Mathematical Foundations of Deep Learning and Embodied Intelligence

From Neural Networks to Diffusion Policies and VLA Models

January 24, 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:

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

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

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

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:

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

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:

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

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):

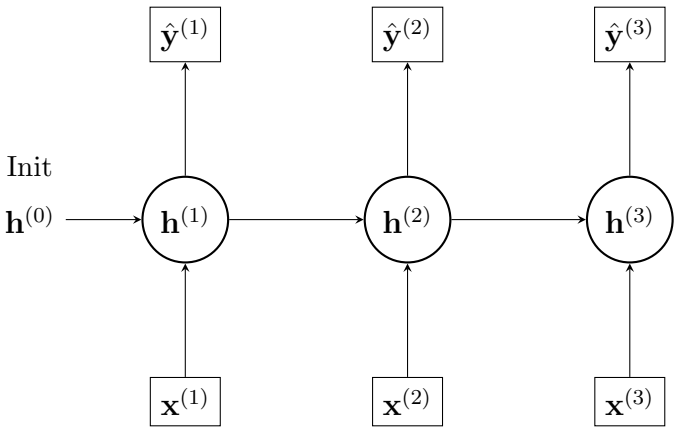

Figure 15.1: Unrolled Recurrent Neural Network. The hidden state $\mathbf{h}^{(t)}$ passes information to the next timestep. (Adapted from Goodfellow et al., 2016)

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:

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

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

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 Partially Observable Markov Decision Processes (POMDPs)

16.2.1 Formal Definition

Definition 16.1 (POMDP Tuple). A Partially Observable Markov Decision Process is a 6-tuple

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{R}, \gamma), \tag{16.1}$$

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 ($0 \leq \gamma < 1$)

16.2.2 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:

$$\mathbf{s}_t = (\mathbf{p}_{\text{obj}}, \mathbf{v}_{\text{obj}}, \mathbf{q}_{\text{robot}}, \dot{\mathbf{q}}_{\text{robot}}, \text{contact flags}, \ldots) \in \mathbb{R}^{n_s}, \tag{16.2}$$

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):

$$\mathbf{s}_t = (\underbrace{\mathbf{p}_{\text{handle}}, \mathbf{p}_{\text{drawer\_body}}, \theta_{\text{joint}}}_{\text{object config}}, \underbrace{\mathbf{q}_{\text{arm}}, \dot{\mathbf{q}}_{\text{arm}}}_{\text{arm state}}, \underbrace{\mu_{\text{friction}}, \text{handle stiffness}}_{\text{hidden params}}) \tag{16.3}$$

Observation Space $\mathcal{O}$

The agent receives multimodal observations, typically:

$$\mathbf{o}_t = (\mathbf{I}_t^{(\text{wrist})}, \mathbf{I}_t^{(\text{overhead})}, \mathbf{q}_t, \mathbf{f}_t) \in \mathcal{O}. \tag{16.4}$$

Visual Component:

$$\mathbf{I}_t^{(c)} \in \mathbb{R}^{H \times W \times 3}, \tag{16.5}$$

where $c \in \{\text{wrist}, \text{overhead}, \ldots\}$ indexes camera viewpoints, typically $H = W = 84$ or 224 for deep learning pipelines.

Proprioceptive Component:

$$\mathbf{q}_t \in \mathbb{R}^D \quad \text{(joint angles)} \tag{16.6}$$

$$\dot{\mathbf{q}}_t \in \mathbb{R}^D \quad \text{(joint velocities, optional)} \tag{16.7}$$

Force/Torque Component (if available):

$$\mathbf{f}_t \in \mathbb{R}^6, \quad \mathbf{f}_t = (\mathbf{F}_{\text{linear}}, \boldsymbol{\tau}_{\text{rotational}}) \tag{16.8}$$

Dimension Analysis:

$$\dim(\mathcal{O}) = (\text{num cameras}) \times (H \times W \times 3) + D + D + 6 \tag{16.9}$$

$$= 2 \times (84 \times 84 \times 3) + 7 + 7 + 6 = 42,258 + 20 \approx 42k \tag{16.10}$$

This high dimensionality necessitates representation learning (vision backbones) before action prediction.

Action Space $\mathcal{A}$

For continuous control (this course's focus):

$$\mathcal{A} = \mathbb{R}^{n_a}, \quad \text{e.g., } n_a \in \{3, 6, 7, 14\} \tag{16.11}$$

Interpretation (Position Control):

$$\mathbf{a}_t = (\Delta\mathbf{q}_{\text{arm}}, \delta_{\text{gripper}}) \in \mathbb{R}^D, \tag{16.12}$$

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):

$$\mathbf{a}_t = (\dot{\mathbf{q}}_{\text{desired}}, \text{gripper speed}) \in \mathbb{R}^D. \tag{16.13}$$

The key constraint: $\mathcal{A}$ is **bounded** (actuator limits), so we may write

$$\mathcal{A} = [-1, 1]^{n_a} \quad \text{(normalized)} \tag{16.14}$$

or

$$\mathcal{A} = \prod_{i=1}^{n_a} [a_i^{\min}, a_i^{\max}]. \tag{16.15}$$

Transition Model $\mathcal{T}$

The transition model specifies how the world evolves:

$$\mathcal{T}(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S}), \tag{16.16}$$

where $\Delta(\mathcal{S})$ is the space of probability distributions over $\mathcal{S}$.

In deterministic settings (idealized):

$$\mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{a}_t) \quad \text{(physics dynamics)}. \tag{16.17}$$

In stochastic settings (realistic):

$$\mathbf{s}_{t+1} \sim \mathcal{T}(\cdot|\mathbf{s}_t, \mathbf{a}_t) = \mathcal{N}(f(\mathbf{s}_t, \mathbf{a}_t), \Sigma_{\text{dynamics}}), \tag{16.18}$$

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:

$$\mathbf{o}_t \sim \mathcal{P}(\cdot|\mathbf{s}_t), \tag{16.19}$$

where $\mathcal{P}$ is the observation model. In practice, this is often deterministic (perfect sensors):

$$\mathbf{o}_t = h(\mathbf{s}_t) + \boldsymbol{\epsilon}_{\text{sensor}}, \quad \boldsymbol{\epsilon}_{\text{sensor}} \sim \mathcal{N}(\mathbf{0}, \Sigma_{\text{obs}}). \tag{16.20}$$

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:

$$\mathcal{R}(\mathbf{s}_t, \mathbf{a}_t) \in \mathbb{R}, \tag{16.21}$$

measuring task progress.

Example (Reaching Task):

$$\mathcal{R}(\mathbf{s}_t, \mathbf{a}_t) = \begin{cases} 1 & \text{if } \|\mathbf{p}_{\text{gripper}} - \mathbf{p}_{\text{target}}\|_2 < \epsilon \\ -\|\mathbf{p}_{\text{gripper}} - \mathbf{p}_{\text{target}}\|_2^2 - \lambda\|\mathbf{a}_t\|_2^2 & \text{otherwise} \end{cases}. \tag{16.22}$$

Example (Drawer Opening):

$$\mathcal{R}(\mathbf{s}_t) = \begin{cases} +10 & \text{if drawer opened} > 0.2 \text{ m} \\ -0.1 & \text{per timestep (action cost)} \end{cases}. \tag{16.23}$$

Discount Factor γ

The discount factor weighs immediate vs. future rewards:

$$\gamma \in [0, 1), \quad \text{typical: } \gamma \in \{0.95, 0.99, 0.999\}. \tag{16.24}$$

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.

16.2.3 Policy Representation

Definition 16.2 (Policy in POMDPs). A policy π is a mapping from **observation histories** to action distributions:

$$\pi : \mathcal{H} \to \Delta(\mathcal{A}), \tag{16.25}$$

where $\mathcal{H} = \mathcal{O}^* = \bigcup_{t=0}^{\infty} \mathcal{O}^t$ is the space of finite observation histories.

Formally, at timestep t:

$$\pi(\mathbf{a}_t|\mathbf{o}_{0:t}) = P(\mathbf{a}_t|\mathbf{o}_0, \mathbf{o}_1, \ldots, \mathbf{o}_t). \tag{16.26}$$

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 $\mathbf{h}_t$:

$$\mathbf{h}_{t+1} = \phi_{\text{history}}(\mathbf{h}_t, \mathbf{o}_{t+1}; \theta_{\text{enc}}), \tag{16.27}$$

$$\pi(\mathbf{a}_t|\mathbf{h}_t) = P(\mathbf{a}_t|\phi_{\text{context}}(\mathbf{h}_t; \theta_{\text{dec}})). \tag{16.28}$$

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.

16.3 Why POMDPs Are Different from Supervised Learning

16.3.1 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

16.3.2 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):

$$P(\mathbf{w}_{t+1}|\mathbf{w}_{<t}) \in \Delta(\mathcal{V}), \quad |\mathcal{V}| \approx 50k, \tag{16.29}$$

where $\mathcal{V} = \{\text{word}_1, \text{word}_2, \ldots\}$ is a finite vocabulary.

Loss: **cross-entropy** on categorical distribution

$$\mathcal{L}_{\text{NLP}} = -\sum_{v=1}^{|\mathcal{V}|} y_v \log \hat{p}_v, \quad y_v \in \{0,1\}, \quad \sum_v y_v = 1. \tag{16.30}$$

Continuous (Robotics / Control):

$$P(\mathbf{a}_t|\mathbf{o}_{0:t}) = \mathcal{N}(\boldsymbol{\mu}(\mathbf{o}_{0:t}), \Sigma(\mathbf{o}_{0:t})), \tag{16.31}$$

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

$$\mathcal{L}_{\text{robot}} = -\log \mathcal{N}(\mathbf{a}_t; \boldsymbol{\mu}, \Sigma) = \frac{1}{2}\log|2\pi\Sigma| + \frac{1}{2}(\mathbf{a}_t - \boldsymbol{\mu})^\top \Sigma^{-1}(\mathbf{a}_t - \boldsymbol{\mu}). \tag{16.32}$$

For diagonal covariance $\Sigma = \text{diag}(\sigma_1^2, \ldots, \sigma_{n_a}^2)$:

$$\mathcal{L}_{\text{robot}} = \frac{n_a}{2}\log(2\pi) + \sum_{i=1}^{n_a} \log \sigma_i + \frac{1}{2\sigma_i^2}(a_i - \mu_i)^2. \tag{16.33}$$

16.3.3 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 $\theta_1 = 30°$

$$\mathbf{a}^{(1)}_{t:t+k} = (x = -5, y = 0, z = -2, \theta = 30°, \text{gripper} = \text{close}) \tag{16.34}$$

- **Mode 2 (Right Grasp):** Approach from the right, angle $\theta_2 = -30°$

$$\mathbf{a}^{(2)}_{t:t+k} = (x = +5, y = 0, z = -2, \theta = -30°, \text{gripper} = \text{close}) \tag{16.35}$$

- **Mode 3 (Top Grasp):** Approach from above

$$\mathbf{a}^{(3)}_{t:t+k} = (x = 0, y = 0, z = -5, \text{approach vertical}, \text{gripper} = \text{close}) \tag{16.36}$$

Averaging Modes is Catastrophic: If we naively average multiple demonstrations:

$$\bar{\mathbf{a}} = \frac{1}{3}(\mathbf{a}^{(1)} + \mathbf{a}^{(2)} + \mathbf{a}^{(3)}) = (x = 0, y = 0, z = -3, \theta = 0°, \text{gripper} = \text{partially close}), \tag{16.37}$$

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:

$$P(\mathbf{a}_{t:t+k}|\mathbf{o}_t) = \sum_{m=1}^{M} p_m \mathcal{N}(\mathbf{a}_{t:t+k}; \boldsymbol{\mu}_m, \Sigma_m), \quad M > 1, \quad p_m > 0. \tag{16.38}$$

Naive Behavioral Cloning (fails):

$$\mathcal{L}_{\text{BC}} = \mathbb{E}_{(\mathbf{o},\mathbf{a})\sim\mathcal{D}} \left[\|\mathbf{a} - \mu_{\text{network}}(\mathbf{o})\|_2^2 \right]. \tag{16.39}$$

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:

$$P(\mathbf{a}_{t:t+k}|\mathbf{o}_t) = \int P(\mathbf{a}_{t:t+k}|\mathbf{z}, \mathbf{o}_t) P(\mathbf{z}|\mathbf{o}_t) d\mathbf{z}. \tag{16.40}$$

Each latent value $\mathbf{z}$ corresponds to a different mode, allowing the policy to generate coherent, physically realistic action sequences.

16.4 Observation Encoding: From High-Dimensional Images to Features

16.4.1 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.

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

Definition 16.4 (Vision Backbone). A vision backbone is a function

$$\Phi_{\text{vision}} : \mathbb{R}^{C_{\text{in}} \times H \times W} \to \mathbb{R}^{C_{\text{out}} \times H' \times W'}, \tag{16.41}$$

where $C_{\text{in}} = 3$ (RGB), and (C_{out}, H', W') are architecture-dependent.

Common Architectures:

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:

$$\mathbf{I} \in \mathbb{R}^{B \times 2 \times 3 \times 84 \times 84}, \tag{16.42}$$

after ResNet-18:

$$\mathbf{F} = \Phi_{\text{vision}}(\mathbf{I}) \in \mathbb{R}^{B \times 2 \times 512 \times 3 \times 3}. \tag{16.43}$$

16.4.2 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**:

$$\text{Tokens}_{\text{vis}} = \text{Reshape}(\mathbf{F}, (H'W', C_{\text{out}})) \in \mathbb{R}^{H'W' \times C_{\text{out}}}. \tag{16.44}$$

Example (Two Cameras):

$$\mathbf{F}_{\text{wrist}} \in \mathbb{R}^{512 \times 3 \times 3} \to \text{Tokens}_{\text{wrist}} \in \mathbb{R}^{9 \times 512} \tag{16.45}$$

$$\mathbf{F}_{\text{overhead}} \in \mathbb{R}^{512 \times 3 \times 3} \to \text{Tokens}_{\text{overhead}} \in \mathbb{R}^{9 \times 512} \tag{16.46}$$

$$\text{Tokens}_{\text{vis}} = [\text{Tokens}_{\text{wrist}}; \text{Tokens}_{\text{overhead}}] \in \mathbb{R}^{18 \times 512}. \tag{16.47}$$

Positional Embeddings (Optional but Recommended): To preserve spatial information, add learnable positional embeddings:

$$\text{Tokens}_{\text{vis}}^{(i)} \leftarrow \text{Tokens}_{\text{vis}}^{(i)} + \mathbf{e}_i, \quad \mathbf{e}_i \in \mathbb{R}^{512}, \tag{16.48}$$

where $\mathbf{e}_i$ is a learnable vector for position i.

Alternatively, use **2D positional encodings** that encode both spatial coordinates:

$$\mathbf{e}_{(h,w)} = \sin\left(\frac{h}{10000^{0/512}}\right) + \cos\left(\frac{h}{10000^{1/512}}\right) + \dots \tag{16.49}$$

16.5 Proprioceptive State Integration

16.5.1 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:

$$\mathbf{q}_t \in \mathbb{R}^D, \quad \dot{\mathbf{q}}_t \in \mathbb{R}^D, \quad \mathbf{f}_t \in \mathbb{R}^6 \quad \text{(if available)}. \tag{16.50}$$

Total proprioceptive dimension: $D_{\text{prop}} = D + D + 6 = 2D + 6$ (or D if only positions).

Encoding as Token:

$$\text{Token}_{\text{prop}} = \text{Linear}_{D_{\text{prop}} \to d_{\text{model}}}(\mathbf{q}_t, \dot{\mathbf{q}}_t, \mathbf{f}_t) \in \mathbb{R}^{d_{\text{model}}}, \tag{16.51}$$

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.

16.5.2 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.

16.6 Action Space: Control Parameterizations

16.6.1 End-Effector vs. Joint Space

Actions can be specified in different spaces:

Joint Space (Workspace):

$$\mathbf{a}_t^{(\text{joint})} = \Delta\mathbf{q}_t \in \mathbb{R}^D, \quad \text{e.g., } \Delta q_i \in [-0.1, 0.1] \text{ rad}. \tag{16.52}$$

Advantages: Direct control, interpretable, avoids IK problems. Disadvantages: Not invariant to robot kinematics changes; difficult to compare across robots.

Task Space (Operational Space):

$$\mathbf{a}_t^{(\text{task})} = (\Delta\mathbf{p}, \Delta\mathbf{R}) \in \mathbb{R}^3 \times SO(3), \tag{16.53}$$

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.

16.6.2 Gripper Commands

Gripper control is typically 1-dimensional:

$$a_{\text{gripper}} \in [-1, 1], \quad \text{where} \quad a_{\text{gripper}} < 0 \Rightarrow \text{open}, \quad a_{\text{gripper}} > 0 \Rightarrow \text{close}. \tag{16.54}$$

Or binary: $a_{\text{gripper}} \in \{0, 1\}$ (open/close).

Some systems use **continuous grasp force**:

$$F_{\text{grasp}} = c_1 \cdot a_{\text{gripper}} + c_2, \quad \text{clamped to } [F_{\min}, F_{\max}]. \tag{16.55}$$

16.7 Distribution Over Trajectories: The Core Challenge

16.7.1 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

$$\mathbf{A}_{t:t+k} = [\mathbf{a}_t, \mathbf{a}_{t+1}, \ldots, \mathbf{a}_{t+k-1}] \in \mathbb{R}^{k \times D}. \tag{16.56}$$

The policy models:

$$P(\mathbf{A}_{t:t+k}|\mathbf{o}_{0:t}) : \text{probability of a coherent sequence of } k \text{ actions}. \tag{16.57}$$

For multimodal distributions, this can be a mixture:

$$P(\mathbf{A}_{t:t+k}|\mathbf{o}_{0:t}) = \sum_{m=1}^{M} w_m \mathcal{N}(\mathbf{A}_{t:t+k}; \boldsymbol{\mu}_m(\mathbf{o}_{0:t}), \Sigma_m(\mathbf{o}_{0:t})), \tag{16.58}$$

where each mode m represents a distinct manipulation strategy (left-grasp, right-grasp, etc.).

16.7.2 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**:

$$\mathcal{R}_{\text{smooth}} = \lambda \sum_{t=0}^{k-2} \|\mathbf{a}_{t+1} - \mathbf{a}_t\|_2^2. \tag{16.59}$$

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).

16.8 Imitation Learning Framework

16.8.1 Behavioral Cloning (Baseline)

The simplest approach to learning π from demonstrations $\mathcal{D} = \{(\mathbf{o}_i, \mathbf{a}_i^*)\}_{i=1}^N$:

Definition 16.9 (Behavioral Cloning). Minimize the supervised loss

$$\mathcal{L}_{\text{BC}} = \mathbb{E}_{(\mathbf{o},\mathbf{a}^*)\sim\mathcal{D}} \left[\ell(\mathbf{a}^*_{\text{true}}, \mathbf{a}^*_{\text{pred}}(\mathbf{o};\theta))\right], \tag{16.60}$$

where ℓ is a regression loss (MSE, L_1, etc.).

Limitations:

1. **Multimodality:** Cannot capture multiple valid modes; averages them destructively
2. **Distribution shift:** Training on $\mathcal{D}_{\text{demo}}$; deployment distribution differs
3. **Compounding errors:** In longer rollouts, small errors accumulate

16.8.2 Conditional Distribution Matching

To handle multimodality rigorously, we match the learned distribution to the empirical distribution of demonstrations:

Definition 16.10 (Conditional Distribution Matching). Instead of fitting a single mean, fit the full conditional distribution:

$$\mathcal{L}_{\text{dist}} = \mathbb{E}_{(\mathbf{o},\mathbf{a}^*)\sim\mathcal{D}} \left[D_{\text{div}}\left(P_{\text{demo}}(\mathbf{a}|\mathbf{o}), P_\theta(\mathbf{a}|\mathbf{o})\right)\right], \tag{16.61}$$

where D_{div} is a divergence measure (KL, Wasserstein, etc.).

For a Gaussian policy with diagonal covariance:

$$\mathcal{L}_{\text{Gaussian}} = \mathbb{E}\left[-\log \mathcal{N}(\mathbf{a}^*; \boldsymbol{\mu}(\mathbf{o};\theta), \text{diag}(\boldsymbol{\sigma}^2(\mathbf{o};\theta)))\right]. \tag{16.62}$$

This learns both mean and variance, allowing the policy to "express uncertainty" in ambiguous situations.

16.8.3 Latent Variable Models (ACT / Diffusion)

To fully capture multimodal structure, introduce a **latent variable z**:

Definition 16.11 (Latent Variable Policy).

$$P(\mathbf{a}_{t:t+k}|\mathbf{o}_{0:t}) = \int P(\mathbf{a}_{t:t+k}|\mathbf{z}, \mathbf{o}_{0:t})P(\mathbf{z}|\mathbf{o}_{0:t})d\mathbf{z}. \tag{16.63}$$

This is **parameterized** by:

- **Decoder:** $P_\theta(\mathbf{a}_{t:t+k}|\mathbf{z}, \mathbf{o}_{0:t})$ (deterministic or with small variance)
- **Prior:** $P(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$

During training, we infer the posterior:

$$P_\phi(\mathbf{z}|\mathbf{a}_{t:t+k}, \mathbf{o}_{0:t}) \quad \text{(encoder, learned via CVAE or other methods).} \tag{16.64}$$

This framework allows the latent code **z** to **select among multiple modes** (different grasp strategies, approaching from different angles, etc.).

16.9 Key Differences from Supervised Learning: Summary Table

Aspect	**Supervised Learning (Vision/NLP)**	**Robot Learning (POM**
Environment	Static dataset	Dynamic, interactive worl
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 gene
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 (G
Temporal Structure	I.i.d. samples	**Temporal dependencies**
Evaluation	Accuracy/perplexity on test set	Success rate in sim/real; z

16.10 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
- **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

Action Chunking Transformers (ACT)

17.1 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.

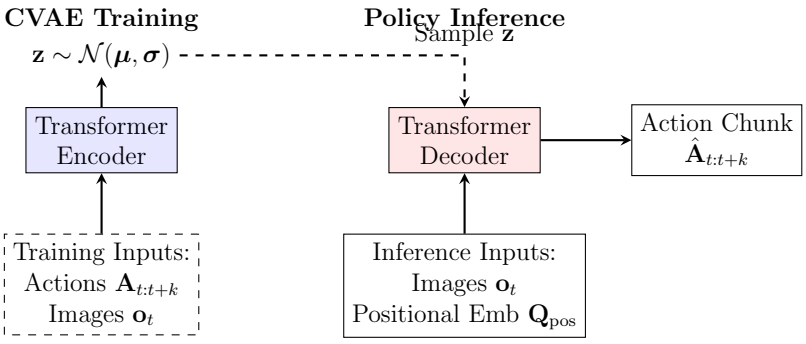

Figure 17.1: ACT Architecture. Left: During training, the CVAE encoder compresses future actions and observations into a latent $\mathbf{z}$. Right: During inference, the decoder uses sampled $\mathbf{z}$ and current observations to generate the action chunk.

17.2 Action Chunking: Problem Formulation

17.2.1 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:

$$\mathbf{A}_t = [\hat{\mathbf{a}}_t, \hat{\mathbf{a}}_{t+1}, \ldots, \hat{\mathbf{a}}_{t+k-1}] \in \mathbb{R}^{k \times D}, \tag{17.1}$$

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.

17.2.2 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.

17.3 Conditional Variational Autoencoder (CVAE) for Distribution Learning

17.3.1 Problem Statement: Multimodal Distribution Modeling

We want to learn a **rich conditional distribution** over action chunks:

$$P(\mathbf{A}_{t:t+k} | \mathbf{o}_{0:t}). \tag{17.2}$$

A standard regression approach (MSE) fails for multimodal data:

$$\min_{\theta} \mathbb{E}\left[\|\mathbf{A}^* - f_\theta(\mathbf{o}_{0:t})\|_2^2\right] \quad \Rightarrow \quad f_\theta \text{ learns the mean, collapsing modes.} \tag{17.3}$$

We need a **generative model** that can capture multiple modes.

17.3.2 CVAE Formulation: Evidence Lower Bound (ELBO)

Definition 17.2 (CVAE). A Conditional VAE models the joint density:

$$P_\theta(\mathbf{A}|\mathbf{o}) = \int P_\theta(\mathbf{A}|\mathbf{z},\mathbf{o})P(\mathbf{z})d\mathbf{z}, \tag{17.4}$$

where:

- $\mathbf{z} \in \mathbb{R}^{d_z}$: Latent variable (typically $d_z \in [16, 64]$)
- $P_\theta(\mathbf{A}|\mathbf{z},\mathbf{o})$: Decoder (learned; typically Gaussian)
- $P(\mathbf{z})$: Prior (standard normal: $\mathcal{N}(\mathbf{0},\mathbf{I})$)

During training, we condition on **both** the true action $\mathbf{A}^*$ and observation $\mathbf{o}$ to infer the posterior:

$$q_\phi(\mathbf{z}|\mathbf{A}^*,\mathbf{o}) \tag{17.5}$$

where ϕ are encoder parameters.

Derivation of ELBO:

Starting from the marginal likelihood:

$$\log P_\theta(\mathbf{A}|\mathbf{o}) = \log \int P_\theta(\mathbf{A}|\mathbf{z},\mathbf{o})P(\mathbf{z})d\mathbf{z}. \tag{17.6}$$

By Jensen's inequality (convexity of log), for any distribution $q_\phi(\mathbf{z}|\mathbf{A},\mathbf{o})$:

$$\log P_\theta(\mathbf{A}|\mathbf{o}) \geq \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{A},\mathbf{o})}\left[\log P_\theta(\mathbf{A}|\mathbf{z},\mathbf{o})\right] - D_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{A},\mathbf{o})\|P(\mathbf{z})). \tag{17.7}$$

This is the **ELBO (Evidence Lower Bound)**:

$$\mathcal{L}_{\mathrm{ELBO}}(\theta,\phi;\mathbf{A},\mathbf{o}) = \underbrace{\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{A},\mathbf{o})}[\log P_\theta(\mathbf{A}|\mathbf{z},\mathbf{o})]}_{\text{Reconstruction}} - \underbrace{D_{\mathrm{KL}}(q_\phi\|P)}_{\text{Regularization}}. \tag{17.8}$$

The training objective is:

$$\mathcal{L} = -\mathcal{L}_{\mathrm{ELBO}} = -\mathbb{E}_{q_\phi}[\log P_\theta(\mathbf{A}|\mathbf{z},\mathbf{o})] + D_{\mathrm{KL}}(q_\phi\|P). \tag{17.9}$$

In practice, we add a weighting coefficient β to balance the two terms:

$$\mathcal{L}_{\mathrm{total}} = \mathcal{L}_{\mathrm{rec}} + \beta \cdot \mathcal{L}_{\mathrm{KL}}, \tag{17.10}$$

where $\beta \in [0.1, 10]$ is a hyperparameter.

17.3.3 Concrete Loss Functions

Reconstruction Loss

Assuming the decoder outputs a Gaussian with diagonal covariance:

$$P_\theta(\mathbf{A}|\mathbf{z},\mathbf{o}) = \mathcal{N}(\mathbf{A};\boldsymbol{\mu}_\theta(\mathbf{z},\mathbf{o}),\mathrm{diag}(\boldsymbol{\sigma}^2_\theta(\mathbf{z},\mathbf{o}))). \tag{17.11}$$

The negative log-likelihood (reconstruction loss) is:

$$\mathcal{L}_{\mathrm{rec}} = -\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{A},\mathbf{o})}\left[\log \mathcal{N}(\mathbf{A};\boldsymbol{\mu},\boldsymbol{\sigma}^2)\right], \tag{17.12}$$

$$= \mathbb{E}_{q_\phi}\left[\frac{1}{2}\log|2\pi\text{diag}(\boldsymbol{\sigma}^2)| + \frac{1}{2}\|\boldsymbol{\sigma}^{-1}(\mathbf{A}-\boldsymbol{\mu})\|_2^2\right], \tag{17.13}$$

$$= \mathbb{E}_{q_\phi}\left[\sum_{i,j}\log\sigma_{ij} + \frac{1}{2}\sum_{i,j}\frac{(A_{ij}-\mu_{ij})^2}{\sigma_{ij}^2}\right]. \tag{17.14}$$

Simplified Form (Fixed Variance): If we fix $\boldsymbol{\sigma}$ to a constant $\sigma = 1$ (learned variance loss), the loss simplifies to MSE:

$$\mathcal{L}_{\text{rec}} = \frac{1}{kD}\|\mathbf{A} - \boldsymbol{\mu}_\theta(\mathbf{z},\mathbf{o})\|_F^2, \tag{17.15}$$

where $\|\cdot\|_F$ is the Frobenius norm and the factor $\frac{1}{kD}$ is for normalization.

KL Divergence Loss

For Gaussian posteriors and priors:

$$q_\phi(\mathbf{z}|\mathbf{A},\mathbf{o}) = \mathcal{N}(\mathbf{z};\boldsymbol{\mu}_\phi(\mathbf{A},\mathbf{o}),\text{diag}(\boldsymbol{\sigma}_\phi^2(\mathbf{A},\mathbf{o}))), \tag{17.16}$$

$$P(\mathbf{z}) = \mathcal{N}(\mathbf{z};\mathbf{0},\mathbf{I}), \tag{17.17}$$

the KL divergence has a closed-form solution:

$$D_{\text{KL}}(q_\phi\|P) = \frac{1}{2}\sum_{j=1}^{d_z}\left(\mu_{\phi,j}^2 + \sigma_{\phi,j}^2 - \log\sigma_{\phi,j}^2 - 1\right). \tag{17.18}$$

Interpretation:

- If $\boldsymbol{\mu}_\phi \approx \mathbf{0}$ and $\boldsymbol{\sigma}_\phi \approx 1$, the posterior is close to the prior, and $D_{\text{KL}} \approx 0$.
- If the posterior diverges, the penalty increases.

The hyperparameter β controls the strength of this regularization:

- β **too small** (< 0.1): Encoder is ignored; posterior collapses to prior
- β **too large** (> 100): Reconstruction is sacrificed for regularization; actions become too smooth/averaged

Annealing Strategy: To avoid posterior collapse early in training, use **KL annealing**:

$$\beta(t) = \min\left(1, \frac{t}{t_{\text{anneal}}}\right)\cdot\beta_{\text{target}}, \tag{17.19}$$

where t is the training step and t_{anneal} is the number of steps to linearly increase β from 0 to β_{target} (e.g., $\beta_{\text{target}} = 10$, $t_{\text{anneal}} = 10000$).

17.4 Encoder: Inferring the Posterior

17.4.1 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:

$$\mathbf{A}_{\text{embed}} = \mathbf{A}\mathbf{W}_{\text{act}} + \mathbf{b}_{\text{act}}, \quad \mathbf{A}_{\text{embed}} \in \mathbb{R}^{B\times k\times d_{\text{model}}}, \tag{17.20}$$

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:

$$\mathbf{A}^{(i)}_{\text{embed}} \leftarrow \mathbf{A}^{(i)}_{\text{embed}} + \mathbf{e}^{\text{pos}}_i, \quad \mathbf{e}^{\text{pos}}_i \in \mathbb{R}^{d_{\text{model}}}. \tag{17.21}$$

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

$$\mathbf{H}_{\text{input}} = [\text{Token}_{[\text{CLS}]}; \text{Token}_{\text{prop}}; \text{Tokens}_{\text{vis}}; \mathbf{A}_{\text{embed}}] \in \mathbb{R}^{B\times(1+1+N_{\text{vis}}+k)\times d_{\text{model}}}. \tag{17.22}$$

Example dimensions:

$$\mathbf{H}_{\text{input}} \in \mathbb{R}^{8\times 128\times 512} \quad (\text{for } B=8, k=100, N_{\text{vis}}=18, d_{\text{model}}=512). \tag{17.23}$$

Step 4: Transformer Encoder Pass through L self-attention blocks:

$$\mathbf{H}^{(0)} = \mathbf{H}_{\text{input}} \tag{17.24}$$

$$\mathbf{H}^{(\ell)} = \text{TransformerBlock}_\ell(\mathbf{H}^{(\ell-1)}), \quad \ell = 1,\dots,L \tag{17.25}$$

$$\mathbf{H}_{\text{output}} = \mathbf{H}^{(L)} \in \mathbb{R}^{B\times 128\times 512}. \tag{17.26}$$

Step 5: Pooling to Latent Parameters Extract the [CLS] token output (first position):

$$\mathbf{h}_{\text{cls}} = \mathbf{H}_{\text{output}}[:, 0, :] \in \mathbb{R}^{B\times d_{\text{model}}} = \mathbb{R}^{B\times 512}. \tag{17.27}$$

Or use mean pooling over all tokens:

$$\mathbf{h}_{\text{pool}} = \frac{1}{128}\sum_{i=0}^{127}\mathbf{H}_{\text{output}}[:, i, :] \in \mathbb{R}^{B\times 512}. \tag{17.28}$$

Step 6: Output Linear Layers

$$\boldsymbol{\mu}_\phi = \text{Linear}_{d_{\text{model}}\to d_z}(\mathbf{h}_{\text{pool}}) \in \mathbb{R}^{B\times d_z} = \mathbb{R}^{B\times 32}, \tag{17.29}$$

$$\log \boldsymbol{\sigma}^2_\phi = \text{Linear}_{d_{\text{model}}\to d_z}(\mathbf{h}_{\text{pool}}) \in \mathbb{R}^{B\times 32}. \tag{17.30}$$

17.4.2 Self-Attention Mechanics in the Encoder

Query, Key, Value Projections:

$$\mathbf{Q} = \mathbf{H}^{(\ell-1)}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{H}^{(\ell-1)}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{H}^{(\ell-1)}\mathbf{W}_V, \tag{17.31}$$

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:

$$\text{Attn} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right) \in \mathbb{R}^{B\times 128\times 128}. \tag{17.32}$$

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:

$$\mathbf{H}^{(\ell)} = \text{Attn}\cdot\mathbf{V} + \text{FFN}(\cdot), \tag{17.33}$$

where FFN is a feed-forward network (position-wise MLP).

17.5 Decoder: Generating Action Chunks from Latent Code

17.5.1 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):

$$\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \quad \mathbf{z} \in \mathbb{R}^{B\times d_z} = \mathbb{R}^{B\times 32}. \tag{17.34}$$

Project to embedding space:

$$\mathbf{z}_{\text{embed}} = \text{Linear}_{d_z \to d_{\text{model}}}(\mathbf{z}) \in \mathbb{R}^{B \times d_{\text{model}}} = \mathbb{R}^{B \times 512}. \tag{17.35}$$

Expand to match action sequence length (by tiling or concatenation):

$$\mathbf{z}_{\text{tiled}} = \text{Repeat}(\mathbf{z}_{\text{embed}}, k) \in \mathbb{R}^{B \times k \times d_{\text{model}}} = \mathbb{R}^{B \times 100 \times 512}. \tag{17.36}$$

Step 2: Query (Action Positions) Learnable position embeddings for the k actions:

$$\mathbf{Q}_{\text{pos}} \in \mathbb{R}^{B \times k \times d_{\text{model}}} = \mathbb{R}^{B \times 100 \times 512}. \tag{17.37}$$

These are shared across all samples in the batch.

Step 3: Key/Value from Observation Concatenate vision tokens and latent code:

$$\mathbf{K}_{\text{cond}} = \text{Concat}([\text{Tokens}_{\text{vis}}, \mathbf{z}_{\text{embed}}]) \in \mathbb{R}^{B \times (N_{\text{vis}}+1) \times d_{\text{model}}} = \mathbb{R}^{B \times 19 \times 512}. \tag{17.38}$$

$$\mathbf{V}_{\text{cond}} = \mathbf{K}_{\text{cond}} \quad \text{(same, or learned projection)}. \tag{17.39}$$

Step 4: Self-Attention (over action sequence) Allows each action position to "see" previous/future actions (causal masking if autoregressive):

$$\text{Attn}_{\text{self}} = \text{softmax}\left(\frac{\mathbf{Q}_{\text{pos}}\mathbf{Q}_{\text{pos}}^\top}{\sqrt{d_k}} + \text{Mask}\right)\mathbf{V}_{\text{pos}}. \tag{17.40}$$

For typical ACT, bidirectional attention is used (not causal), since we can see the entire chunk:

$$\text{Attn}_{\text{self}} = \text{softmax}\left(\frac{\mathbf{Q}_{\text{pos}}\mathbf{Q}_{\text{pos}}^\top}{\sqrt{d_k}}\right)\mathbf{Q}_{\text{pos}}. \tag{17.41}$$

Step 5: Cross-Attention (to observation) Each action attends to observation features:

$$\text{Attn}_{\text{cross}} = \text{softmax}\left(\frac{\mathbf{Q}_{\text{pos}}\mathbf{K}_{\text{cond}}^\top}{\sqrt{d_k}}\right)\mathbf{V}_{\text{cond}} \in \mathbb{R}^{B \times k \times d_{\text{model}}}. \tag{17.42}$$

Dimension check:

$$\frac{\mathbf{Q}_{\text{pos}}\mathbf{K}_{\text{cond}}^\top}{\sqrt{d_k}} \in \mathbb{R}^{B \times k \times (N_{\text{vis}}+1)} = \mathbb{R}^{B \times 100 \times 19}. \tag{17.43}$$

After softmax and multiplication by $\mathbf{V}_{\text{cond}} \in \mathbb{R}^{B \times 19 \times 512}$:

$$\text{Attn}_{\text{cross}} \in \mathbb{R}^{B \times 100 \times 512}. \tag{17.44}$$

Step 6: Output Projection

$$\hat{\mathbf{A}} = \text{MLP}_{\text{out}}(\text{Attn}_{\text{cross}}) \in \mathbb{R}^{B \times k \times D} = \mathbb{R}^{B \times 100 \times 7}. \tag{17.45}$$

The MLP projects from $d_{\text{model}} = 512$ to $D = 7$ (action dimension).

17.5.2 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:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H)\mathbf{W}_O, \tag{17.46}$$

where each head computes:

$$\text{head}_h = \text{softmax}\left(\frac{\mathbf{Q}_h \mathbf{K}_h^\top}{\sqrt{d_k}}\right)\mathbf{V}_h, \tag{17.47}$$

$$\mathbf{Q}_h = \mathbf{Q}\mathbf{W}_Q^{(h)}, \quad d_k = \frac{d_{\text{model}}}{H}. \tag{17.48}$$

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)

17.6 Reparameterization Trick and Backpropagation

17.6.1 Reparameterization Trick

The sampling operation $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{A}, \mathbf{o})$ is **stochastic**, which breaks gradient flow during backpropagation. The reparameterization trick enables gradients to flow through the sampling operation.

Definition 17.6 (Reparameterization Trick). Instead of sampling $\mathbf{z}$ directly from q_ϕ, we write:

$$\mathbf{z} = \boldsymbol{\mu}_\phi(\mathbf{A}, \mathbf{o}) + \boldsymbol{\sigma}_\phi(\mathbf{A}, \mathbf{o}) \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{17.49}$$

where $\odot$ is element-wise multiplication.

This reformulates $\mathbf{z}$ as a **deterministic function** of $(\boldsymbol{\mu}, \boldsymbol{\sigma}, \boldsymbol{\epsilon})$, allowing the chain rule to be applied:

$$\frac{\partial \mathbf{z}_j}{\partial \mu_\phi^{(i)}} = \delta_{ij}, \quad \frac{\partial \mathbf{z}_j}{\partial \sigma_\phi^{(i)}} = \epsilon_i \delta_{ij}. \tag{17.50}$$

Gradient Flow:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}_\phi} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \boldsymbol{\mu}_\phi} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}}, \tag{17.51}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\sigma}_\phi} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \boldsymbol{\sigma}_\phi} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}} \odot \boldsymbol{\epsilon}. \tag{17.52}$$

17.6.2 Gradient Computation for ELBO Terms

Reconstruction Loss Gradient (via decoder):

$$\frac{\partial \mathcal{L}_{\text{rec}}}{\partial \hat{\mathbf{A}}} = 2(\hat{\mathbf{A}} - \mathbf{A}_{\text{true}}) \in \mathbb{R}^{B\times k\times D}. \tag{17.53}$$

This flows back through:

$$\text{Action Projection} \to \text{Cross-Attention} \to \text{Self-Attention} \to \text{Latent Projection} \to \text{Reparameterization} \to \tag{17.54}$$

KL Divergence Gradient (via encoder): The KL loss depends on $\boldsymbol{\mu}_\phi$ and $\boldsymbol{\sigma}_\phi$ directly:

$$\mathcal{L}_{\text{KL}} = \frac{1}{2}\sum_j (\mu_{\phi,j}^2 + \sigma_{\phi,j}^2 - \log \sigma_{\phi,j}^2 - 1). \tag{17.55}$$

Gradients:

$$\frac{\partial \mathcal{L}_{\text{KL}}}{\partial \mu_{\phi,j}} = \mu_{\phi,j}, \tag{17.56}$$

$$\frac{\partial \mathcal{L}_{\text{KL}}}{\partial \sigma_{\phi,j}^2} = \frac{1}{2}\left(1 - \frac{1}{\sigma_{\phi,j}^2}\right). \tag{17.57}$$

These are "direct" penalties that encourage the posterior to stay close to the standard normal prior.

17.6.3 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):

$$\phi_{t+1} = \phi_t - \alpha_{\text{enc}} m_t^{(\phi)} / (\sqrt{v_t^{(\phi)}} + \epsilon), \tag{17.58}$$

$$\theta_{t+1} = \theta_t - \alpha_{\text{dec}} m_t^{(\theta)} / (\sqrt{v_t^{(\theta)}} + \epsilon), \tag{17.59}$$

where $\alpha_{\text{enc}}, \alpha_{\text{dec}}$ are learning rates (potentially different).

17.7 Temporal Ensembling and Smooth Action Execution

17.7.1 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.

17.7.2 Exponential Moving Average (EMA) Ensembling

Definition 17.7 (Temporal Ensembling). For each action index i, collect all predictions and form a weighted average:

$$\mathbf{a}_i^{\text{final}} = \frac{\sum_{\tau=0}^{k-1} w_\tau \hat{\mathbf{a}}_i^{(\text{chunk at } i-\tau)}}{\sum_{\tau=0}^{k-1} w_\tau}, \tag{17.60}$$

where the weights decay exponentially:

$$w_\tau = \exp(-\lambda\tau), \quad \lambda > 0 \text{ (decay constant, e.g., } \lambda = 0.1). \tag{17.61}$$

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.

17.7.3 Smoothness Guarantee

Temporal ensembling provides **implicit smoothness regularization**. The executed actions are:

$$\mathbf{a}_i^{\text{final}} = \frac{1}{Z} \sum_\tau w_\tau \hat{\mathbf{a}}_i^{(\tau)}, \quad Z = \sum_\tau w_\tau. \tag{17.62}$$

If the policy predicts $\hat{\mathbf{a}}_i \approx \hat{\mathbf{a}}_{i+1}$ (smooth predictions), then:

$$\mathbf{a}_{i+1}^{\text{final}} - \mathbf{a}_i^{\text{final}} \approx \text{small}. \tag{17.63}$$

Even if individual chunks are slightly jerky, ensemble averaging smooths out discontinuities.

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

17.8 Handling Multimodality: How ACT Solves the Averaging Problem

17.8.1 Revisiting the Problem

Recall from Chapter 16: averaging demonstrations from multiple modes produces **invalid trajectories**.

Example: Cup grasping with 3 modes

Why ACT Works:

The CVAE introduces a **discrete choice variable** (the latent code) that selects among modes. During training:

$$\mathbf{z}^{(1)} \to \text{Mode 1 (Left)}, \quad \mathbf{z}^{(2)} \to \text{Mode 2 (Right)}, \quad \ldots \tag{17.64}$$

The encoder learns to infer which mode the demonstration follows, and the decoder learns to reconstruct each mode faithfully.

17.8.2 Mathematical Characterization

Definition 17.8 (Mixture Interpretation). The learned conditional distribution can be written as an (implicit) mixture:

$$P_\theta(\mathbf{A}|\mathbf{o}) = \int P_\theta(\mathbf{A}|\mathbf{z}, \mathbf{o}) P(\mathbf{z}) d\mathbf{z} \tag{17.65}$$

$$\approx \sum_{m=1}^{M} P(\mathbf{z} = \mathbf{z}_m|\mathbf{o}) \cdot \mathcal{N}(\mathbf{A}; \boldsymbol{\mu}_m(\mathbf{o}), \Sigma_m), \tag{17.66}$$

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)

17.8.3 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:

$$\mathbf{z}_L \sim \mathcal{N}(\mu_L, \sigma_L) \Rightarrow \text{Left-side coordination} \tag{17.67}$$

$$\mathbf{z}_R \sim \mathcal{N}(\mu_R, \sigma_R) \Rightarrow \text{Right-side coordination} \tag{17.68}$$

17.9 Practical Considerations: Hyperparameter Selection

17.9.1 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.

17.9.2 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.

17.9.3 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:

$$\beta(e) = \min\left(1, \frac{e}{e_{\text{warmup}}}\right) \cdot \beta_{\text{target}}, \quad e \in \{1, 2, \ldots, E\}. \tag{17.69}$$

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)

17.9.4 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):

$$\alpha_{\text{backbone}} = 0.1 \times \alpha_{\text{decoder}}. \tag{17.70}$$

17.10 Training Procedure: Full Algorithm

17.10.1 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
```

17.10.2 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**

17.11 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:

$$\mathcal{L}_{\text{total}} = \underbrace{\|\mathbf{A} - \hat{\mathbf{A}}\|_2^2}_{\text{Reconstruction}} + \beta \underbrace{\frac{1}{2}\sum_j (\mu_j^2 + \sigma_j^2 - \log \sigma_j^2 - 1)}_{\text{KL Regularization}}, \tag{17.71}$$

$$\mathbf{z} = \boldsymbol{\mu}_\phi + \boldsymbol{\sigma}_\phi \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{17.72}$$

$$\hat{\mathbf{A}} = \text{Decoder}_\theta(\mathbf{z}, \text{Vision Features}, \mathbf{o}). \tag{17.73}$$

This framework is the foundation for later chapters: Diffusion Policy (alternative to CVAE), VLA (unified token space), and the numerical walkthrough (Chapter 20).

Chapter 18

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.

18.1 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).

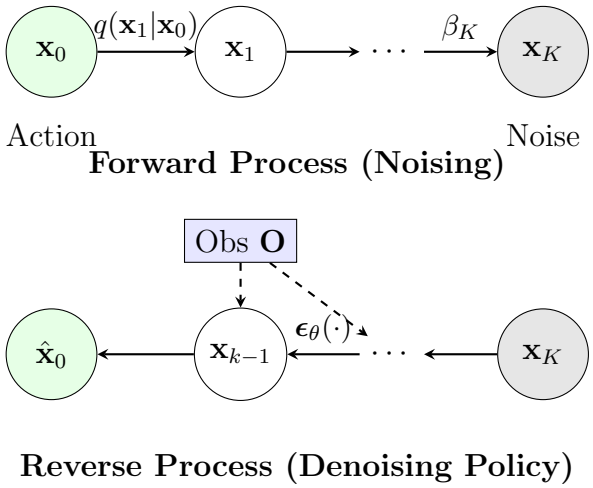

Figure 18.1: Diffusion Policy. Top: The forward process gradually destroys structure by adding noise. Bottom: The reverse process learns to remove noise, conditioned on observations $\mathbf{O}$, to recover the action $\mathbf{x}_0$.

18.2 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 $k = 1, \ldots, K$:

$$q(\mathbf{x}_k|\mathbf{x}_{k-1}) = \mathcal{N}\left(\mathbf{x}_k; \sqrt{1-\beta_k}\mathbf{x}_{k-1}, \beta_k\mathbf{I}\right) \tag{18.1}$$

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 $\bar{\alpha}_k = \prod_{s=1}^k \alpha_s$.

$$q(\mathbf{x}_k|\mathbf{x}_0) = \mathcal{N}\left(\mathbf{x}_k; \sqrt{\bar{\alpha}_k}\mathbf{x}_0, (1-\bar{\alpha}_k)\mathbf{I}\right) \tag{18.2}$$

Thus:

$$\mathbf{x}_k = \sqrt{\bar{\alpha}_k}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_k}\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{18.3}$$

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.

18.3 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:

$$\boldsymbol{\mu}_\theta(\mathbf{x}_k, k, \mathbf{O}) = \frac{1}{\sqrt{\alpha_k}}\left(\mathbf{x}_k - \frac{\beta_k}{\sqrt{1-\bar{\alpha}_k}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_k, k, \mathbf{O})\right) \tag{18.4}$$

Instead of predicting the mean directly, we train the network to predict the **noise** $\boldsymbol{\epsilon}$ that was added.

18.4 Training Objective

The loss function is the Mean Squared Error (MSE) between the actual noise $\boldsymbol{\epsilon}$ and the predicted noise $\boldsymbol{\epsilon}_\theta$:

$$\mathcal{L}_{\text{diff}} = \mathbb{E}_{k\sim\mathcal{U}(1,K),\mathbf{x}_0\sim\mathcal{D},\boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}\left[\left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\underbrace{\sqrt{\bar{\alpha}_k}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_k}\boldsymbol{\epsilon}}_{\mathbf{x}_k}, k, \mathbf{O})\right\|^2\right] \tag{18.5}$$

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.

18.5 Network Architectures

Two main architectures are dominant in Diffusion Policy: **CNN-based (1D U-Net)** and **Transformer-based**.

18.5.1 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.

$$\text{FiLM}(\mathbf{h}; \gamma, \beta) = \gamma(\mathbf{e}_{\text{obs}}) \odot \mathbf{h} + \beta(\mathbf{e}_{\text{obs}}) \tag{18.6}$$

- **Structure:** Downsamples temporal resolution (Conv1D $\rightarrow$ Pooling) to capture high-level structure, then upsamples (ConvTranspose1D) to reconstruct details.

18.5.2 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.

18.6 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 $\mathbf{x}_0$:**

$$\hat{\mathbf{x}}_0 = \frac{\mathbf{x}_k - \sqrt{1 - \bar{\alpha}_k}\boldsymbol{\epsilon}_\theta(\mathbf{x}_k, k, \mathbf{O})}{\sqrt{\bar{\alpha}_k}} \tag{18.7}$$

2. **Point to next state $\mathbf{x}_{k'}$:**

$$\mathbf{x}_{k'} = \sqrt{\bar{\alpha}_{k'}}\hat{\mathbf{x}}_0 + \sqrt{1 - \bar{\alpha}_{k'} - \sigma_{k'}^2}\boldsymbol{\epsilon}_\theta(\mathbf{x}_k, k, \mathbf{O}) + \sigma_{k'}\boldsymbol{\epsilon}' \tag{18.8}$$

Setting $\sigma_{k'} = 0$ yields a deterministic sampling process (ODE flow), which is often preferred for stability in robotics.

18.7 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)

18.8 Why Diffusion for Robotics?

18.8.1 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.

18.8.2 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.

18.8.3 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 19

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.

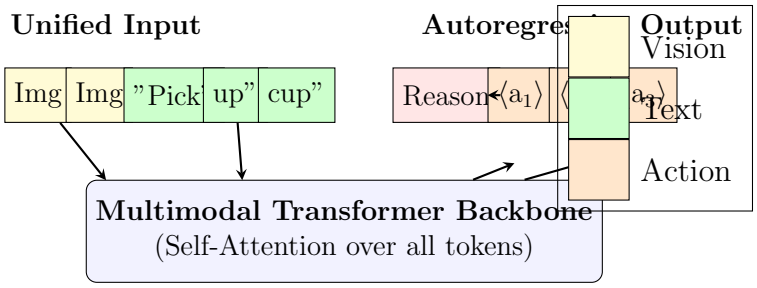

Figure 19.1: VLA Architecture (RT-2). Vision, Text, and Actions are treated as a unified sequence of tokens. The model generates reasoning (Chain-of-Thought) followed by discretized action tokens.

19.1 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:

$$\mathcal{V} = \mathcal{V}_{\text{text}} \cup \mathcal{V}_{\text{image}} \cup \mathcal{V}_{\text{action}}, \tag{19.1}$$

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 $|\mathcal{V}| = |\mathcal{V}_{\text{text}}| + |\mathcal{V}_{\text{image}}| + |\mathcal{V}_{\text{action}}|$.

19.2 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:

$$\phi(x) = \text{clip}\left(\left\lfloor \frac{x - a_{\min}}{a_{\max} - a_{\min}} \times N_{\text{bins}} \right\rfloor, 0, N_{\text{bins}} - 1 \right). \tag{19.2}$$

For a 7-DOF arm ($D = 7$) with $N_{\text{bins}} = 256$, the total action vocabulary size contribution is:

$$|\mathcal{V}_{\text{action}}| = D \times N_{\text{bins}} = 7 \times 256 = 1792 \text{ tokens}. \tag{19.3}$$

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):

$$\Delta\theta = \frac{2\pi}{256} \approx 0.0245 \text{ rad} \approx 1.4^\circ. \tag{19.4}$$

The expected quantization error (uniform distribution) is half the bin width: $\approx 0.7^\circ$. This provides sufficient precision for most manipulation tasks.

19.3 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 $P = 16$:

$$N_p = \frac{224}{16} \times \frac{224}{16} = 14 \times 14 = 196 \text{ patches}. \tag{19.5}$$

Each patch $\mathbf{x}_p$ is flattened and projected linearly to dimension d_{model}:

$$\mathbf{z}_p = \mathbf{x}_p \mathbf{W}_{\text{proj}} + \mathbf{e}_{\text{pos}}^{(p)}, \tag{19.6}$$

where $\mathbf{e}_{\text{pos}}$ is a learnable position embedding.

19.4 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:

$$\mathbf{h} = \mathbf{E}[s]. \tag{19.7}$$

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:

$$\mathbf{S}_{\text{in}} = [\underbrace{\mathbf{v}_1, \ldots, \mathbf{v}_{196}}_{\text{Image Patches}}, \quad \underbrace{t_1, t_2, t_3}_{\text{"Grasp the cup"}} \quad]. \tag{19.8}$$

19.5 Transformer-Based Action Generation

The model operates as a standard causal decoder.

Definition 19.6 (Forward Pass). Let $\mathbf{H}^{(0)} = [\mathbf{h}_1, \ldots, \mathbf{h}_T]$ comprise the embeddings of the input sequence. The sequence flows through L layers:

$$\mathbf{H}^{(\ell)} = \text{TransformerBlock}(\mathbf{H}^{(\ell-1)}), \quad \ell = 1, \ldots, L. \tag{19.9}$$

The final layer $\mathbf{H}^{(L)}$ is projected back to the vocabulary size to produce logits:

$$\mathbf{L} = \mathbf{H}^{(L)}\mathbf{W}_{\text{unembed}}, \quad \mathbf{L} \in \mathbb{R}^{T \times |\mathcal{V}|}. \tag{19.10}$$

The probability of the next token w_{t+1} given history $w_{1:t}$ is:

$$P(w_{t+1}|w_{1:t}) = \text{softmax}(\mathbf{L}_{t,:}). \tag{19.11}$$

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:

$$M_{i,j} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{otherwise} \end{cases}. \tag{19.12}$$

This enforces that position i can only attend to positions $j \leq i$.

Definition 19.8 (Language Modeling Loss). The objective function minimizes the negative log-likelihood of the ground truth sequence:

$$\mathcal{L}_{\text{VLA}} = -\sum_{t=1}^{T-1} \log P(w_{t+1}^{*}|w_1, \ldots, w_t), \tag{19.13}$$

where only the loss on $\mathcal{V}_{\text{action}}$ tokens (and optionally reasoning text) is typically back-propagated during fine-tuning on robot data.

19.6 Chain-of-Thought (CoT) Reasoning

VLA models can "think" before they act.

Definition 19.9 (Reasoning Tokens). Let $\mathbf{R} = [r_1, \ldots, r_m] \in \mathcal{V}_{\text{text}}^{m}$ be intermediate reasoning steps generated by the model before specifying actions.

Definition 19.10 (Reasoning-Action Sequence). The target output sequence becomes:

$$\mathbf{S}_{\text{out}} = [\underbrace{\text{"(1) Locate handle..."}}_{\text{Reasoning } \mathbf{R}}, \underbrace{\langle \text{act}_0 \rangle, \ldots, \langle \text{act}_{D-1} \rangle}_{\text{Action } \mathbf{A}}]. \tag{19.14}$$

Definition 19.11 (Conditional Action Loss). The probability of correct action depends explicitly on the reasoning trace:

$$P(\mathbf{A}|\text{Obs}) = P(\mathbf{A}|\mathbf{R}, \text{Obs}) \cdot P(\mathbf{R}|\text{Obs}). \tag{19.15}$$

Maximizing $P(\mathbf{R}|\text{Obs})$ forces the model to ground its plan in the observation before committing to motor commands.

19.7 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:

$$\hat{a}^{(d)} = a_{\min}^{(d)} + (k + 0.5) \times \frac{a_{\max}^{(d)} - a_{\min}^{(d)}}{N_{\text{bins}}}. \tag{19.16}$$

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.

19.8 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 19.1: 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.

19.9 Computational Complexity and Optimization

Theorem 19.2 (Attention Complexity). *The computational cost of the self-attention mechanism scales quadratically with sequence length N.*

$$\text{Cost} \propto O(N^2 d_{\text{model}}). \tag{19.17}$$

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:

$$\text{Cache}_t = \{(\mathbf{k}_i, \mathbf{v}_i) \mid i = 1 \dots t-1\}. \tag{19.18}$$

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.

19.10 Comparison and Summary

Table 19.2: 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)

19.11 Mathematical Summary

The transformation from observation to action in VLA is a chain of probabilities:

$$\mathbf{a}^* = \text{decode}\left(\arg\max_{w \in \mathcal{V}_{\text{action}}} P(w|\text{Encoder}(\mathbf{I}), \text{Tokenizer}(\text{Cmd}))\right). \tag{19.19}$$

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 20

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.

20.1 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:** $H = 8$.

20.2 Forward Pass: Complete Tensor Tracking

20.2.1 Vision Backbone (ResNet-18)

Input

Two images per sample, batch of 8.

$$\mathbf{I} \in \mathbb{R}^{B \times 2 \times 3 \times 224 \times 224} \tag{20.1}$$

Feature Exploration

The ResNet-18 removes the pooling layer and the final fully connected layer. Output of the last convolutional layer (layer4):

$$\text{Shape: } [B \cdot 2, 512, 7, 7] \tag{20.2}$$

We flatten the spatial dimensions to treat them as a sequence.

$$\text{Tokens}_{\text{vis}} \in \mathbb{R}^{(B\cdot 2)\times 49\times 512} \tag{20.3}$$

Reshape back to batch dimension:

$$\mathbf{F}_{\text{vis}} \in \mathbb{R}^{B\times(2\times 49)\times 512} = \mathbb{R}^{8\times 98\times 512} \tag{20.4}$$

We add a learnable 2D sinusoidal position embedding to these 98 tokens.

20.2.2 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.

$$\mathbf{X}_{\text{enc}} \in \mathbb{R}^{8\times 100\times 512} \tag{20.5}$$

Self-Attention

Standard Transformer Encoder steps (Self-Attention + MLP) for $L = 4$ layers. Output:

$$\mathbf{H}_{\text{enc}} \in \mathbb{R}^{8\times 100\times 512} \tag{20.6}$$

20.2.3 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 $\mathbf{z}$:

$$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}, \quad \mathbf{z} \in \mathbb{R}^{8\times 32} \tag{20.7}$$

20.2.4 Transformer Decoder (Policy)

Inputs

- **Query (Q):** Fixed positional embeddings representing the action chunk time steps $0 \dots 99$.
$$\mathbf{Q}_{\text{dec}} \in \mathbb{R}^{B \times 100 \times 512} \tag{20.8}$$
- **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:
$$\mathbf{K}, \mathbf{V} = \text{Concat}(\mathbf{H}_{\text{enc}}, \text{Project}(\mathbf{z})) \in \mathbb{R}^{8 \times 101 \times 512} \tag{20.9}$$

Cross-Attention Layers

The decoder attends to the perception features ($\mathbf{H}_{\text{enc}}$) conditioned on the "time queries".

$$\text{Attn}(\mathbf{Q}_{\text{dec}}, \mathbf{K}, \mathbf{V}) \to \mathbb{R}^{8 \times 100 \times 512} \tag{20.10}$$

Prediction Heads

A simple MLP projects the 512-dim tokens to action dimension 14.

$$\hat{\mathbf{A}} = \text{MLP}(\mathbf{H}_{\text{dec}}) \in \mathbb{R}^{8 \times 100 \times 14} \tag{20.11}$$

20.3 Computational Cost Analysis

20.3.1 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 ($\approx 80\%$).

20.3.2 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.

20.4 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	$\hat{\mathbf{A}}$	$(8, 100, 14)$