf{h}_{\text{dec}} = \text{TransformerDecoder}(\mathbf{q}, \mathbf{h}_{\text{enc}}) \in \mathbb{R}^{100\times256}. \tag{27.21}$$

7. Class head (80 COCO classes + 1 background):

$$\mathbf{c} \in \mathbb{R}^{100\times81}. \tag{27.22}$$

8. Box head:

$$\mathbf{b} \in \mathbb{R}^{100\times4}. \tag{27.23}$$

27.4.2 Post-processing

1. For each of 100 predictions, take max class score (excluding background if desired).
2. Filter by confidence threshold (e.g., > 0.5).
3. Return top predictions (typically all 100 are kept as unique).

Result: 100 detected objects (exact number depends on confidence threshold).

27.5 YOLO vs. DETR Comparison

Aspect	YOLO	DETR
Architecture	CNN backbone + grid head	CNN + Transformer
Speed	Fast	Moderate
Accuracy (mAP)	Good	Excellent
Small object detection	Weak	Strong
Training time	Fast (∼24-48 hrs)	Slow (∼500 hrs)
Anchor-free	Yes	Yes
NMS required	Yes	No
Real-time capable	Yes	Depends on HW

Chapter 28

RT-DETR: Real-Time Detection Transformer

RT-DETR bridges the gap between the speed of YOLO-based detectors and the end-to-end capabilities of DETR. While standard DETR suffers from slow convergence and high computational cost due to its heavy Transformer encoder, RT-DETR introduces an efficient hybrid encoder and a query selection mechanism that eliminates the need for NMS while maintaining real-time performance.

28.1 Architecture Overview

The architecture consists of three main components: a backbone, an efficient hybrid encoder, and a Transformer decoder with auxiliary prediction heads.

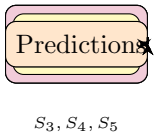

Figure 28.1: RT-DETR Architecture: Efficient hybrid encoder with AIFI (intra-scale attention on S_5 only) and CCFF (cross-scale fusion). Query selection provides high-quality initial proposals.

28.1.1 Backbone and Multi-scale Features

Let the input image be $\mathbf{I} \in \mathbb{R}^{H \times W \times 3}$. The backbone (typically ResNet or HGNet) produces a set of multi-scale feature maps $\{S_3, S_4, S_5\}$ with strides $\{8, 16, 32\}$ respectively.

$$S_i \in \mathbb{R}^{H/2^i \times W/2^i \times C_i}, \quad i \in \{3, 4, 5\}. \tag{28.1}$$

28.2 Efficient Hybrid Encoder

The core innovation of RT-DETR is replacing the fully coupled Transformer encoder with a hybrid structure that separates intra-scale interaction and cross-scale fusion.

28.2.1 AIFI: Attention-based Intra-scale Feature Interaction

Standard DETR flattens all multi-scale features into a single sequence, causing quadratic complexity $\mathcal{O}((\sum H_i W_i)^2)$. RT-DETR observes that high-level features (S_5) contain the most semantic information. Therefore, self-attention is applied *only* to S_5.

Let $\mathbf{F}_5 \in \mathbb{R}^{N_5 \times C}$ be the flattened projection of S_5, where $N_5 = \frac{H}{32} \cdot \frac{W}{32}$. The AIFI output $\mathbf{F}'_5$ is computed as:

$$\mathbf{F}'_5 = \text{MHA}(\text{Query} = \mathbf{F}_5, \text{Key} = \mathbf{F}_5, \text{Value} = \mathbf{F}_5), \tag{28.2}$$

where MHA denotes Multi-Head Attention. This reduces computational complexity significantly compared to full-scale attention.

28.2.2 CCFF: Cross-scale Feature-fusion Module

After AIFI, the features $\{S_3, S_4, \mathbf{F}'_5\}$ are fused using a path similar to PANet (Path Aggregation Network), but with Transformer-based fusion blocks. Let $f_{fusion}(\cdot)$ be the fusion block (typically consisting of RepConv layers). The fusion process is:

$$\mathbf{H}_5 = \mathbf{F}'_5 \tag{28.3}$$

$$\mathbf{H}_4 = f_{fusion}(\text{Concat}(S_4, \text{Upsample}(\mathbf{H}_5))) \tag{28.4}$$

$$\mathbf{H}_3 = f_{fusion}(\text{Concat}(S_3, \text{Upsample}(\mathbf{H}_4))) \tag{28.5}$$

The final encoder output consists of the flattened and concatenated sequence of these fused features:

$$\mathbf{E}_{enc} = \text{Concat}(\text{proj}(\mathbf{H}_3), \text{proj}(\mathbf{H}_4), \text{proj}(\mathbf{H}_5)) \in \mathbb{R}^{L \times d}, \tag{28.6}$$

where $\text{proj}(\cdot)$ denotes flattening and linear projection to dimension d.

28.3 Uncertainty-Minimal Query Selection

Unlike standard DETR which uses static learnable query embeddings $\mathbf{q} \in \mathbb{R}^{N_q \times d}$, RT-DETR selects the initial object queries directly from the encoder features. This acts as a learnable, end-to-end region proposal mechanism.

28.3.1 Selection Mechanism

Let $\mathbf{E}_{enc} \in \mathbb{R}^{L \times d}$ be the encoder features. A classification head predicts the "objectness" (or class probability) for each feature token:

$$\hat{\mathbf{s}} = \sigma(\mathbf{E}_{enc}\mathbf{W}_{cls}) \in \mathbb{R}^{L \times K}, \tag{28.7}$$

where K is the number of classes. We identify the indices of the top N_q scoring tokens:

$$\mathcal{I}_{top} = \text{topk}(\max_k(\hat{\mathbf{s}}), N_q). \tag{28.8}$$

The initial decoder queries (content part) are then gathered from the encoder features:

$$\mathbf{Q}^{(0)}_{content} = \text{Gather}(\mathbf{E}_{enc}, \mathcal{I}_{top}) \in \mathbb{R}^{N_q \times d}. \tag{28.9}$$

The corresponding positional queries $\mathbf{Q}^{(0)}_{pos}$ are the bounding box coordinates predicted from these selected features (treated as initial anchors).

28.3.2 Mathematical Interpretation

This selection creates a prior for the decoder. If we view the decoder as refining an initial guess:

$$\text{Box}_{final} = \text{Box}_{init} + \Delta\text{Box}_{decoder}, \tag{28.10}$$

RT-DETR ensures that Box_{init} is already a high-quality proposal derived from the CNN backbone, reducing the optimization difficulty for the Transformer decoder layers.

28.4 Decoder and Loss

28.4.1 Decoder with IoU-aware Query Selection

The decoder follows the standard Transformer architecture, taking $\mathbf{Q}^{(0)}$ as input. However, to resolve the discrepancy between classification score and localization quality, the training objective is modified.

28.4.2 Loss Function

The loss combines classification and box regression, computed via bipartite matching. RT-DETR uses *Varifocal Loss* (VFL) for classification to weigh examples by their IoU quality. For a ground truth class c and predicted score p, with IoU score q (between predicted box and GT):

$$\mathcal{L}_{VFL}(p, q) = \begin{cases} -q(q\log(p) + (1-q)\log(1-p)) & \text{if } q > 0 \text{ (foreground)} \\ -\alpha p^{\gamma}\log(1-p) & \text{if } q = 0 \text{ (background)} \end{cases} \tag{28.11}$$

This loss encourages the selected queries $\mathbf{Q}^{(0)}$ to have high scores only if they also have high IoU potential, aligning the "uncertainty" of classification with localization accuracy.

28.5 Summary: RT-DETR vs. YOLO vs. DETR

Feature	**YOLOv8**	**DETR**	**RT-DETR**
Architecture	CNN-only	CNN + Transformer	Hybrid
Inference	Static Grid	Set Prediction	Set Prediction
Post-processing	NMS Required	NMS-free	NMS-free
Attention Scope	None (Conv)	Global (all scales)	Intra-scale (S_5) only
Query Init	Fixed Anchors	Learnable Embeddings	Encoder Features

RT-DETR mathematically proves that the NMS-free property of DETR can be preserved without the computational burden of global attention over all feature scales, by confining self-attention to the highest semantic level (AIFI).

Chapter 29

Vision Transformers for Detection

While DETR introduced Transformers to detection, it still relied on a CNN backbone. Vision Transformers (ViT) replace the entire architecture with pure Transformer blocks, treating images as sequences of patches. This chapter covers ViT fundamentals and their application to detection.

29.1 From Image Classification to Detection

29.1.1 ViT for classification

ViT divides an image into non-overlapping patches, embeds each patch, and processes them as a sequence.

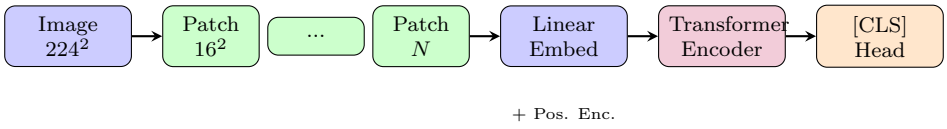

Figure 29.1: Vision Transformer (ViT) Architecture: Image is split into patches, embedded, and processed by Transformer encoder. The [CLS] token output is used for classification.

Patch embedding: For an image $X \in \mathbb{R}^{H \times W \times 3}$ with patch size $P \times P$:

$$\text{Number of patches} = \frac{H}{P} \times \frac{W}{P}. \tag{29.1}$$

Example: 224224 image, 1616 patches

$$\text{Patches} = \frac{224}{16} \times \frac{224}{16} = 14 \times 14 = 196 \text{ patches}. \tag{29.2}$$

Each patch is flattened to a vector and linearly embedded:

$$\mathbf{x}_i = \mathbf{E} \cdot \text{flatten}(\text{patch}_i) + \mathbf{b}, \quad \mathbf{x}_i \in \mathbb{R}^d, \tag{29.3}$$

where $\mathbf{E} \in \mathbb{R}^{d \times (3 \cdot P^2)}$ is the embedding matrix.

29.1.2 ViT architecture for classification

1. Add learnable class token $\mathbf{c}_{cls} \in \mathbb{R}^d$ at the beginning.
2. Add positional encodings to all patch embeddings (including class token).
3. Pass through L Transformer encoder layers.
4. Apply classification head to the class token output.

Forward pass (classification):

$$\mathbf{X} = [\mathbf{c}_{cls}; \mathbf{x}_1; \ldots; \mathbf{x}_N] + \mathbf{PE} \in \mathbb{R}^{(N+1)\times d}, \tag{29.4}$$

$$\mathbf{H} = \text{TransformerEncoder}(\mathbf{X}) \in \mathbb{R}^{(N+1)\times d}, \tag{29.5}$$

$$\textbf{logits} = \mathbf{W}_c \mathbf{H}_{[0]} + \mathbf{b}_c \in \mathbb{R}^K, \tag{29.6}$$

where $\mathbf{H}_{[0]}$ is the class token output.

29.2 ViT-Det: Hierarchical Vision Transformer

Pure ViT lacks multi-scale features (all patches at same resolution). ViT-Det (and Swin Transformer) introduce hierarchy by progressively merging patches.

29.2.1 Hierarchical structure

Stage 1: Patch partition, embedding. Output: $H_1 \times W_1 \times d_1$.

Stage 2: Merge 2×2 patches (1/2 spatial, 4 channel reduction).

$$(H_1, W_1, d_1) \to (H_1/2, W_1/2, 4d_1). \tag{29.7}$$

Apply linear projection to normalize dimension:

$$\mathbf{x}' = \text{LinearProj}([x_{2i}, x_{2i+1}]_{2j}, x_{2j+1}]) \in \mathbb{R}^{d_2}. \tag{29.8}$$

Stage 3 & 4: Repeat merging.

29.2.2 Swin Transformer blocks

Swin introduces **shifted windows** to reduce computation:

$$\text{Attention} = \text{softmax}\left(\frac{QK^T}{\sqrt{d}} + \text{bias}\right) V. \tag{29.9}$$

Instead of global attention (quadratic in sequence length), compute attention within local windows:

$$\text{Complexity: Local window} \sim (P^2) \text{ per position, vs. } (N^2) \text{ global.} \tag{29.10}$$

Window size: Typically 7×7 or 8×8.

29.2.3 Swin-T architecture (Tiny variant)

Stage	Blocks	Spatial Res.	Dim	Window Size
1	2	56×56	96	7×7
2	2	28×28	192	7×7
3	6	14×14	384	7×7
4	2	7×7	768	7×7

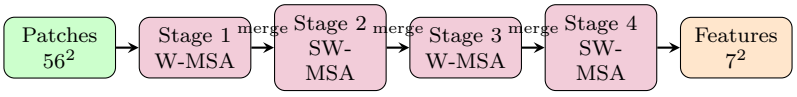

Figure 29.2: Swin Transformer Architecture: Hierarchical stages with patch merging. W-MSA = Window Multi-head Self-Attention, SW-MSA = Shifted Window MSA.

29.3 Inductive Bias: CNN vs. ViT

29.3.1 CNN inductive biases

- **Locality:** Convolution operates on local patches. Receptive field grows with depth.
- **Translation equivariance:** Shifting input shifts output by same amount.
- **Weight sharing:** Same filter across all spatial locations.

These biases reduce parameters and improve generalization, especially with small datasets.

29.3.2 ViT inductive biases

- **Tokenization:** Patch-based representation; fixed at input.
- **Permutation invariance:** Attention is permutation-invariant over patches (mitigated by positional encodings).
- **No weight sharing:** Attention weights depend on query and key; not tied across positions.

ViT requires more data to learn spatial structure but can capture long-range dependencies more easily.

29.3.3 Performance implications

Small data ($<$ 1M images): CNN outperforms ViT.

Large data ($>$ 10M images): ViT often superior; learns effective spatial priors.

Transfer learning: Pre-trained ViT on ImageNet-21K transfers better to downstream tasks than CNN in many cases.

29.4 Patch Embedding Process

29.4.1 Detailed computation

Input: $X \in \mathbb{R}^{H\times W\times 3}$.

Reshape into patches:

$$\text{patches} \in \mathbb{R}^{(H/P)\times(W/P)\times(P^2\cdot 3)}. \tag{29.11}$$

Example: 224224 RGB, 1616 patches

$$\text{patches} \in \mathbb{R}^{14\times 14\times 768}, \quad (768 = 16\times 16\times 3). \tag{29.12}$$

Flatten patches:

$$\mathbf{x}_{\text{flat}} \in \mathbb{R}^{196\times 768}. \tag{29.13}$$

Linear projection:

$$\mathbf{x}_{\text{embed}} = \mathbf{x}_{\text{flat}}\mathbf{W}_e + \mathbf{b}_e \in \mathbb{R}^{196\times d}, \tag{29.14}$$

where $d = 768$ (ViT-Base).

29.4.2 Positional encodings

Standard sinusoidal or learnable encodings:

$$\text{PE}_{(i,2j)} = \sin\left(\frac{i}{10000^{2j/d}}\right), \quad \text{PE}_{(i,2j+1)} = \cos\left(\frac{i}{10000^{2j/d}}\right). \tag{29.15}$$

Or learnable:

$$\mathbf{PE}_i \in \mathbb{R}^d, \quad \text{trained with other parameters.} \tag{29.16}$$

29.4.3 Sequence with class token

$$\mathbf{S} = [\mathbf{c}_{cls}; \mathbf{x}_1; \ldots; \mathbf{x}_{196}] + \mathbf{PE} \in \mathbb{R}^{197\times 768}. \tag{29.17}$$

29.5 Computational Complexity: YOLO vs. DETR vs. ViT

29.5.1 FLOPs and memory

YOLO (416416 input):

$$\text{Backbone (CSPDarknet):} \quad \sim 15 \text{ GFLOPs}, \tag{29.18}$$

$$\text{Total:} \quad \sim 20 \text{ GFLOPs}. \tag{29.19}$$

DETR (480640 input):

$$\text{ResNet-50 backbone:} \quad \sim 100 \text{ GFLOPs}, \tag{29.20}$$
$$\text{Transformer encoder:} \quad \sim 50 \text{ GFLOPs}, \tag{29.21}$$
$$\text{Transformer decoder:} \quad \sim 20 \text{ GFLOPs}, \tag{29.22}$$
$$\text{Total:} \quad \sim 170 \text{ GFLOPs}. \tag{29.23}$$

Swin-T Detection (480640 input):

$$\text{Swin backbone:} \quad \sim 80 \text{ GFLOPs}, \tag{29.24}$$
$$\text{Detection head:} \quad \sim 10 \text{ GFLOPs}, \tag{29.25}$$
$$\text{Total:} \quad \sim 90 \text{ GFLOPs}. \tag{29.26}$$

29.5.2 Throughput (images/sec at typical batch size)

Model	GPU (V100)	TPU	CPU
YOLO-v8s	300		10
DETR	20	50	0.5
Swin-T	40	80	2

29.5.3 Summary: Speed vs. Accuracy

$$\text{YOLO} \gg \text{Swin/DETR} \gg \text{ViT-Det (full)}. \tag{29.27}$$

$$\text{Accuracy: ViT-Det} > \text{Swin} \approx \text{DETR} > \text{YOLO}. \tag{29.28}$$

Choose based on latency constraints and dataset size.

Chapter 30

Implementation and Integration

This chapter provides complete numerical walkthroughs and pseudo-code for training loops.

30.1 Complete Numerical Walkthrough: YOLO

30.1.1 Toy dataset and model

Dataset: - Image size: 416×416. - Number of classes: 2 (person, car). - 5 training images with annotations.

Model: - Simplified YOLO with one detection head at 1313. - 3 boxes per cell, 255 output channels.

30.1.2 Forward pass (single image)

Input: $X \in \mathbb{R}^{416\times416\times3}$ (person at center, car at bottom-right).
Backbone: Output $F_5 \in \mathbb{R}^{13\times13\times512}$.
Head: $(13, 13, 512) \to (13, 13, 255)$.
Example cell output (cell [6, 6]):

$$\text{output}[6, 6, :] = [t_x, t_y, t_w, t_h, o_1, \ldots, o_3, c_1, \ldots, c_3, \ldots], \tag{30.1}$$

where first 5 values per box 3 boxes + 80 class scores 3 boxes = 255 values.

Raw outputs (example):

$$\text{Box 1:} \quad t_x = 0.1, t_y = 0.2, t_w = 0.5, t_h = 0.6, o = 0.9, \tag{30.2}$$

$$\text{Classes:} \quad c_{\text{person}} = 0.8, c_{\text{car}} = 0.1. \tag{30.3}$$

30.1.3 Decoding predictions

Apply sigmoid to offsets, exp to dimensions:

$$b_x = \sigma(0.1) + 6 = 0.525 + 6 = 6.525, \tag{30.4}$$

$$b_y = \sigma(0.2) + 6 = 0.550 + 6 = 6.550, \tag{30.5}$$

$$b_w = p_w \exp(0.5) = 16 \cdot 1.649 = 26.38, \tag{30.6}$$

$$b_h = p_h \exp(0.6) = 16 \cdot 1.822 = 29.15, \tag{30.7}$$

where p_w, p_h are anchor dimensions, and multiply by cell size (32) to get image coordinates:

$$\text{Box in image:} \quad (6.525 \times 32, 6.550 \times 32, 26.38, 29.15) \tag{30.8}$$

$$= (209, 210, 26, 29) \text{ in pixels.} \tag{30.9}$$

30.1.4 Loss computation

Assume ground truth: person box at approximately $(208, 208, 80, 80)$ (center of cell [6, 6]).

Localization loss:

$$L_{\text{box}} = (\sqrt{26} - \sqrt{80})^2 + (\sqrt{29} - \sqrt{80})^2 \approx 2.8. \tag{30.10}$$

Objectness loss:

$$L_{\text{obj}} = (0.9 - 1)^2 + (\text{background boxes}). \tag{30.11}$$

Classification loss:

$$L_{\text{cls}} = (0.8 - 1)^2 + (0.1 - 0)^2 = 0.04 + 0.01 = 0.05. \tag{30.12}$$

30.1.5 Total loss

$$L = 5 \cdot 2.8 + 1.0 \cdot 0.01 + 1.0 \cdot 0.05 = 14 + 0.01 + 0.05 = 14.06. \tag{30.13}$$

30.2 Complete Numerical Walkthrough: DETR

30.2.1 Setup

Same toy dataset. DETR with: - ResNet-50 backbone. - 100 queries. - 6 encoder/decoder layers. - $d = 256$ embedding dimension.

30.2.2 Backbone output

Input: $480 \times 640 \times 3$. ResNet-50 stride-32: $15 \times 20 \times 2048$.

30.2.3 Encoder

Flatten: $300 \times 2048 \to 300 \times 256$ (projection).

Add positional encodings (sinusoidal):

$$\text{PE}_{(i,2j)} = \sin\left(\frac{i}{10000^{2j/256}}\right), \quad i \in [0, 299], j \in [0, 127]. \tag{30.14}$$

Pass through 6-layer Transformer encoder.

30.2.4 Decoder

100 learnable queries:

$$\mathbf{q}_1, \ldots, \mathbf{q}_{100} \in \mathbb{R}^{256}. \tag{30.15}$$

Cross-attention to encoder output:

$$\text{Attn} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{256}}\right)\mathbf{V}. \tag{30.16}$$

30.2.5 Prediction heads

Class head output:

$$\mathbf{c} \in \mathbb{R}^{100\times 3}, \quad \text{(2 classes + 1 background)}. \tag{30.17}$$

Example predictions:

$$\text{Query 10:} \quad \mathbf{c}_{10} = [0.1, 0.8, 0.1] \quad \text{(person=0.8)}, \tag{30.18}$$

$$\text{Query 47:} \quad \mathbf{c}_{47} = [0.9, 0.05, 0.05] \quad \text{(background)}. \tag{30.19}$$

Box head output:

$$\mathbf{b} \in \mathbb{R}^{100\times 4}. \tag{30.20}$$

Example:

$$\mathbf{b}_{10} = [0.4, 0.4, 0.3, 0.3] \quad \text{(normalized XYXY)}. \tag{30.21}$$

30.2.6 Bipartite matching

Cost for query 10 and ground truth person:

$$\text{Cost} = -0.8 + 5 \times L_{\text{GIoU}}. \tag{30.22}$$

Assume $L_{\text{GIoU}} \approx 0.2$:

$$\text{Cost} = -0.8 + 1.0 = 0.2. \tag{30.23}$$

Hungarian algorithm assigns query 10 to person, query 47 to background, etc.

30.2.7 Loss computation

Focal loss (for class imbalance):

$$L_{\text{cls}} = -(1 - p_t)^\gamma \log(p_t), \tag{30.24}$$

where $p_t = 0.8$ (predicted prob. of correct class person), $\gamma = 2$:

$$L_{\text{cls}} = -(1 - 0.8)^2 \log(0.8) = -0.04 \times (-0.223) = 0.009. \tag{30.25}$$

GIoU loss:

$$L_{\text{bbox}} = 1 - \text{GIoU} = 1 - 0.8 = 0.2. \tag{30.26}$$

Total:

$$L = 5 \cdot 0.2 + 2 \cdot 0.009 = 1.0 + 0.018 = 1.018. \tag{30.27}$$

Algorithm 5 Detection Training Loop

Initialize model f_θ (YOLO/DETR/ViT).
Initialize optimizer (SGD/Adam) with learning rate η.
for each epoch **do**
  Shuffle training dataset.
  for each mini-batch $\mathcal{B}$ **do**
    $\mathcal{L} \leftarrow 0$ (accumulate loss)
    for each image X, annotations Y in $\mathcal{B}$ **do**
      $\hat{Y} \leftarrow f_\theta(X)$ (forward pass)
      Assign targets (NMS for YOLO, Hungarian for DETR).
      Compute loss $\ell(Y, \hat{Y})$ (box + class + confidence)
      $\mathcal{L} \leftarrow \mathcal{L} + \ell(Y, \hat{Y})$
    end for
    $\mathcal{L} \leftarrow \mathcal{L}/|\mathcal{B}|$ (average)
    $g \leftarrow \nabla_\theta \mathcal{L}$ (backprop)
    $\theta \leftarrow \theta - \eta \cdot g$ (update)
  end for
  Validate on validation set, compute mAP.
  Adjust learning rate (e.g., decay if mAP plateaus).
end for

30.3 Training Loop Pseudo-code

30.4 Inference Optimization

30.4.1 Quantization (INT8)

Reduce model size and increase speed:

$$x_{\text{int8}} = \text{round}\left(\frac{x_{\text{fp32}} - \min}{(\max - \min)/255}\right). \tag{30.28}$$

Typical speedup: 2-4 with minimal accuracy loss.

30.4.2 Knowledge distillation

Train a small student model to mimic a large teacher:

$$L_{\text{distill}} = \alpha L_{\text{task}} + (1-\alpha)\text{KL}(p_{\text{teacher}} \| p_{\text{student}}). \tag{30.29}$$

30.4.3 Pruning

Remove low-magnitude weights:

$$w' = \begin{cases} w & \text{if } |w| > \tau, \\ 0 & \text{otherwise.} \end{cases} \tag{30.30}$$

Can reduce model size by 50-90

30.4.4 Batch size effects

Batch Size	Throughput (img/s)	Latency (ms/img)
1	10	100
4	35	114
16	100	160
64	200	320

Trade-off: Larger batches increase throughput but latency per image rises due to queueing.

30.5 Benchmark Comparison

30.5.1 COCO test-dev results (top models)

Model	mAP	Speed (fps)	Params (M)
YOLO-v8n (nano)	37.3	80	2.7
YOLO-v8s (small)	44.9	60	11.2
YOLO-v8m (medium)	50.2	30	25.9
YOLO-v8l (large)	52.9	15	43.7
DETR	45.0	20	41.3
DETR (ResNet-101)	47.3	12	60.2
Swin-T DETR	49.7	15	38.7
Swin-L DETR	56.7	5	153.6

Chapter 31

Architecture Comparison and Decision Trees

31.1 Comparison Matrix

Criterion	YOLO	DETR	ViT-Det
Real-time on GPU	Yes	Partial	No
Real-time on CPU	No	No	No
Real-time on edge	Yes	No	No
Small object detection	Weak	Strong	Strong
High-resolution images	Slow	OK	Very slow
Dense object scenes	Weak	OK	Good
Training time	24h	500h	1000h+
Convergence speed	Fast	Slow	Very slow
Hyperparameter sensitivity	Medium	High	Very high
Anchor-dependent	No	No	No
NMS required	Yes	No	No
Post-processing overhead	5-10ms	¡1ms	¡1ms
Interpretability	Low	Medium	Medium
Visualization (attention)	Hard	Easy	Easy

31.2 Decision Tree

31.3 Use Case Recommendations

31.3.1 Autonomous driving

Requirements: Real-time (30+ fps), high-resolution (1920×1080), robust small object (pedestrians, signs).

Recommendation: **YOLO (large variant)** or **YOLO-v8l** with TensorRT optimization.

Rationale: Real-time critical. YOLO's speed trade-off acceptable.

Algorithm 6 Detection Architecture Decision Tree

Question 1: Do you need real-time inference (> 30 fps)?
if YES **then**
→ **Use YOLO** (if hardware available).
if NO suitable GPU **then**
→ **Use YOLO-nano** or **TensorRT**.
end if
else
→ Proceed to Q2.
end if
Question 2: Do you have $> 100K$ labeled images?
if YES **then**
→ Consider **DETR** or **Swin-DETR** for better accuracy.
else
→ Fine-tune **pre-trained YOLO** or **DETR** (transfer learning).
end if
Question 3: Are small objects critical?
if YES **then**
→ **Prefer DETR/Swin** (better small object detection).
else
→ **YOLO sufficient** with multi-scale features.
end if
Question 4: Do you need interpretability (attention maps)?
if YES **then**
→ **Use DETR or ViT** (easy attention visualization).
else
→ **YOLO is simpler**.
end if

31.3.2 Surveillance (CCTV)

Requirements: 24/7 operation, person/intrusion detection, moderate latency tolerance ($\sim$ 1s).

Recommendation: **DETR** with batch inference.

Rationale: Latency not critical; superior accuracy for dense scenes.

31.3.3 Medical imaging (X-ray/CT anomaly detection)

Requirements: Highest accuracy, can accept slow inference.

Recommendation: **Swin-L DETR** or **ViT-Det** with test-time augmentation.

Rationale: Accuracy paramount. ViT's long-range understanding valuable.

31.3.4 Mobile/IoT (on-device inference)

Requirements: Low latency (< 100ms), minimal memory (< 50MB), low power.

Recommendation: **YOLO-nano** + quantization (INT8).

Rationale: Only option for edge deployment.

31.4 Performance Trade-offs

31.4.1 Speed vs. Accuracy Pareto frontier

$$\text{Efficiency} = \frac{\text{mAP}}{\text{Latency (ms)}} \times 100. \tag{31.1}$$

Model	mAP	Efficiency
YOLO-v8s	44.9	0.75
YOLO-v8m	50.2	1.67
DETR	45.0	0.27
Swin-T DETR	49.7	0.90

YOLO-v8m achieves best efficiency; recommended for most production scenarios.

Chapter 32

Vision-Based Robot Control

Integrating vision detection with robot control enables autonomous manipulation and navigation.

32.1 Perception-to-Control Pipeline

32.1.1 System architecture

$$\text{Camera} \rightarrow \text{Detection (YOLO/DETR)} \rightarrow \text{3D Localization} \rightarrow \text{Planning (ACT/Diffusion)} \rightarrow \text{Robot Exec} \tag{32.1}$$

32.1.2 Information flow

1. **Frame capture:** RGB-D camera at f_{fps} Hz.
2. **Object detection:** Predict bounding boxes in 2D.
3. **Depth estimation:** Map 2D boxes to 3D coordinates using depth map.
4. **Policy inference:** Generate action sequence via trained policy (ACT, Diffusion Policy, VLA).
5. **Execution:** Send joint angles/velocities to robot.

32.2 2D Detection to 3D Localization

32.2.1 Camera model (pinhole)

Intrinsic camera matrix:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \tag{32.2}$$

where f_x, f_y are focal lengths (pixels), (c_x, c_y) is principal point.

32.2.2 2D to 3D projection

For a pixel (u, v) with depth d from depth map:

$$X = \frac{(u - c_x) \cdot d}{f_x}, \tag{32.3}$$

$$Y = \frac{(v - c_y) \cdot d}{f_y}, \tag{32.4}$$

$$Z = d. \tag{32.5}$$

Example:

- Camera: RealSense D455 (RGB-D).
- Detection: Object bounding box $[u_1, v_1, u_2, v_2] = [200, 150, 300, 250]$ pixels.
- Depth at center: $d = 0.5$ meters.
- Focal length: $f_x = f_y = 600$ pixels.
- Principal point: $(c_x, c_y) = (320, 240)$.

Center of bbox: $(u, v) = (250, 200)$.
3D position:

$$X = \frac{(250 - 320) \times 0.5}{600} = \frac{-35}{600} = -0.058 \text{ m}, \tag{32.6}$$

$$Y = \frac{(200 - 240) \times 0.5}{600} = \frac{-20}{600} = -0.033 \text{ m}, \tag{32.7}$$

$$Z = 0.5 \text{ m}. \tag{32.8}$$

Object pose: $\mathbf{p}_{\text{obj}} = [-0.058, -0.033, 0.5]^T$ meters (relative to camera).

32.2.3 Camera-to-robot frame transformation

Known rigid transformation $T_{\text{cam}\to\text{robot}}$ (calibrated beforehand):

$$\mathbf{p}_{\text{robot}} = T_{\text{cam}\to\text{robot}} \cdot \mathbf{p}_{\text{cam}} = R\mathbf{p}_{\text{cam}} + \mathbf{t}, \tag{32.9}$$

where $R \in SO(3)$ is rotation, $\mathbf{t} \in \mathbb{R}^3$ is translation.

Example:

$$T_{\text{cam}\to\text{robot}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0.3 \\ 0 & 1 & 0 & 0.2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(rotation + offset in robot frame).} \tag{32.10}$$

$$\mathbf{p}_{\text{robot}} = \begin{bmatrix} -0.058 \\ 0.033 + 0.3 \\ 0.5 + 0.2 \end{bmatrix} = \begin{bmatrix} -0.058 \\ 0.333 \\ 0.7 \end{bmatrix} \text{ meters.} \tag{32.11}$$

32.3 Pick-and-Place Robot: Complete Example

32.3.1 Scenario

- Robot: 6-DOF robotic arm (UR10e).
- Task: Pick red cube, place in bin.
- Sensors: RGB-D camera mounted on gripper.
- Policy: Trained Action Chunking Transformer (ACT).

32.3.2 Frame 0: Initial state

RGB image: 640×480.

Detection: YOLO inference.

$$\text{Detected objects: Red cube [200, 150, 300, 250] (bbox)}, \quad \text{Confidence: 0.95}. \tag{32.12}$$

3D localization:

$$\mathbf{p}_{\text{cube}} = [0.2, 0.15, 0.3]^T \text{ (meters, robot frame)}. \tag{32.13}$$

32.3.3 Action generation (ACT)

Input to policy:

$$\mathbf{s} = [\text{current joint angles}, \text{gripper state}, \text{cube position}]. \tag{32.14}$$

Policy outputs action sequence ($T_a = 16$ action steps):

$$\mathbf{a}_1, \ldots, \mathbf{a}_{16} \in \mathbb{R}^7 \quad \text{(6 DOF angles + gripper)}. \tag{32.15}$$

32.3.4 Trajectory (step-by-step)

Step 0-4 (Move to cube):

$$\mathbf{a}_1 = [\Delta\theta_1, \ldots, \Delta\theta_6, \text{open gripper}], \tag{32.16}$$
$$\mathbf{a}_2 = [\Delta\theta_1, \ldots, \Delta\theta_6, \text{open gripper}], \tag{32.17}$$
$$\vdots \tag{32.18}$$

Step 5 (Reach):

$$\mathbf{a}_5 = [0, 0, 0, 0, 0, 0, \text{open gripper}]. \tag{32.19}$$

Step 6 (Close gripper):

$$\mathbf{a}_6 = [0, 0, 0, 0, 0, 0, \text{close gripper}]. \tag{32.20}$$

Step 7-14 (Lift and move):

$$\mathbf{a}_7, \ldots, \mathbf{a}_{14} = [\text{move to bin}, \text{gripper closed}]. \tag{32.21}$$

Step 15-16 (Place):

$$\mathbf{a}_{15}, \mathbf{a}_{16} = [0, 0, 0, 0, 0, 0, \text{open gripper}]. \tag{32.22}$$

32.3.5 Execution

For each action $\mathbf{a}_i$:

1. Compute joint angles: $\boldsymbol{\theta}_i = \text{IK}(\mathbf{a}_i)$ (inverse kinematics).
2. Send to robot controller (typically 50-100 Hz control loop).
3. Receive feedback: joint angles, gripper state.
4. Repeat until all actions executed.

32.4 Multi-Object Tracking (MOT)

When objects move or scene is dynamic, tracking associates detections across frames.

32.4.1 Problem formulation

Input: Detections at each frame t:

$$D_t = \{\mathbf{b}_1^{(t)}, \ldots, \mathbf{b}_{N_t}^{(t)}\}. \tag{32.23}$$

Output: Tracked objects with consistent IDs:

$$T = \{(id_i, \mathbf{b}_i^{(t_1)}, \ldots, \mathbf{b}_i^{(t_K)})\}. \tag{32.24}$$

32.4.2 Hungarian algorithm for matching

Match detections in frame t to tracks from frame $t-1$ based on IoU or centroid distance.

Cost matrix:

$$C_{ij} = -\text{IoU}(\text{det}_i^{(t)}, \text{track}_j^{(t-1)}) + \lambda \cdot d(\text{centroid}_i, \text{centroid}_j). \tag{32.25}$$

Solving: Hungarian algorithm finds minimum-cost assignment.

32.4.3 Numerical example (3 detections, 2 tracks)

	Track 0 (history)	**Track 1 (history)**
Det 0	Cost = 0.2	Cost = 1.0
Det 1	Cost = 0.8	Cost = 0.3
Det 2	Cost = 1.5	Cost = 0.7

Optimal assignment (Hungarian):

1. Min: Det 0 Track 0 (cost 0.2).
2. Min: Det 1 Track 1 (cost 0.3).
3. Det 2 unmatched New track or discard.

32.5 Latency Analysis

Total latency breakdown (per frame):

Component	Latency (ms)
Camera capture	5
Detection (YOLO-v8s)	15
3D localization	2
Action generation (ACT)	20
IK + trajectory planning	10
Robot execution (sending command)	5
Total	**57 ms**
Max frequency	**$\sim$17 Hz**

Practical control loop: Typically 10 Hz (100 ms) for smooth robot motion.

32.6 Integration with VLMs

Vision-Language Models (e.g., GPT-4V, Flamingo) can provide semantic understanding beyond detection.

32.6.1 Multi-modal pipeline

$$\text{Image} \rightarrow \text{VLM} \rightarrow \text{Text prompt} \rightarrow \text{Policy} \rightarrow \text{Actions}. \tag{32.26}$$

Example:

1. VLM: “I see a red cube on the table and a blue bin to the right.”
2. Policy: Fine-tuned to handle high-level commands (“pick red, place blue”).
3. Execution: Same as before, but commanded by natural language.

Chapter 33

Conclusion and Future Directions

33.1 Summary of Vision Architectures

We have covered:

- **Chapter 21:** CNN foundations (convolution, ResNet-50, FPN).
- **Chapter 22:** Detection basics (anchors, IoU, NMS, mAP).
- **Chapter 23:** YOLO (single-shot, real-time, grid-cell predictions).
- **Chapter 24:** DETR (Transformer detection, bipartite matching).
- **Chapter 25:** Vision Transformers (ViT, hierarchical, Swin).
- **Chapter 26:** Implementation (training loops, inference optimization).
- **Chapter 27:** Architecture comparison and decision trees.
- **Chapter 28:** Robot control integration (perception-to-action).

33.2 When to Use Each

- **YOLO:** Real-time applications, mobile/edge, speed-critical.
- **DETR:** High accuracy, moderate latency, fine-grained detection.
- **ViT:** Unlimited compute, maximum accuracy, semantic richness.

33.3 Emerging Trends

33.3.1 Efficient vision (MobileViT, EfficientDet)

Compression and mobile-friendly designs are pushing real-time detection to phones and IoT.

33.3.2 Unified models (YOLO-World, OWLv2)

Open-vocabulary detection: detect any object by text description, without retraining.

33.3.3 End-to-end learning

Combining vision, language, and control in single models (e.g., VLA frameworks).

33.4 Further Reading

- YOLO series: Redmon et al. (2016–2023).
- DETR: Carion et al. (2020), "End-to-End Object Detection with Transformers."
- Vision Transformers: Dosovitskiy et al. (2020), "An Image is Worth 16x16 Words."
- Swin Transformer: Liu et al. (2021), "Swin Transformer: Hierarchical Vision Transformer."
- Robotic manipulation with vision: ACT (Zhao et al., 2023), Diffusion Policy (Chi et al., 2023).

33.5 Beyond Vision: Unified Architectures Across Modalities

The architectures studied in this part—convolutional backbones, Transformer encoders, and detection heads—share the same mathematical core as the large language models discussed in Part IV. This section briefly outlines the unifying principles.

33.5.1 The Common Mathematical Framework

At the most abstract level, both vision and language models implement the same computation:

$$f_\theta : \mathbb{R}^{n_{\text{in}}} \to \mathbb{R}^{n_{\text{out}}}, \quad f_\theta = f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(1)}, \tag{33.1}$$

where each $f^{(\ell)}$ is a composition of affine maps and nonlinearities, optimized by SGD and backpropagation.

The key difference lies in the input representation:

- **Vision:** Input tensor $X \in \mathbb{R}^{H \times W \times C}$ (image pixels).
- **Language:** Input tensor $X \in \mathbb{R}^{T \times d}$ (token embeddings).

33.5.2 From Patch Embeddings to Token Embeddings

The Vision Transformer (ViT, Chapter 26) and BERT (Chapter 31) share the same encoder architecture. The primary distinction is:

Model	Input	Embedding
BERT	Token IDs $\in \mathcal{V}$	$E_{\text{tok}} \cdot \text{one-hot}(x_t)$
ViT	Image patches $\in \mathbb{R}^{P^2C}$	$E_{\text{patch}} \cdot x_p$

After embedding, both models apply identical Transformer encoder blocks with multi-head self-attention and feed-forward networks.

33.5.3 Detection and Generation as Different Output Heads

- **DETR** (Chapter 25): Transformer encoder-decoder with set prediction loss for object detection.
- **GPT** (Chapter 32): Transformer decoder with causal attention for next-token prediction.
- **T5** (Chapter 35): Transformer encoder-decoder with span corruption for text-to-text generation.

The architectural differences (encoder-only, decoder-only, encoder-decoder) determine the attention mask and output head, but the core attention mechanism remains mathematically identical.

33.5.4 Implications for Practice

This unified view has practical consequences:

1. **Transfer learning:** Pre-trained vision encoders (ViT) and language encoders (BERT) can be combined in multimodal models.
2. **Shared optimization:** Adam, learning rate schedules, and regularization techniques apply across modalities.
3. **Architecture search:** Innovations in one domain (e.g., sparse attention in Longformer) transfer to others (e.g., efficient ViT variants).

The following Part IV chapters formalize the mathematical details of specific language model architectures, building on the same foundations established in Parts I–III.

Part III

Large Language Models and Foundation Architectures

Chapter 34

BERT: Bidirectional Encoder with Masked Language Modeling

34.1 Architecture Overview

Figure 34.1 illustrates the BERT architecture, which uses a stack of Transformer encoder layers with bidirectional self-attention.

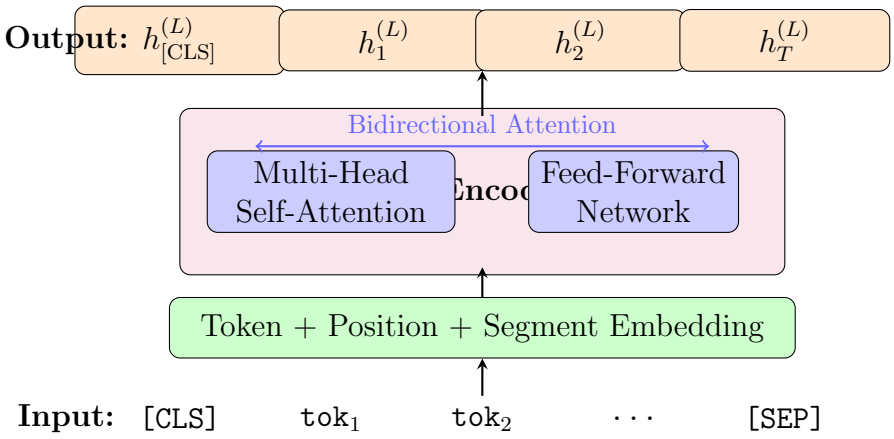

Figure 34.1: BERT Architecture: Encoder-only Transformer with bidirectional self-attention. All tokens can attend to all other tokens in the sequence.

34.1.1 Intuitive Understanding

Why Bidirectional? Traditional language models (like GPT) read text left-to-right, predicting each word based only on previous words. BERT revolutionized this by allowing each token to "see" both its left and right context simultaneously. This is like reading a sentence by looking at the whole sentence at once, rather than word by word.

Key Insight: When humans understand language, we naturally use context from both directions. For example, in "The bank by the river was steep," understanding "bank" requires seeing "river" which comes later. BERT captures this bidirectional understanding.

[CLS] Token: A special token prepended to every input. After processing through all layers, its representation aggregates information from the entire sequence, making it

ideal for classification tasks.

Trade-off: Bidirectional attention means BERT cannot generate text autoregressively (it would see future tokens). Thus, BERT excels at understanding tasks (classification, NER, QA) but not text generation.

34.2 Notation and Input Representation

Let $\mathcal{V}$ be the vocabulary (with $|\mathcal{V}|$ tokens), $T_{\max}$ the maximum sequence length, and d_{model} the model dimension. An input sequence is denoted as

$$(x_1, x_2, \ldots, x_T), \quad x_t \in \mathcal{V},\ 1 \le T \le T_{\max}.$$

We define the token embedding matrix

$$E_{\text{tok}} \in \mathbb{R}^{d_{\text{model}} \times |\mathcal{V}|},$$

the position embedding matrix

$$E_{\text{pos}} \in \mathbb{R}^{d_{\text{model}} \times T_{\max}},$$

and the segment embedding matrix (to distinguish sentence A/B)

$$E_{\text{seg}} \in \mathbb{R}^{d_{\text{model}} \times 2}.$$

Token x_t is represented as a one-hot vector $\text{one\_hot}(x_t) \in \{0,1\}^{|\mathcal{V}|}$, and its embedding is

$$e_t^{\text{tok}} = E_{\text{tok}}\, \text{one\_hot}(x_t) \in \mathbb{R}^{d_{\text{model}}}.$$

The embedding for position t is

$$e_t^{\text{pos}} = E_{\text{pos}}[:, t] \in \mathbb{R}^{d_{\text{model}}},$$

and for segment label $s_t \in \{0,1\}$ (sentence A/B)

$$e_t^{\text{seg}} = E_{\text{seg}}[:, s_t] \in \mathbb{R}^{d_{\text{model}}}.$$

These are summed to form the input to the Transformer:

$$h_t^{(0)} = e_t^{\text{tok}} + e_t^{\text{pos}} + e_t^{\text{seg}} \in \mathbb{R}^{d_{\text{model}}}, \quad t = 1, \ldots, T.$$

Stacking the column vectors $h_t^{(0)}$ vertically:

$$H^{(0)} = \begin{bmatrix} (h_1^{(0)})^\top \\ \vdots \\ (h_T^{(0)})^\top \end{bmatrix} \in \mathbb{R}^{T \times d_{\text{model}}}.$$

For batch processing with padding, with mini-batch size B and maximum length $T_{\max}$:

$$H_{\text{batch}}^{(0)} \in \mathbb{R}^{B \times T_{\max} \times d_{\text{model}}},$$

using a mask matrix

$$M_{\text{pad}} \in \{0, -\infty\}^{B \times 1 \times T_{\max}}$$

to ignore padding positions.

34.3 Encoder Architecture: Bidirectional Self-Attention

The BERT encoder consists of L Transformer blocks. For each layer $\ell = 1, \dots, L$, self-attention output is computed from input $H^{(\ell-1)} \in \mathbb{R}^{T \times d_{\text{model}}}$.

34.3.1 Multi-Head Self-Attention

For each head $h = 1, \dots, H$, define the query, key, and value matrices as

$$Q^{(\ell,h)} = H^{(\ell-1)} W_Q^{(\ell,h)} \in \mathbb{R}^{T \times d_k},$$
$$K^{(\ell,h)} = H^{(\ell-1)} W_K^{(\ell,h)} \in \mathbb{R}^{T \times d_k},$$
$$V^{(\ell,h)} = H^{(\ell-1)} W_V^{(\ell,h)} \in \mathbb{R}^{T \times d_v},$$

where

$$W_Q^{(\ell,h)}, W_K^{(\ell,h)} \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_V^{(\ell,h)} \in \mathbb{R}^{d_{\text{model}} \times d_v},$$

and typically $d_k = d_v = d_{\text{model}}/H$.

The score matrix

$$S^{(\ell,h)} = \frac{Q^{(\ell,h)} (K^{(\ell,h)})^\top}{\sqrt{d_k}} \in \mathbb{R}^{T \times T}$$

is computed. In BERT, bidirectional attention is allowed between all tokens, so only padding is masked. Extending the padding mask $M_{\text{pad}}^{(\text{1D})} \in \{0, -\infty\}^T$:

$$M_{ij}^{(\ell)} = \begin{cases} 0 & \text{if neither is padding,} \\ -\infty & \text{if at least one is padding,} \end{cases}$$

and

$$\tilde{S}^{(\ell,h)} = S^{(\ell,h)} + M^{(\ell)}.$$

Attention weights are computed via softmax:

$$A^{(\ell,h)} = \text{softmax}\big(\tilde{S}^{(\ell,h)}\big) \in \mathbb{R}^{T \times T}, \quad A_{ij}^{(\ell,h)} = \frac{\exp(\tilde{S}_{ij}^{(\ell,h)})}{\sum_{k=1}^{T} \exp(\tilde{S}_{ik}^{(\ell,h)})}.$$

Head output:

$$Y^{(\ell,h)} = A^{(\ell,h)} V^{(\ell,h)} \in \mathbb{R}^{T \times d_v}.$$

All heads are concatenated and transformed by output projection $W_O^{(\ell)} \in \mathbb{R}^{H d_v \times d_{\text{model}}}$:

$$Y^{(\ell)} = \text{Concat}\big(Y^{(\ell,1)}, \dots, Y^{(\ell,H)}\big) W_O^{(\ell)} \in \mathbb{R}^{T \times d_{\text{model}}}.$$

34.3.2 Residual Connection and Layer Normalization

The output of the self-attention sublayer is

$$\tilde{H}^{(\ell)} = \text{LN}\big(H^{(\ell-1)} + Y^{(\ell)}\big),$$

where LN is LayerNorm applied position-wise. For a vector $x \in \mathbb{R}^{d_{\text{model}}}$:

$$\mu(x) = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} x_i, \quad \sigma^2(x) = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} (x_i - \mu(x))^2,$$

$$\text{LN}(x) = \gamma \odot \frac{x - \mu(x)\mathbf{1}}{\sqrt{\sigma^2(x) + \varepsilon}} + \beta,$$

where $\gamma, \beta \in \mathbb{R}^{d_{\text{model}}}$ are learnable parameters and $\varepsilon > 0$ is a small constant.

34.3.3 Position-Wise Feed-Forward Network

The FFN at each position t is:

$$\mathrm{FFN}(u_t) = W_2^{(\ell)}\,\phi\big(W_1^{(\ell)}u_t + b_1^{(\ell)}\big) + b_2^{(\ell)},$$
$$W_1^{(\ell)} \in \mathbb{R}^{d_{\mathrm{ff}}\times d_{\mathrm{model}}}, \quad W_2^{(\ell)} \in \mathbb{R}^{d_{\mathrm{model}}\times d_{\mathrm{ff}}},$$

where ϕ is a nonlinear function such as ReLU. In matrix form:

$$\mathrm{FFN}(\tilde{H}^{(\ell)}) = \phi\big(\tilde{H}^{(\ell)}W_1^{(\ell)\top} + \mathbf{1}(b_1^{(\ell)})^\top\big)W_2^{(\ell)\top} + \mathbf{1}(b_2^{(\ell)})^\top,$$

where $\mathbf{1} \in \mathbb{R}^T$ is a vector of all ones.

With residual connection and LayerNorm, the layer output is:

$$H^{(\ell)} = \mathrm{LN}\Big(\tilde{H}^{(\ell)} + \mathrm{FFN}\big(\tilde{H}^{(\ell)}\big)\Big).$$

34.4 Pretraining Objective I: Masked Language Modeling

34.4.1 Masking Strategy

From the original sequence $(x_1, \ldots, x_T)$, mask position set $\mathcal{M} \subset \{1, \ldots, T\}$ is sampled randomly (e.g., 15% of total tokens). For $t \in \mathcal{M}$:

- With 80% probability, replace with [MASK];
- With 10% probability, replace with a random token;
- With 10% probability, keep the original token (but still include in loss).

The resulting input sequence $(\tilde{x}_1, \ldots, \tilde{x}_T)$ is passed through the encoder:

$$H^{(L)} = \mathrm{Encoder}(\tilde{x}_{1:T}), \quad h_t^{(L)} \in \mathbb{R}^{d_{\mathrm{model}}}.$$

34.4.2 Token-Level Logits and Probabilities

For masked position $t \in \mathcal{M}$, the output logits are:

$$z_t = W_{\mathrm{MLM}}h_t^{(L)} + b_{\mathrm{MLM}} \in \mathbb{R}^{|\mathcal{V}|},$$
$$p_t = \mathrm{softmax}(z_t), \quad p_t(v) = \frac{\exp(z_{t,v})}{\sum_{u\in\mathcal{V}}\exp(z_{t,u})}.$$

Let the true token be $x_t^\star \in \mathcal{V}$. The loss for a single position is the categorical cross-entropy:

$$\ell_{\mathrm{MLM}}(t) = -\log p_t(x_t^\star).$$

The MLM loss per sequence is:

$$\mathcal{L}_{\mathrm{MLM}} = \frac{1}{|\mathcal{M}|}\sum_{t\in\mathcal{M}}\ell_{\mathrm{MLM}}(t) = -\frac{1}{|\mathcal{M}|}\sum_{t\in\mathcal{M}}\log p_t(x_t^\star).$$

The expected loss (empirical risk) over dataset $\mathcal{D}$ is:

$$\mathcal{L}_{\mathrm{MLM}}(\theta) = \frac{1}{|\mathcal{D}|}\sum_{(x,y)\in\mathcal{D}}\mathbb{E}_{\mathcal{M}\sim\Pi}\left[-\frac{1}{|\mathcal{M}|}\sum_{t\in\mathcal{M}}\log p_t(x_t^\star;\theta)\right],$$

where Π is the masking distribution and θ is the full parameter set of BERT.

34.4.3 Gradient w.r.t. Logits

For position t, we derive the gradient with respect to logits $z_t \in \mathbb{R}^{|\mathcal{V}|}$. Define the one-hot label vector $y_t \in \{0,1\}^{|\mathcal{V}|}$ as:

$$(y_t)_v = \begin{cases} 1 & v = x_t^\star, \\ 0 & \text{otherwise}, \end{cases}$$

then

$$\ell_{\text{MLM}}(t) = -\sum_{v \in \mathcal{V}} (y_t)_v \log p_t(v).$$

By the standard softmax+CCE result:

$$\nabla_{z_t} \ell_{\text{MLM}}(t) = p_t - y_t.$$

Therefore, for the mini-batch average:

$$\nabla_{z_t} \mathcal{L}_{\text{MLM}} = \frac{1}{|\mathcal{M}|}(p_t - y_t).$$

This gradient propagates to the output weights W_{MLM} and hidden representation $h_t^{(L)}$:

$$\nabla_{W_{\text{MLM}}} \mathcal{L}_{\text{MLM}} = \sum_{t \in \mathcal{M}} (p_t - y_t)\,(h_t^{(L)})^\top,$$

$$\nabla_{h_t^{(L)}} \mathcal{L}_{\text{MLM}} = W_{\text{MLM}}^\top (p_t - y_t), \quad t \in \mathcal{M}.$$

34.5 Pretraining Objective II: Next Sentence Prediction

34.5.1 Pair Representation

NSP performs binary classification on a concatenated sequence of two sentences (A, B) using the [CLS] token representation. The input sequence is:

$$[\text{CLS}], A, [\text{SEP}], B, [\text{SEP}]$$

and the output vector at the [CLS] position is denoted $h_{\text{CLS}}^{(L)}$.

Linear classifier:

$$u = W_{\text{NSP}} h_{\text{CLS}}^{(L)} + b_{\text{NSP}} \in \mathbb{R}, \quad q = \sigma(u) = \frac{1}{1 + \exp(-u)},$$

For label $y_{\text{NSP}} \in \{0,1\}$ (1: correct next sentence, 0: random sentence), the loss is:

$$\mathcal{L}_{\text{NSP}} = -\left[y_{\text{NSP}} \log q + (1 - y_{\text{NSP}}) \log(1 - q)\right].$$

34.5.2 Joint Loss

The total loss for a single sample (sentence pair with masked tokens) is:

$$\mathcal{L}_{\text{BERT}} = \mathcal{L}_{\text{MLM}} + \lambda_{\text{NSP}} \mathcal{L}_{\text{NSP}},$$

Dataset average:

$$\min_\theta \frac{1}{|\mathcal{D}|} \sum_{(x,y) \in \mathcal{D}} \mathcal{L}_{\text{BERT}}(x, y; \theta).$$

34.6 Batching, Masks, and Complexity

34.6.1 Attention Mask Matrix

With mini-batch size B and maximum length $T_{\max}$, the batch tensor is:

$$H^{(\ell)} \in \mathbb{R}^{B \times T_{\max} \times d_{\text{model}}}$$

with padding mask $M_{\text{pad}} \in \{0, -\infty\}^{B \times 1 \times T_{\max}}$.

For self-attention, broadcast is applied to the score tensor

$$S \in \mathbb{R}^{B \times H \times T_{\max} \times T_{\max}}$$

as:

$$\tilde{S}_{b,h,i,j} = S_{b,h,i,j} + M_{\text{pad}}[b, 0, j] + M_{\text{pad}}[b, 0, i],$$

so that if either position is PAD, it becomes $-\infty$.

34.6.2 Computational Cost

The attention computation per layer is:

$$O\big(H\, T_{\max}^2 d_k\big),$$

FFN is:

$$O\big(T_{\max} d_{\text{model}} d_{\text{ff}}\big).$$

Total forward pass computation for all L layers:

$$O\big(L(H T_{\max}^2 d_k + T_{\max} d_{\text{model}} d_{\text{ff}})\big),$$

and backpropagation is of the same order.

34.7 Summary

BERT

- constructs contextual representations $h_t^{(L)}$ via bidirectional self-attention encoder (fully connected attention mask),
- uses a pretraining objective combining masked token reconstruction (MLM) and sentence pair prediction (NSP)

to learn general-purpose representations. All of these can be rigorously formulated as token-level log-likelihood maximization problems.

Chapter 35

GPT: Decoder-Only Autoregressive Transformer

35.1 Architecture Overview

Figure 35.1 illustrates the GPT architecture, which uses a stack of Transformer decoder layers with causal (unidirectional) self-attention.

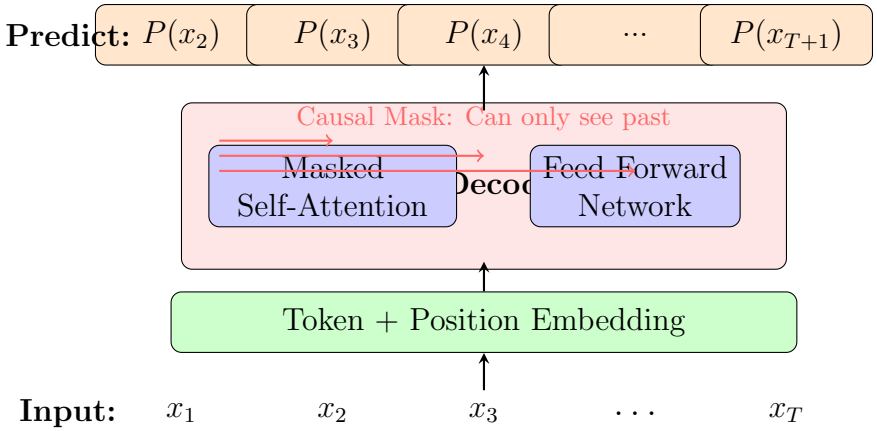

Figure 35.1: GPT Architecture: Decoder-only Transformer with causal (left-to-right) self-attention. Each position can only attend to previous positions.

35.1.1 Intuitive Understanding

Why Causal/Autoregressive? GPT models language as a sequential prediction problem: given all previous words, predict the next word. This mirrors how humans write text—one word at a time, building on what came before.

The Causal Mask: The key mechanism is a triangular attention mask that prevents each position from "peeking" at future tokens. Position 3 can see positions 1 and 2, but not 4, 5, etc. This enables training on entire sequences in parallel while maintaining the autoregressive property.

Why Decoder-Only? Unlike encoder-decoder models (T5), GPT uses only decoder blocks. This simplicity, combined with massive scale, leads to emergent abilities like in-context learning and few-shot reasoning.

Generation Process: At inference time, GPT generates text iteratively:

1. Given prompt tokens, compute all hidden states in parallel
2. Sample next token from the predicted distribution
3. Append sampled token and repeat

Scaling Insight: The GPT family demonstrated that simply scaling model size, data, and compute leads to qualitative improvements in reasoning, knowledge, and generalization ("scaling laws").

35.2 Notation and Factorization of the Language Model

Let $\mathcal{V}$ be the vocabulary with size $|\mathcal{V}|$, and $T_{\max}$ the maximum sequence length. A text is represented as a token sequence

$$x_{1:T} = (x_1, x_2, \ldots, x_T), \quad x_t \in \mathcal{V},\ 1 \le T \le T_{\max}.$$

An autoregressive language model represents the probability distribution via the chain rule:

$$p_\theta(x_{1:T}) = \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t}) \quad (\text{where } x_{<t} = x_1, \ldots, x_{t-1})$$

with parameters θ.

The negative log-likelihood (per sequence) is:

$$\mathcal{L}_{\text{NLL}}(x_{1:T}; \theta) = -\log p_\theta(x_{1:T}) = -\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t}).$$

In practice, training uses input and output sequences shifted by one token. For example, input is $(x_1, \ldots, x_T)$, target is $(x_2, \ldots, x_{T+1})$ (last is EOS):

$$\mathcal{L}_{\text{GPT}}(x_{1:T+1}; \theta) = -\frac{1}{T} \sum_{t=1}^{T} \log p_\theta(x_{t+1} \mid x_{1:t}).$$

In the following, we formulate how this conditional probability $p_\theta(\cdot \mid x_{1:t})$ is computed by the Transformer decoder.

35.3 Token, Position, and Input Embeddings

We use the vocabulary embedding matrix

$$E_{\text{tok}} \in \mathbb{R}^{d_{\text{model}} \times |\mathcal{V}|},$$

and position embedding matrix

$$E_{\text{pos}} \in \mathbb{R}^{d_{\text{model}} \times T_{\max}}.$$

Token x_t is represented as a one-hot vector $\text{one\_hot}(x_t) \in \{0,1\}^{|\mathcal{V}|}$:

$$e_t^{\text{tok}} = E_{\text{tok}}\, \text{one\_hot}(x_t) \in \mathbb{R}^{d_{\text{model}}}, \quad e_t^{\text{pos}} = E_{\text{pos}}[:, t] \in \mathbb{R}^{d_{\text{model}}}.$$

The input embedding is:

$$h_t^{(0)} = e_t^{\text{tok}} + e_t^{\text{pos}} \in \mathbb{R}^{d_{\text{model}}}, \quad t = 1, \ldots, T.$$

In matrix form:

$$H^{(0)} = \begin{bmatrix} (h_1^{(0)})^\top \\ \vdots \\ (h_T^{(0)})^\top \end{bmatrix} \in \mathbb{R}^{T \times d_{\text{model}}}.$$

35.4 Causal Multi-Head Self-Attention

GPT consists of L Transformer decoder blocks, with each layer ℓ:

$$H^{(\ell)} = \text{Block}^{(\ell)}(H^{(\ell-1)}), \quad \ell = 1, \ldots, L.$$

35.4.1 Single Head Self-Attention with Causal Mask

For $H^{(\ell-1)} \in \mathbb{R}^{T \times d_{\text{model}}}$, define the single-head query, key, and value matrices as:

$$Q = H^{(\ell-1)} W_Q, \quad K = H^{(\ell-1)} W_K, \quad V = H^{(\ell-1)} W_V,$$

$$W_Q, W_K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_V \in \mathbb{R}^{d_{\text{model}} \times d_v}.$$

Let row t be q_t, k_t, v_t.

The score matrix:

$$S \in \mathbb{R}^{T \times T}, \quad S_{ij} = \frac{q_i^\top k_j}{\sqrt{d_k}}.$$

The causal mask $M \in \{0, -\infty\}^{T \times T}$ is:

$$M_{ij} = \begin{cases} 0 & j \le i, \\ -\infty & j > i, \end{cases}$$

to prevent attention to future tokens. Masked scores:

$$\tilde{S} = S + M.$$

Attention weights via row-wise softmax:

$$A_{ij} = \frac{\exp(\tilde{S}_{ij})}{\sum_{k=1}^{T} \exp(\tilde{S}_{ik})}, \quad A \in \mathbb{R}^{T \times T}.$$

Single head output:

$$Y = AV \in \mathbb{R}^{T \times d_v}, \quad y_i = \sum_{j=1}^{T} A_{ij} v_j.$$

35.4.2 Multi-Head Attention and Output Projection

Let the number of heads be H. For each head h:

$$Q^{(h)} = H^{(\ell-1)}W_Q^{(h)}, \quad K^{(h)} = H^{(\ell-1)}W_K^{(h)}, \quad V^{(h)} = H^{(\ell-1)}W_V^{(h)},$$

$$Y^{(h)} = \mathrm{Attention}_{\mathrm{causal}}(Q^{(h)}, K^{(h)}, V^{(h)}), \quad Y^{(h)} \in \mathbb{R}^{T\times d_v}.$$

Concatenate and apply linear transformation:

$$Y^{(\ell)} = \mathrm{Concat}\big(Y^{(1)}, \ldots, Y^{(H)}\big)W_O^{(\ell)} \in \mathbb{R}^{T\times d_{\mathrm{model}}},$$

$$W_O^{(\ell)} \in \mathbb{R}^{Hd_v\times d_{\mathrm{model}}}.$$

35.4.3 Residual and Pre/Post-Norm Variants

The standard Post-LN form is:

$$\tilde{H}^{(\ell)} = \mathrm{LN}\big(H^{(\ell-1)} + Y^{(\ell)}\big),$$

$$H^{(\ell)} = \mathrm{LN}\Big(\tilde{H}^{(\ell)} + \mathrm{FFN}\Big(\tilde{H}^{(\ell)}\Big)\Big).$$

Many GPT-style models use the Pre-LN form:

$$\hat{H}^{(\ell)} = H^{(\ell-1)} + \mathrm{MHA}_{\mathrm{causal}}\big(\mathrm{LN}(H^{(\ell-1)})\big),$$

$$H^{(\ell)} = \hat{H}^{(\ell)} + \mathrm{FFN}\Big(\mathrm{LN}(\hat{H}^{(\ell)})\Big).$$

LayerNorm LN for vector $x \in \mathbb{R}^{d_{\mathrm{model}}}$ is:

$$\mathrm{LN}(x) = \gamma \odot \frac{x - \mu(x)\mathbf{1}}{\sqrt{\sigma^2(x) + \varepsilon}} + \beta,$$

$$\mu(x) = \frac{1}{d_{\mathrm{model}}}\sum_{i=1}^{d_{\mathrm{model}}} x_i, \quad \sigma^2(x) = \frac{1}{d_{\mathrm{model}}}\sum_{i=1}^{d_{\mathrm{model}}} (x_i - \mu(x))^2,$$

where $\gamma, \beta \in \mathbb{R}^{d_{\mathrm{model}}}$ are learnable parameters.

35.5 Position-Wise Feed-Forward Network

The FFN at each layer ℓ is applied independently to each position:

$$\mathrm{FFN}^{(\ell)}(u) = W_2^{(\ell)}\,\phi\Big(W_1^{(\ell)}u + b_1^{(\ell)}\Big) + b_2^{(\ell)},$$

$$W_1^{(\ell)} \in \mathbb{R}^{d_{\mathrm{ff}}\times d_{\mathrm{model}}}, \quad W_2^{(\ell)} \in \mathbb{R}^{d_{\mathrm{model}}\times d_{\mathrm{ff}}},$$

where ϕ is ReLU, GELU, etc. In matrix form:

$$\mathrm{FFN}^{(\ell)}(H) = \phi\Big(HW_1^{(\ell)\top} + \mathbf{1}(b_1^{(\ell)})^\top\Big)\,W_2^{(\ell)\top} + \mathbf{1}(b_2^{(\ell)})^\top,$$

where $\mathbf{1} \in \mathbb{R}^T$ is the all-ones vector.

35.6 Output Layer and Conditional Distribution

The final layer output is:

$$H^{(L)} = \begin{bmatrix} (h_1^{(L)})^\top \\ \vdots \\ (h_T^{(L)})^\top \end{bmatrix},$$

and vocabulary logits are:

$$z_t = W_{\text{LM}} h_t^{(L)} + b_{\text{LM}} \in \mathbb{R}^{|\mathcal{V}|},$$

$$p_\theta(x_{t+1} = v \mid x_{1:t}) = \frac{\exp(z_{t,v})}{\sum_{u \in \mathcal{V}} \exp(z_{t,u})}, \quad v \in \mathcal{V}.$$

With weight tying, $W_{\text{LM}} = E_{\text{tok}}^\top$, sharing parameters between embedding and output projection.

35.7 Training Objective and Gradients

35.7.1 Sequence Loss and Per-Token Loss

Loss for input $x_{1:T+1}$:

$$\mathcal{L}_{\text{GPT}}(x_{1:T+1}; \theta) = -\frac{1}{T} \sum_{t=1}^{T} \log p_\theta(x_{t+1} \mid x_{1:t}).$$

Loss at time t:

$$\ell_t(\theta) = -\log p_\theta(x_{t+1}^\star \mid x_{1:t}),$$

With 1-hot label $y_t \in \{0,1\}^{|\mathcal{V}|}$:

$$(y_t)_v = \begin{cases} 1 & v = x_{t+1}^\star, \\ 0 & \text{otherwise}, \end{cases}$$

then

$$\ell_t(\theta) = -\sum_{v \in \mathcal{V}} (y_t)_v \log p_\theta(v \mid x_{1:t}).$$

35.7.2 Gradient w.r.t. Logits and Hidden States

By the standard softmax + cross-entropy result:

$$\nabla_{z_t} \ell_t = p_t - y_t, \quad p_t = p_\theta(\cdot \mid x_{1:t}).$$

Thus for batch average (including normalization factor $1/T$):

$$\nabla_{z_t} \mathcal{L}_{\text{GPT}} = \frac{1}{T}(p_t - y_t).$$

From linear projection $z_t = W_{\text{LM}} h_t^{(L)} + b_{\text{LM}}$:

$$\nabla_{W_{\text{LM}}} \mathcal{L}_{\text{GPT}} = \frac{1}{T} \sum_{t=1}^{T} (p_t - y_t)\, (h_t^{(L)})^\top,$$

$$\nabla_{b_{\mathrm{LM}}} \mathcal{L}_{\mathrm{GPT}} = \frac{1}{T} \sum_{t=1}^{T} (p_t - y_t),$$

$$\nabla_{h_t^{(L)}} \mathcal{L}_{\mathrm{GPT}} = \frac{1}{T} W_{\mathrm{LM}}^{\top} (p_t - y_t).$$

This $\nabla_{h_t^{(L)}} \mathcal{L}_{\mathrm{GPT}}$ is backpropagated through the Transformer decoder.

35.8 Teacher Forcing and Inference

35.8.1 Teacher Forcing During Training

During training, to evaluate the conditional distribution

$$p_\theta(x_{t+1} \mid x_{1:t})$$

the input sequence $(x_1, \ldots, x_t)$ is always given the "ground truth tokens" (teacher forcing). Thus, the likelihood computed during training is:

$$p_\theta(x^\star_{2:T+1} \mid x^\star_{1:T}) = \prod_{t=1}^{T} p_\theta(x^\star_{t+1} \mid x^\star_{1:t}).$$

35.8.2 Autoregressive Generation at Test Time

During generation, model samples are recursively fed back:

$$\begin{aligned} &\text{given } x_1, \ldots, x_t, \\ &\quad p_\theta(x_{t+1} \mid x_{1:t}) = \mathrm{softmax}(W_{\mathrm{LM}} h_t^{(L)} + b_{\mathrm{LM}}), \\ &\quad x_{t+1} \sim p_\theta(\cdot \mid x_{1:t}). \end{aligned}$$

With temperature $\tau > 0$:

$$p_\theta^\tau(v \mid x_{1:t}) = \frac{\exp(z_{t,v}/\tau)}{\sum_u \exp(z_{t,u}/\tau)}.$$

35.9 Perplexity and Evaluation Metric

The average negative log-likelihood per token on test distribution $\mathcal{D}_{\mathrm{test}}$ is:

$$\bar{\ell} = \mathbb{E}_{x_{1:T} \sim \mathcal{D}_{\mathrm{test}}} \left[-\frac{1}{T} \sum_{t=1}^{T} \log_2 p_\theta(x_t \mid x_{<t}) \right],$$

then Perplexity is defined as:

$$\mathrm{PPL} = 2^{\bar{\ell}}.$$

This can be interpreted as the effective vocabulary size representing "how many choices the model is selecting from on average."

35.10 Batching, Causal Mask, and Complexity

35.10.1 Batch-Wise Causal Attention

With mini-batch size B and maximum length $T_{\max}$:

$$H^{(\ell)} \in \mathbb{R}^{B\times T_{\max}\times d_{\text{model}}},$$

for each head:

$$Q, K, V \in \mathbb{R}^{B\times H\times T_{\max}\times d_k},$$

Scores:

$$S_{b,h,i,j} = \frac{1}{\sqrt{d_k}} Q_{b,h,i,:} \cdot K_{b,h,j,:},$$

Mask:

$$M_{i,j} = \begin{cases} 0 & j \le i, \\ -\infty & j > i, \end{cases}$$

broadcast to:

$$\tilde{S}_{b,h,i,j} = S_{b,h,i,j} + M_{i,j}.$$

Attention weights via softmax:

$$A_{b,h,i,j} = \frac{\exp(\tilde{S}_{b,h,i,j})}{\sum_{k=1}^{T_{\max}} \exp(\tilde{S}_{b,h,i,k})},$$

Output:

$$Y_{b,h,i,:} = \sum_{j=1}^{T_{\max}} A_{b,h,i,j} V_{b,h,j,:}.$$

35.10.2 Computational Complexity

Per-layer computation:

$$O\big(BHT_{\max}^2 d_k\big) \quad \text{(Attention)},$$

$$O\big(BT_{\max} d_{\text{model}} d_{\text{ff}}\big) \quad \text{(FFN)}.$$

For all L layers:

$$O\Big(L\big(BHT_{\max}^2 d_k + BT_{\max} d_{\text{model}} d_{\text{ff}}\big)\Big).$$

During inference, KV cache reduces computation at time t to:

$$O\big(BHtd_k + Bd_{\text{model}} d_{\text{ff}}\big),$$

and for generating sequence of length T:

$$O\big(BHT^2 d_k + BTd_{\text{model}} d_{\text{ff}}\big).$$

35.11 Summary

GPT

- expresses $p_\theta(x_t \mid x_{<t})$ via causal masked self-attention,
- defines conditional categorical distribution via softmax from token embedding and output linear projection,
- minimizes negative log-likelihood in mini-batch units,

formulating a rigorous probabilistic model. This framework forms the foundation for scaling laws, in-context learning, and RLHF discussed in later chapters.

Chapter 36

RoBERTa: Robustly Optimized BERT Pretraining

36.1 Architectural Overview

RoBERTa has fundamentally the same Transformer encoder-only structure as BERT, applying bidirectional self-attention to all tokens.

Let $\mathcal{V}$ be the vocabulary, $T_{\max}$ the maximum sequence length, d_{model} the model dimension, and L the number of layers. For input token sequence

$$x_{1:T} = (x_1, \ldots, x_T), \quad x_t \in \mathcal{V},\ 1 \le T \le T_{\max},$$

the encoder output is:

$$H^{(L)} = \begin{bmatrix} (h_1^{(L)})^\top \\ \vdots \\ (h_T^{(L)})^\top \end{bmatrix} \in \mathbb{R}^{T \times d_{\text{model}}}.$$

The essential differences of RoBERTa are:

- Training objective (MLM only with dynamic masking),
- Removal of NSP,
- Training schedule (large batch, longer training, larger corpus).

36.2 Token and Segment Representation

Vocabulary embedding matrix:

$$E_{\text{tok}} \in \mathbb{R}^{d_{\text{model}} \times |\mathcal{V}|},$$

Position embedding:

$$E_{\text{pos}} \in \mathbb{R}^{d_{\text{model}} \times T_{\max}}.$$

Unlike BERT, RoBERTa does not use segment embeddings, treating all as a single sentence (or concatenated sentences).

With one-hot vector of token x_t as $\text{one\_hot}(x_t) \in \{0,1\}^{|\mathcal{V}|}$:

$$e_t^{\text{tok}} = E_{\text{tok}}\,\text{one\_hot}(x_t) \in \mathbb{R}^{d_{\text{model}}}, \quad e_t^{\text{pos}} = E_{\text{pos}}[:, t] \in \mathbb{R}^{d_{\text{model}}}.$$

Input embedding:

$$h_t^{(0)} = e_t^{\text{tok}} + e_t^{\text{pos}} \in \mathbb{R}^{d_{\text{model}}}, \quad t = 1, \ldots, T.$$

36.3 Bidirectional Self-Attention Encoder

For each layer ℓ:

$$H^{(\ell)} = \text{EncoderBlock}^{(\ell)}\big(H^{(\ell-1)}\big), \quad \ell = 1, \dots, L.$$

EncoderBlock consists of multi-head self-attention (unmasked), position-wise FFN, and residual connections with LayerNorm.

36.3.1 Multi-Head Self-Attention (Unmasked)

For layer ℓ input $H^{(\ell-1)} \in \mathbb{R}^{T \times d_{\text{model}}}$, for head $h = 1, \dots, H$:

$$Q^{(h)} = H^{(\ell-1)} W_Q^{(h)}, \quad K^{(h)} = H^{(\ell-1)} W_K^{(h)}, \quad V^{(h)} = H^{(\ell-1)} W_V^{(h)}.$$

Score matrix:

$$S_{ij}^{(h)} = \frac{(q_i^{(h)})^\top k_j^{(h)}}{\sqrt{d_k}},$$

Since this is BERT/RoBERTa, no causal mask is used, and scores for all i, j are passed directly to softmax:

$$A_{ij}^{(h)} = \frac{\exp(S_{ij}^{(h)})}{\sum_{k=1}^{T} \exp(S_{ik}^{(h)})}.$$

36.3.2 Pre-LN Encoder Block

In practice, Pre-LN is often used:

$$\hat{H}^{(\ell)} = H^{(\ell-1)} + \text{MHA}\big(\text{LN}(H^{(\ell-1)})\big),$$

$$H^{(\ell)} = \hat{H}^{(\ell)} + \text{FFN}\Big(\text{LN}(\hat{H}^{(\ell)})\Big).$$

36.4 Masked Language Modeling with Dynamic Masking

36.4.1 Masking Strategy as a Random Process

For input sequence $x_{1:T}$, mask position set $M \subset \{1, \dots, T\}$ is selected probabilistically. For each position t, Bernoulli variable $m_t \sim \text{Bernoulli}(p_{\text{mask}})$, $p_{\text{mask}} \approx 0.15$, mask set $M = \{t \mid m_t = 1\}$.

For each $t \in M$:

$$\tilde{x}_t = \begin{cases} \texttt{[MASK]} & \text{with prob. } 0.8, \\ u \sim \text{Unif}(\mathcal{V}) & \text{with prob. } 0.1, \\ x_t & \text{with prob. } 0.1. \end{cases}$$

RoBERTa adopts "dynamic masking," where the same sentence is masked with different M in different iterations.

36.4.2 MLM Objective as Conditional Likelihood

Let the masked input sequence be $\tilde{x}_{1:T}$ and encoder output be $H^{(L)}(\tilde{x}_{1:T})$. From hidden state $h_t^{(L)}$ at position t:

$$z_t = W_{\text{MLM}} h_t^{(L)} + b_{\text{MLM}} \in \mathbb{R}^{|\mathcal{V}|},$$
$$p_\theta(v \mid \tilde{x}_{1:T}, t) = \frac{\exp(z_{t,v})}{\sum_{u \in \mathcal{V}} \exp(z_{t,u})}.$$

MLM loss per sentence:

$$\mathcal{L}_{\text{MLM}}(x_{1:T}; \theta) = \mathbb{E}_{M, \tilde{x}_{1:T}} \left[-\frac{1}{|M|} \sum_{t \in M} \log p_\theta(x_t \mid \tilde{x}_{1:T}, t) \right].$$

36.4.3 Per-Token Cross-Entropy and Gradients

Loss for position $t \in M$:

$$\ell_t(\theta) = -\log p_\theta(x_t^\star \mid \tilde{x}_{1:T}, t).$$

By standard softmax + cross-entropy result:

$$\nabla_{z_t} \ell_t = p_t - y_t.$$

Gradient w.r.t. projection parameters:

$$\nabla_{W_{\text{MLM}}} \hat{\mathcal{L}}_{\text{MLM}} = \frac{1}{|M|} \sum_{t \in M} (p_t - y_t) \, (h_t^{(L)})^\top.$$

36.5 Dynamic vs. Static Masking: Distributional View

BERT's "static masking" samples M only once during corpus preprocessing, then trains for multiple epochs with the same mask pattern.

RoBERTa's dynamic masking samples new $M^{(e)}$ each epoch:

$$\mathcal{L}_{\text{MLM}}(x_{1:T}; \theta) = \mathbb{E}_M \big[L(x_{1:T}, M; \theta) \big],$$

providing Monte Carlo iterations that more faithfully approximate the expectation.

36.6 Pretraining Objective and Optimization

For large-scale corpus $\mathcal{D}$, RoBERTa's pretraining objective is:

$$\min_\theta \; \mathbb{E}_{x_{1:T} \sim \mathcal{D}} \Big[\mathcal{L}_{\text{MLM}}(x_{1:T}; \theta) \Big].$$

RoBERTa uses:

- Larger batch size B,
- Longer training steps,
- Larger dataset

to improve representation capability while maintaining the same architecture as BERT.

Chapter 37

Longformer: Efficient Long-Document Transformer

37.1 Motivation and the $O(T^2)$ Bottleneck

Standard Transformer self-attention scales as $O(T^2)$ in time and memory for sequence length T. For long documents ($T > 4096$ etc.), this quadratic complexity becomes a practical bottleneck. Longformer introduces **sparse attention** patterns to reduce computation to linear while preserving long-range dependencies.

37.2 Attention Mask and Sparsity Pattern

37.2.1 Full Attention Recap

In standard full attention, position i attends to all positions $j \in \{1, \ldots, T\}$:

$$A_{ij} = \frac{\exp(S_{ij})}{\sum_{k=1}^{T} \exp(S_{ik})}, \quad y_i = \sum_{j=1}^{T} A_{ij} v_j.$$

37.2.2 Sparse Attention as a Structured Mask

Longformer defines a "permitted attention set" $\mathcal{N}(i) \subseteq \{1, \ldots, T\}$ for each position i:

$$M_{ij} = \begin{cases} 0 & j \in \mathcal{N}(i), \\ -\infty & j \notin \mathcal{N}(i). \end{cases}$$

37.3 Sliding Window Attention

37.3.1 Definition of Window Size w

Sliding window attention where each position i attends only to local window of radius w:

$$\mathcal{N}_{\text{local}}(i) = \big\{ j \in \{1, \ldots, T\} \;\big|\; |i - j| \le w \big\}.$$

Total attention entries computed across all positions is $O(Tw)$.

37.3.2 Multi-Layer Reception Field

With L stacked layers, each position can indirectly reach distance:

$$r_{\text{receptive}} = L \cdot w.$$

37.4 Dilated Sliding Window (Optional)

For further reception field expansion, dilated windows can be introduced. With stride d_ℓ at layer ℓ:

$$\mathcal{N}^{(\ell)}_{\text{dilated}}(i) = \big\{ j \;\big|\; j \in \{i - wd_\ell, i - (w-1)d_\ell, \ldots, i + wd_\ell\} \big\}.$$

37.5 Global Attention

37.5.1 Global Token Set $\mathcal{G} \subset \{1, \ldots, T\}$

Pre-specified global token set $\mathcal{G}$:

- Position $i \in \mathcal{G}$ can attend to all tokens $j \in \{1, \ldots, T\}$,
- Position $i \notin \mathcal{G}$ attends to tokens within window and all global tokens.

Thus:

$$\mathcal{N}_{\text{full}}(i) = \begin{cases} \{1, \ldots, T\} & i \in \mathcal{G}, \\ \mathcal{N}_{\text{local}}(i) \cup \mathcal{G} & i \notin \mathcal{G}. \end{cases}$$

37.5.2 Global Token Selection Strategies

$\mathcal{G}$ is determined based on task:

1. CLS token: First token $\mathcal{G} = \{1\}$.
2. Question tokens (QA task): All question tokens as $\mathcal{G}$.
3. Fixed interval: $\mathcal{G} = \{1, 1 + s, 1 + 2s, \ldots\}$.

37.6 Computational Complexity Analysis

37.6.1 Per-Layer Complexity

With number of global tokens $|\mathcal{G}| = g$:

$$\text{Attention cost per layer} = O\big(T(w + g)d_k\big).$$

In practice, $g \ll T$, so dominant term is $O(Twd_k)$, achieving linear scaling.

37.6.2 Total Model Complexity

Including all L layers and FFN:

$$\text{Total cost} = O\Big(L\big(Twd_k + gTd_k + Td_{\text{model}}d_{\text{ff}}\big)\Big).$$

37.7 Formal Attention Definition in Longformer

37.7.1 Score and Mask Computation

Longformer mask:

$$M_{ij}^{\text{Longformer}} = \begin{cases} 0 & j \in \mathcal{N}_{\text{full}}(i), \\ -\infty & j \notin \mathcal{N}_{\text{full}}(i). \end{cases}$$

37.7.2 Sparse Softmax

Softmax for row i is computed over non-zero scores only:

$$A_{ij} = \begin{cases} \dfrac{\exp(\tilde{S}_{ij})}{\sum_{k \in \mathcal{N}_{\text{full}}(i)} \exp(\tilde{S}_{ik})} & j \in \mathcal{N}_{\text{full}}(i), \\ 0 & \text{otherwise.} \end{cases}$$

Output:

$$y_i = \sum_{j \in \mathcal{N}_{\text{full}}(i)} A_{ij} v_j.$$

37.8 Gradient Flow Through Sparse Attention

Gradient computation during backpropagation is also performed only for permitted attention entries. The softmax Jacobian is:

$$\frac{\partial A_{ij}}{\partial S_{ik}} = \begin{cases} A_{ij}(\delta_{jk} - A_{ik}) & k \in \mathcal{N}_{\text{full}}(i), \\ 0 & k \notin \mathcal{N}_{\text{full}}(i). \end{cases}$$

37.9 Comparison with Other Efficient Transformers

37.9.1 BigBird

BigBird also uses sparse attention, adding random attention and block-sparse attention.

37.9.2 Linformer / Performer

Linformer achieves $O(T)$ via low-rank approximation, approximating the attention matrix itself. Performer achieves $O(T)$ via kernel trick linearization of softmax, but is not strictly equivalent to standard softmax attention.

Longformer's distinguishing feature is preserving exact attention on sparse connections.

37.10 Summary

Longformer:

- Captures local dependencies in $O(Tw)$ via sliding window attention,

- Handles long-range dependencies and sequence-wide information aggregation via global attention,
- With overall $O(T(w+g)d_k)$ complexity, handles long documents with $T \sim 4096$ or more,

significantly improving Transformer practicality for document-level NLP tasks.

Chapter 38

T5: Text-to-Text Transfer Transformer

38.1 Architecture Overview

Figure 38.1 illustrates the T5 architecture, which uses the full encoder-decoder Transformer structure.

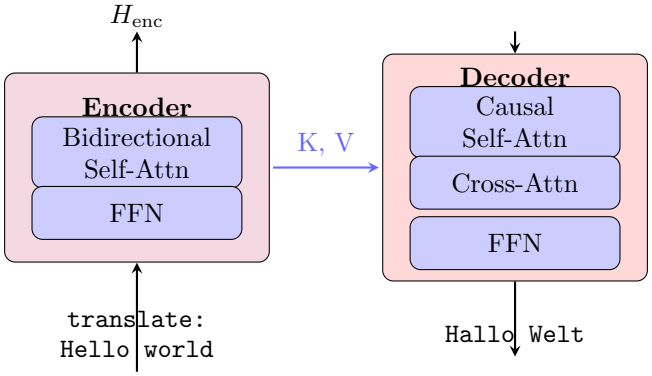

Figure 38.1: T5 Architecture: Full encoder-decoder Transformer. The encoder processes input bidirectionally; the decoder generates output autoregressively while attending to encoder representations via cross-attention.

38.1.1 Intuitive Understanding

Text-to-Text Unification: T5's revolutionary insight is that *every* NLP task can be framed as text generation:

- **Translation**: "translate English to German: Hello" → "Hallo"
- **Summarization**: "summarize: [long article]" → "[summary]"
- **Classification**: "sentiment: I love this!" → "positive"
- **Question Answering**: "question: What is 2+2? context: ..." → "4"

This unified interface allows a single model to handle diverse tasks.

Encoder-Decoder Structure: Unlike decoder-only GPT, T5 separates "understanding" (encoder) from "generation" (decoder):

- **Encoder**: Reads entire input bidirectionally, building rich representations
- **Decoder**: Generates output autoregressively, attending to encoder via cross-attention

Span Corruption Pretraining: Instead of masking single tokens (BERT) or predicting next tokens (GPT), T5 masks contiguous spans and predicts them. This teaches the model both local and global patterns.

Why This Works: By casting all tasks as sequence-to-sequence, T5 leverages the same architecture and training objective universally, enabling strong transfer learning.

38.2 Unified Text-to-Text Framework

T5 is a unified framework that treats all NLP tasks as the same input-to-output text transformation problem.

38.2.1 Task Formulation as Conditional Generation

For vocabulary $\mathcal{V}$, input sequence $x_{1:T_x} = (x_1, \ldots, x_{T_x})$, output sequence $y_{1:T_y} = (y_1, \ldots, y_{T_y})$, with $x_t, y_s \in \mathcal{V}$.

All tasks are unified as a conditional probability model:

$$p_\theta(y_{1:T_y} \mid x_{1:T_x}) = \prod_{s=1}^{T_y} p_\theta(y_s \mid x_{1:T_x}, y_{<s})$$

to treat uniformly.

38.2.2 Task Prefix and Examples

Embed task information as prefix in text:

- Translation: $x =$ "translate English to German: That is good.", $y =$ "Das ist gut."
- Summarization: $x =$ "summarize: [long document]", $y =$ "[summary]"
- Classification: $x =$ "sentiment: This movie is great!", $y =$ "positive"

38.3 Encoder-Decoder Architecture

T5 adopts standard Transformer Encoder-Decoder.

38.3.1 Encoder: Bidirectional Self-Attention

For input $x_{1:T_x}$, encoder layers $\ell = 1, \ldots, L_{\text{enc}}$ apply bidirectional self-attention where all positions can attend to each other:

$$H_{\text{enc}}^{(\ell)} = \text{EncoderBlock}^{(\ell)}\left(H_{\text{enc}}^{(\ell-1)}\right).$$

Final output:

$$H_{\text{enc}} = H_{\text{enc}}^{(L_{\text{enc}})} \in \mathbb{R}^{T_x \times d_{\text{model}}}.$$

38.3.2 Decoder: Causal Self-Attention + Cross-Attention

Decoder layers $\ell = 1, \ldots, L_{\text{dec}}$ have 3 sub-layers:

1. Masked (causal) self-attention: Apply causal mask to prevent seeing future output tokens.
2. Cross-attention: Read information from encoder output H_{enc}.
3. Feed-forward network: Position-wise nonlinear transformation.

38.3.3 Masked Self-Attention in Decoder

Causal mask:

$$M_{\text{causal},ij} = \begin{cases} 0 & j \le i, \\ -\infty & j > i, \end{cases}$$

applied as:

$$\tilde{S}_{\text{self},ij} = S_{\text{self},ij} + M_{\text{causal},ij}.$$

38.3.4 Cross-Attention to Encoder

Cross-attention to encoder output H_{enc}:

$$Q_{\text{cross}} = \hat{H}_{\text{dec}}^{(\ell)} W_Q^{\text{cross}}, \quad K_{\text{cross}} = H_{\text{enc}} W_K^{\text{cross}}, \quad V_{\text{cross}} = H_{\text{enc}} W_V^{\text{cross}}.$$

No mask here (each decoder position can see entire input):

$$A_{\text{cross},ij} = \frac{\exp(S_{\text{cross},ij})}{\sum_{k=1}^{T_x} \exp(S_{\text{cross},ik})}.$$

38.3.5 Decoder Block Composition

In Pre-LN form:

$$\begin{aligned} \hat{H}_{\text{dec}}^{(\ell)} &= H_{\text{dec}}^{(\ell-1)} + \text{MaskedSelfAttn}\big(\text{LN}(H_{\text{dec}}^{(\ell-1)})\big), \\ \tilde{H}_{\text{dec}}^{(\ell)} &= \hat{H}_{\text{dec}}^{(\ell)} + \text{CrossAttn}\big(\text{LN}(\hat{H}_{\text{dec}}^{(\ell)}), H_{\text{enc}}\big), \\ H_{\text{dec}}^{(\ell)} &= \tilde{H}_{\text{dec}}^{(\ell)} + \text{FFN}\big(\text{LN}(\tilde{H}_{\text{dec}}^{(\ell)})\big). \end{aligned}$$

38.4 Relative Position Bias

T5 uses relative position bias added to attention scores instead of absolute position embedding.

38.4.1 Bias Definition

Introduce relative position bias matrix $B^{(\ell)} \in \mathbb{R}^{T \times T}$:

$$S_{ij}^{(\text{with bias})} = \frac{q_i^\top k_j}{\sqrt{d_k}} + B_{ij}^{(\ell)}.$$

38.4.2 Bucketed Relative Position

For relative distance $d = j - i$, prepare learnable bias table $\beta^{(\ell)} \in \mathbb{R}^{N_{\text{bucket}}}$:

$$B_{ij}^{(\ell)} = \beta_{\text{bucket}(j-i)}^{(\ell)}.$$

Bucket function discretizes finely for short distances, coarsely for long distances.

38.5 Pretraining Objective: Span Corruption

T5 pretraining uses span corruption task, extending BERT's MLM.

38.5.1 Span Masking as a Random Process

For input sentence $x_{1:T}$, randomly select spans with average length λ_{span}, replace with special tokens $[\texttt{MASK}_k]$.

38.5.2 Target Sequence Construction

Target $y_{1:T_y}$ concatenates masked spans separated by sentinel tokens:
Example:

$x =$ "Thank you for inviting me to your party last week."
$\tilde{x} =$ "Thank you [X] me to your [Y] week."
$y =$ "[X] for inviting [Y] party last [Z]."

38.5.3 Denoising Objective as Conditional Likelihood

Pretraining loss:

$$\mathcal{L}_{\text{denoise}}(\theta) = \mathbb{E}_{x_{1:T},\text{spans}} \left[-\sum_{s=1}^{T_y} \log p_\theta(y_s \mid \tilde{x}_{1:T'}, y_{<s}) \right].$$

38.5.4 Per-Token Loss and Gradient

Logits at position s:

$$z_s = W_{\text{LM}} h_s^{(L_{\text{dec}})} + b_{\text{LM}} \in \mathbb{R}^{|\mathcal{V}|},$$

Gradient:

$$\nabla_{z_s} \ell_s = p_s - \text{one\_hot}(y_s^\star).$$

38.6 Teacher Forcing and Autoregressive Decoding

38.6.1 Training with Teacher Forcing

During training, ground truth output $y_{1:T_y}$ is directly fed to decoder (teacher forcing). This enables parallel computation for efficient training.

38.6.2 Inference with Autoregressive Generation

During inference, decoder is run recursively until $y_s = [\texttt{EOS}]$.

38.7 Simplified Layer Normalization (RMSNorm)

T5 uses RMSNorm:

$$\text{RMSNorm}(x) = \gamma \odot \frac{x}{\text{RMS}(x) + \varepsilon}, \quad \text{RMS}(x) = \sqrt{\frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} x_i^2}.$$

38.8 Computational Complexity

38.8.1 Encoder Complexity

For input length T_x, encoder complexity is:

$$O\big(L_{\text{enc}} \cdot T_x^2 d_k + L_{\text{enc}} \cdot T_x d_{\text{model}} d_{\text{ff}}\big).$$

38.8.2 Decoder Complexity

For output length T_y: self-attention $O(T_y^2 d_k)$, cross-attention $O(T_y T_x d_k)$, FFN $O(T_y d_{\text{model}} d_{\text{ff}})$.

38.9 Training Strategies and Hyperparameters

38.9.1 Pre-Training Corpus: C4

T5 is pretrained on Colossal Clean Crawled Corpus (C4) (approximately 750GB of clean web text).

38.9.2 Model Sizes

Model	d_{model}	d_{ff}	Parameters
T5-Small	512	2048	60M
T5-Base	768	3072	220M
T5-Large	1024	4096	770M
T5-3B	1024	16384	3B
T5-11B	1024	65536	11B

38.10 Comparison with BERT and GPT

Model	Architecture	Pretraining Objective	Target Tasks
BERT	Encoder-only	MLM + NSP	Classification/Extraction
GPT	Decoder-only	Causal LM	Generation
T5	Encoder-Decoder	Span corruption	All tasks unified

38.11 Summary

T5:

- Unifies all NLP tasks as conditional generation $p_\theta(y_{1:T_y} \mid x_{1:T_x})$,
- Introduces relative position bias to Encoder-Decoder Transformer,
- Performs denoising pretraining via span corruption,
- Achieves general-purpose transfer learning through text-to-text framework,

demonstrating the possibility of unified modeling across diverse NLP tasks.

Chapter 39

Vision Transformer (ViT): Transformers for Image Classification

39.1 Architecture Overview

Figure 39.1 illustrates the Vision Transformer architecture, which applies a standard Transformer encoder directly to sequences of image patches.

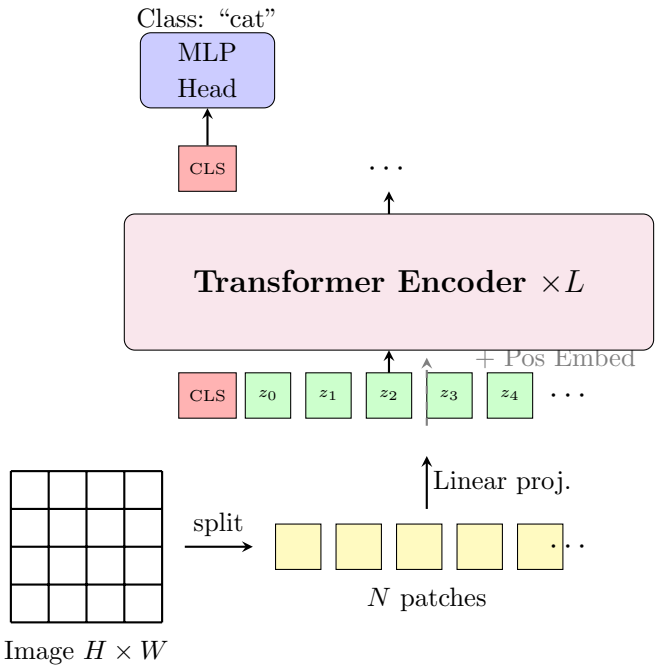

Figure 39.1: Vision Transformer (ViT) Architecture: An image is split into fixed-size patches, linearly embedded, and processed by a standard Transformer encoder. The [CLS] token’s output is used for classification.

39.1.1 Intuitive Understanding

Images as Sequences: ViT's key insight is treating an image not as a 2D grid, but as a sequence of patches—just like a sentence is a sequence of words. A 224×224 image with 16×16 patches becomes a sequence of 196 "visual tokens."

Why Patches? Self-attention has $O(n^2)$ complexity. Using individual pixels (224×224 = 50,176 tokens) would be prohibitive. Patches (14×14 = 196 tokens) make it tractable while preserving local structure within each patch.

No Convolutions: ViT demonstrates that the inductive biases of CNNs (locality, translation invariance) are not strictly necessary. Given enough data, self-attention can learn these patterns from scratch, and potentially discover more flexible representations.

The [CLS] Token: Borrowed from BERT, a learnable "class" token is prepended to the patch sequence. After passing through all layers, this token's representation contains a global summary of the image, which is fed to a classification head.

Data Hunger: Without CNN's inductive biases, ViT requires massive datasets (ImageNet-21k, JFT-300M) for pretraining. On smaller datasets, CNNs still outperform ViT. This highlights the trade-off between flexibility and data efficiency.

Position Embeddings: Unlike CNNs that inherently understand spatial relationships, ViT must learn positions via learnable embeddings. Interestingly, these learned embeddings exhibit 2D spatial structure matching patch positions.

39.2 Motivation: From Convolution to Self-Attention

Traditional image classification was dominated by convolutional neural networks (CNN) with local feature extraction and hierarchical representation learning. Vision Transformer (ViT) treats images as sequence data, directly applying standard Transformer Encoder, removing CNN's inductive bias (locality, translation invariance), enabling learning of global dependencies via pure self-attention mechanism.

39.3 Image as a Sequence: Patch Embedding

39.3.1 Image Partitioning into Patches

Let input image be $I \in \mathbb{R}^{H\times W\times C}$. Partition image into $P \times P$ pixel square patches. Number of patches:

$$N = \frac{H \cdot W}{P^2}.$$

Typically $H = W = 224$, $P = 16$ gives $N = 196$.

39.3.2 Patch Extraction as Tensor Reshaping

Partition image I into N patch vectors $\{\mathbf{x}_p^{(i)}\}_{i=1}^N$, $\mathbf{x}_p^{(i)} \in \mathbb{R}^{P^2C}$.

39.3.3 Linear Projection to Embedding Dimension

Linearly project each patch vector to model dimension d_{model}:

$$\mathbf{z}_i^{(0)} = E\mathbf{x}_p^{(i)} + \mathbf{b}, \quad E \in \mathbb{R}^{d_{\text{model}}\times(P^2C)}.$$

39.4 Prepending the Class Token

For classification tasks, add learnable class token $\mathbf{z}_{\text{cls}} \in \mathbb{R}^{d_{\text{model}}}$ at the beginning of sequence:

$$Z_{\text{full}}^{(0)} = \begin{bmatrix} \mathbf{z}_{\text{cls}}^\top \\ (\mathbf{z}_1^{(0)})^\top \\ \vdots \\ (\mathbf{z}_N^{(0)})^\top \end{bmatrix} \in \mathbb{R}^{(N+1)\times d_{\text{model}}}.$$

39.5 Positional Encoding

39.5.1 Learnable 1D Position Embedding

Add learnable position embedding for each position $i \in \{0, 1, \ldots, N\}$:

$$\mathbf{z}_i^{(0)\prime} = \mathbf{z}_i^{(0)} + \mathbf{e}_{\text{pos}}^{(i)}.$$

Position embedding matrix $E_{\text{pos}} \in \mathbb{R}^{(N+1)\times d_{\text{model}}}$ is learned from training.

39.5.2 Input Embedding Composition

Final input embedding:

$$H^{(0)} = Z_{\text{full}}^{(0)} + E_{\text{pos}} \in \mathbb{R}^{(N+1)\times d_{\text{model}}}.$$

39.6 Transformer Encoder

ViT stacks L layers of standard Transformer Encoder.

39.6.1 Multi-Head Self-Attention

For layer ℓ input $H^{(\ell-1)} \in \mathbb{R}^{(N+1)\times d_{\text{model}}}$, ViT uses bidirectional attention without mask:

$$A_{ij}^{(h)} = \frac{\exp(S_{ij}^{(h)})}{\sum_{k=0}^{N} \exp(S_{ik}^{(h)})}.$$

39.6.2 Layer Normalization and Residual Connections

In Pre-LN form:

$$\hat{H}^{(\ell)} = H^{(\ell-1)} + \text{MHA}\big(\text{LN}(H^{(\ell-1)})\big),$$
$$H^{(\ell)} = \hat{H}^{(\ell)} + \text{FFN}\Big(\text{LN}(\hat{H}^{(\ell)})\Big).$$

39.6.3 Position-Wise Feed-Forward Network

ViT uses GELU as activation function:

$$\text{FFN}(u) = W_2\,\text{GELU}(W_1 u + b_1) + b_2,$$
$$\text{GELU}(x) = x\Phi(x) = x \cdot \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right].$$

39.7 Classification Head

39.7.1 Extracting the CLS Token

Extract CLS token representation $\mathbf{h}_{\text{cls}}^{(L)} \in \mathbb{R}^{d_{\text{model}}}$ from final layer L output.

39.7.2 Linear Classification Layer

For K-class classification:

$$\mathbf{z}_{\text{cls}} = W_{\text{head}}\mathbf{h}_{\text{cls}}^{(L)} + \mathbf{b}_{\text{head}} \in \mathbb{R}^K,$$

Prediction distribution:

$$p_\theta(y = k \mid I) = \frac{\exp(z_{\text{cls},k})}{\sum_{j=1}^{K} \exp(z_{\text{cls},j})}.$$

39.7.3 Cross-Entropy Loss

For 1-hot label $\mathbf{y} \in \{0, 1\}^K$:

$$\mathcal{L}_{\text{CE}}(\theta) = -\sum_{k=1}^{K} y_k \log p_\theta(y = k \mid I).$$

Gradient:

$$\nabla_{\mathbf{z}_{\text{cls}}} \mathcal{L}_{\text{CE}} = p_\theta - \mathbf{y}, \quad \nabla_{W_{\text{head}}} \mathcal{L}_{\text{CE}} = (p_\theta - \mathbf{y})(\mathbf{h}_{\text{cls}}^{(L)})^\top.$$

39.8 Pre-Training and Transfer Learning

39.8.1 Pre-Training on Large Datasets

ViT achieves high performance through pretraining on large-scale datasets:

- ImageNet-21k: approximately 14 million images, 21,000 classes,
- JFT-300M: approximately 300 million images.

39.8.2 Fine-Tuning on Target Datasets

For downstream tasks, reinitialize classification head and update all parameters. When fine-tuning on higher resolution images, 2D interpolate position embeddings.

39.9 Model Variants and Scaling

Model	Layers L	Hidden dim d_{model}	Heads H	Parameters
ViT-Base	12	768	12	86M
ViT-Large	24	1024	16	307M
ViT-Huge	32	1280	16	632M

39.10 Computational Complexity

39.10.1 Self-Attention Complexity

Self-attention for sequence length $N + 1$:

$$O\big((N+1)^2 d_k\big) = O\left(\left(\frac{HW}{P^2}\right)^2 d_k\right).$$

Larger P means smaller N, reducing computation.

39.10.2 Comparison with CNN

CNN is linear in image size due to local receptive fields. ViT is $O((HW/P^2)^2)$, controllable by patch size P.

39.11 Inductive Bias and Data Efficiency

CNN has inductive bias of locality and translation invariance, while ViT directly learns global dependencies across all patches via self-attention, lacking these biases.

Due to weak inductive bias, ViT underperforms CNN without large-scale dataset pretraining.

39.12 Extensions and Variants

39.12.1 DeiT (Data-efficient image Transformer)

Uses distillation to improve training efficiency on smaller datasets.

39.12.2 Swin Transformer

Hierarchical structure and shifted window self-attention for efficient computation with locality.

39.12.3 BEiT

BERT-style self-supervised learning by masking and predicting image patches.

39.13 Summary

Vision Transformer (ViT):

- Partitions image into $P \times P$ patches and embeds via linear projection,
- Adds CLS token and learnable position embeddings,
- Applies standard Transformer Encoder, learning global dependencies via bidirectional self-attention,

- Predicts via linear classification head from CLS token representation,
- Achieves performance exceeding CNN through large-scale pretraining,

establishing the paradigm of directly applying Transformers to vision tasks.

Chapter 40

PaLM: Pathways Language Model

40.1 Overview and Scaling Philosophy

PaLM (Pathways Language Model) is an ultra-large-scale decoder-only language model developed by Google, with the largest configuration having **540B parameters**. While following the autoregressive language model design of the GPT family, it improves efficiency and performance through the following innovations:

- **Pathways system**: Ultra-parallel distributed training across thousands of TPUs,
- **SwiGLU activation**: Improved FFN expressiveness,
- **Parallel layers**: Parallel execution of Attention and FFN,
- **Multi-query attention**: Efficient KV cache,
- **RoPE**: Rotary Position Embedding for relative positions.

40.2 Autoregressive Language Modeling

PaLM learns the autoregressive distribution of token sequences $x_{1:T}$:

$$p_\theta(x_{1:T}) = \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t}),$$

Training objective is negative log-likelihood minimization:

$$\mathcal{L}_{\text{LM}}(\theta) = \mathbb{E}_{x_{1:T}\sim\mathcal{D}}\left[-\frac{1}{T}\sum_{t=1}^{T}\log p_\theta(x_t \mid x_{1:t-1})\right].$$

40.3 Parallel Layers Architecture

In standard Transformer, each layer is sequential: Attention $\to$ Add&Norm $\to$ FFN $\to$ Add&Norm. PaLM parallelizes Attention and FFN:

$$H^{(\ell)} = H^{(\ell-1)} + \text{Attention}^{(\ell)}\big(\text{LN}(H^{(\ell-1)})\big) + \text{FFN}^{(\ell)}\big(\text{LN}(H^{(\ell-1)})\big).$$

40.4 Multi-Query Attention

Multi-query attention shares K, V across all heads:

$$Q^{(h)} = HW_Q^{(h)} \in \mathbb{R}^{T \times d_k}, \quad h = 1, \ldots, H,$$

$$K = HW_K \in \mathbb{R}^{T \times d_k}, \quad V = HW_V \in \mathbb{R}^{T \times d_v},$$

where W_K, W_V are single matrices shared across heads. KV cache size is reduced from $O(H \cdot T \cdot d_k)$ to $O(T \cdot d_k)$.

40.5 RoPE: Rotary Position Embedding

RoPE embeds relative position information into attention scores via rotation matrices. For position t, apply rotation:

$$\tilde{q}_t = R_t q_t, \quad \tilde{k}_t = R_t k_t,$$

where R_t is a block-diagonal rotation matrix. The score becomes:

$$S_{ij} = \frac{q_i^\top R_{j-i} k_j}{\sqrt{d_k}},$$

depending only on relative position $j - i$.

40.6 SwiGLU Activation

PaLM adopts SwiGLU:

$$\text{SwiGLU}(x) = (W_1 x) \odot \text{Swish}(W_2 x),$$

$$\text{Swish}(z) = z \cdot \sigma(z) = \frac{z}{1 + e^{-z}}.$$

40.7 Model Configurations

Model	Layers	d_{model}	Heads	d_{ff}	Params
PaLM-8B	32	4096	32	16384	8B
PaLM-62B	64	8192	64	32768	62B
PaLM-540B	118	18432	48	49152	540B

40.8 Summary

PaLM achieves scaling and efficiency through RoPE, SwiGLU, parallel layers, multi-query attention, and Pathways distributed training across thousands of TPUs.

Chapter 41

LLaMA: Large Language Model Meta AI

41.1 Overview and Design Philosophy

LLaMA (Large Language Model Meta AI) is an open-source large-scale decoder-only language model developed by Meta. Key design features include:

- **RMSNorm**: Simplified LayerNorm for efficiency,
- **SwiGLU activation**: Improved FFN expressiveness,
- **RoPE**: Efficient relative position encoding,
- **Pre-normalization**: Training stability,
- **Public data only**: Transparency and reproducibility.

LLaMA provides 4 sizes: 7B, 13B, 33B, and 65B.

41.2 RMSNorm: Root Mean Square Layer Normalization

LLaMA adopts RMSNorm:

$$\text{RMSNorm}(x) = \gamma \odot \frac{x}{\text{RMS}(x)},$$

$$\text{RMS}(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \varepsilon},$$

where $\gamma \in \mathbb{R}^d$ is a learnable scale parameter. Unlike LayerNorm, RMSNorm omits mean shift and bias term.

41.3 Pre-Normalization Architecture

LLaMA uses Pre-LN structure:

$$\tilde{H}^{(\ell)} = H^{(\ell-1)} + \text{Attention}^{(\ell)}\big(\text{RMSNorm}(H^{(\ell-1)})\big)\,,$$

$$H^{(\ell)} = \tilde{H}^{(\ell)} + \text{FFN}^{(\ell)}\Big(\text{RMSNorm}(\tilde{H}^{(\ell)})\Big)\,.$$

41.4 Causal Self-Attention with RoPE

For query and key at position t, apply rotation matrix R_t:

$$\tilde{Q}_t^{(h)} = R_t Q_t^{(h)}, \quad \tilde{K}_t^{(h)} = R_t K_t^{(h)}.$$

Score depends only on relative position:

$$S_{ij}^{(h)} = \frac{(Q_i^{(h)})^\top R_{j-i} K_j^{(h)}}{\sqrt{d_k}}.$$

41.5 SwiGLU Feed-Forward Network

$$\text{SwiGLU}(x) = (W_1 x) \odot \text{Swish}(W_2 x),$$
$$\text{FFN}(x) = W_3\,\text{SwiGLU}(x).$$

No bias terms are used (LLaMA omits biases in all layers).

41.6 Model Configurations

Model	Layers	d_{model}	Heads	d_{ff}	Params
LLaMA-7B	32	4096	32	11008	6.7B
LLaMA-13B	40	5120	40	13824	13.0B
LLaMA-33B	60	6656	52	17920	32.5B
LLaMA-65B	80	8192	64	22016	65.2B

41.7 Training Data

LLaMA is trained on public datasets only (approximately 1.4T tokens): CommonCrawl (67%), C4 (15%), GitHub (4.5%), Wikipedia (4.5%), Books (4.5%), ArXiv (2.5%), StackExchange (2%).

41.8 LLaMA 2 Improvements

LLaMA 2 introduces:

- Grouped-Query Attention (GQA): Balance between multi-head and multi-query,
- Longer context window: $T_{\max} = 4096$,
- RLHF for safety alignment.

41.9 Summary

LLaMA achieves efficient and transparent large-scale language modeling through RMSNorm, SwiGLU, RoPE, pre-normalization, and public data training.

Chapter 42

Mixtral: Sparse Mixture of Experts Language Model

42.1 Architecture Overview

Figure 42.1 illustrates the Sparse Mixture of Experts (MoE) architecture used in Mixtral.

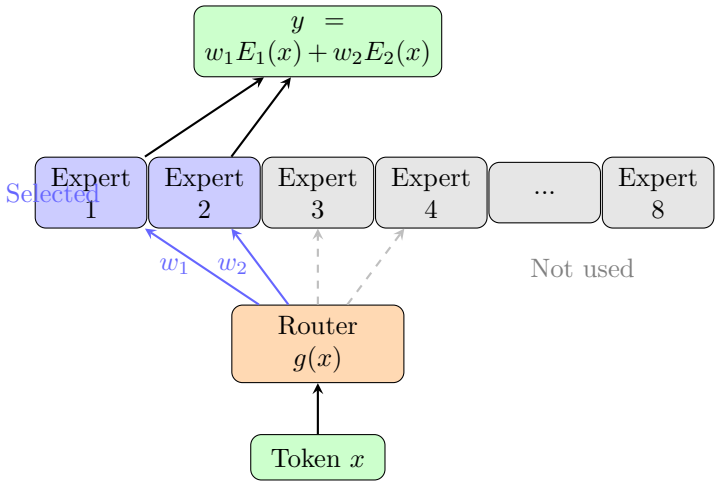

Figure 42.1: Mixtral Sparse MoE Architecture: A gating network routes each token to the top-2 experts (out of 8). Only selected experts are computed, achieving 47B total parameters with 13B active computation.

42.1.1 Intuitive Understanding

The Expert Analogy: Imagine a hospital with specialists: cardiologists, neurologists, dermatologists, etc. When a patient arrives, a triage system (router) directs them to the 1-2 most relevant specialists, not all doctors. Similarly, MoE routes each token to experts best suited for it.

Sparse = Efficient: With 8 experts but only 2 active per token, Mixtral uses $\frac{2}{8} = 25\%$ of expert computation. Total parameters (47B) provide capacity; active parameters (13B) determine speed. This decouples model capacity from inference cost.

Why It Works: Different tokens may need different computations:

- Code tokens → Expert specializing in programming patterns

- Math tokens → Expert specializing in numerical reasoning
- Language tokens → Expert specializing in linguistic patterns

The router learns to make these assignments automatically.

Load Balancing Challenge: Without constraints, the router might always pick the same experts ("expert collapse"). The auxiliary loss encourages uniform expert utilization.

Key Insight: Mixtral shows that conditional computation (using different parameters for different inputs) is a promising scaling direction, achieving better quality per FLOP than dense models.

42.2 Sparse MoE Architecture Details

Mixtral is a large-scale language model with Sparse Mixture of Experts (SMoE) architecture developed by Mistral AI. Key features:

- **8 expert FFNs**: Multiple specialist networks per layer,
- **Top-2 routing**: Select optimal 2 experts per input,
- **Total 47B parameters**: Sum of all expert parameters,
- **Active 13B parameters**: Actually used during inference,
- **Load balancing**: Auxiliary loss for equal expert utilization.

This achieves 47B parameter expressiveness with 13B model computation cost.

42.3 Mixture of Experts Formulation

$N = 8$ expert networks, each with SwiGLU structure:

$$\text{Expert}_e(x) = W_{3,e}\,\text{SwiGLU}(x).$$

Gating network computes routing probabilities:

$$g(x) = \text{softmax}(W_g x) \in \mathbb{R}^N.$$

42.4 Top-k Routing

Top-2 routing ($k = 2$) selects the two highest-scoring experts:

$$\mathcal{T}_2(x) = \{e_1, e_2\},$$

$$y = \tilde{g}_{e_1}(x) \cdot \text{Expert}_{e_1}(x) + \tilde{g}_{e_2}(x) \cdot \text{Expert}_{e_2}(x),$$

where $\tilde{g}_e$ are renormalized weights over selected experts.

Computation is reduced to $k/N = 2/8 = 1/4$ of dense MoE.

42.5 Load Balancing Loss

To prevent expert load imbalance, auxiliary loss is introduced:

$$f_e = \frac{1}{B}\sum_{i=1}^{B} g_e(x_i), \quad P_e = \frac{1}{B}\sum_{i=1}^{B} \mathbb{1}\{e \in \mathcal{T}_k(x_i)\},$$

$$\mathcal{L}_{\text{balance}} = \alpha \cdot N \sum_{e=1}^{N} f_e \cdot P_e.$$

Total loss:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{LM}} + \mathcal{L}_{\text{balance}}.$$

42.6 Sliding Window Attention

Mixtral uses sliding window attention for long context (32k tokens): Window size $w = 4096$, position i attends to $[i - w, i]$. Reduces attention from $O(T^2 d_k)$ to $O(Twd_k)$.

42.7 Parameter Count

- **Total**: 47B parameters (8 experts per layer),
- **Active**: 13B parameters (top-2 experts).

42.8 Comparison with Dense Models

Model	Total params	Active params	Inference cost
LLaMA-2-13B	13B	13B	1.0×
LLaMA-2-70B	70B	70B	5.4×
Mixtral-8x7B	47B	13B	1.0×

Mixtral achieves 70B model performance with 13B computation cost.

42.9 Summary

Mixtral achieves breakthrough efficiency through:

- Sparse MoE architecture with 8 expert FFNs,
- Top-2 routing selecting optimal 2 experts per input,
- 47B total / 13B active parameters,
- Load balancing loss for expert utilization,
- Sliding window attention for 32k context,
- LLaMA-style RMSNorm, SwiGLU, RoPE.

Mixtral demonstrates that Sparse MoE is a promising direction for future LLM scaling.