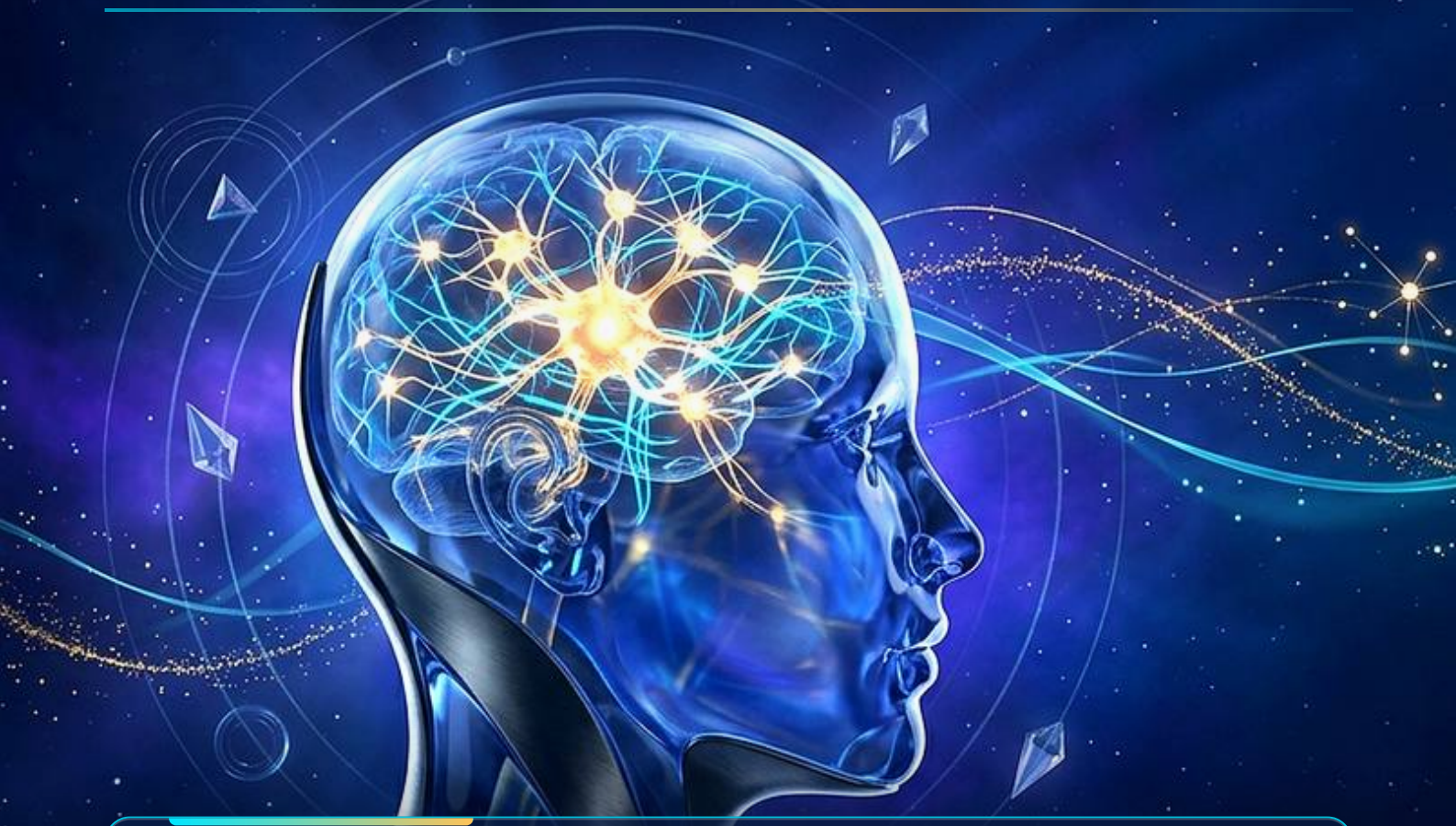
Electrical & Computer Engineering Department
UNIVERSITY OF PELOPONNESE

UNIVERSITY OF THE PELOPONNESE

Department of Electrical & Computer Engineering (ECE)



SEMINAR HANDBOOK & LABORATORY GUIDE

Artificial Intelligence (AI), Machine Learning, Deep Learning & Neural Networks

From the mathematical foundations of the artificial neuron and convolutional networks to the Transformer architecture, Large Language Models (LLMs), open-source models (Open-Source, LoRA, RAG & AI Agents), SETI Radio Astronomy, 33 applied laboratories in Python 3.12+, ethical governance, and quantum computing.

NEURAL NETWORKS | LLMS & OPEN-SOURCE | PYTHON 3.12 & SETI | QUANTUM & ETHICS

18 Modules
Comprehensive theoretical, mathematical, and architectural analysis with large, high-clarity diagrams.

33 Python Labs
Modern, documented codebases in NumPy, Keras 3, PyTorch 2, LoRA, RAG, MCP & SETI (code.zip).

Science & Humanity
Creative AI, "Starlight 1980", the EU AI Act, quantum computing, and "The Empty Seat".

Evangelos V. Pefanis
Laboratory Teaching Staff (E.T.E.P.), Dept. of Electrical & Computer Engineering, University of the Peloponnese

Educational Seminar Monograph
Patras • Companion Code: code.zip

INTRODUCTORY NOTE

Preface & Dedication

The spirit of research, acknowledgments, and a practical study guide for the seminar.

This monograph, the accompanying seminar presentation, and the source code of its specialized applications are the result of personal study, research, and systematic engagement with the fields of Artificial Intelligence, Machine Learning, and Deep Learning.

I dedicate this entire endeavor to the faculty, staff, and students of the University of the Peloponnese who honor me with their participation in this seminar, and especially to all those who seek the truth of the world and of things through research and every form of creative expression.

This work aspires to serve as a practical and substantive resource for students and academic staff alike, while also providing the spark for further engagement, experimentation, and scientific inquiry in this rapidly evolving field.

I extend my warm gratitude to the Chair of the Department of Electrical and Computer Engineering, Professor Sotirios Christodoulou, who supported the organization of this seminar and gave me the opportunity to advance this work, as well as to express myself creatively in developing the entire body of educational material.

Evangelos V. Pefanis
Laboratory Teaching Staff (E.T.E.P.), Dept. of Electrical & Computer Engineering, University of the Peloponnese

Study Guide & Making Use of the Companion Code (code.zip)

This handbook is structured into eighteen thematic modules and a philosophical epilogue, closely following the logical progression of the seminar presentation and instructional video. Each theoretical module connects directly to corresponding executable laboratory exercises in **Python 3.12+**, which are included in the companion archive `code.zip`.

Theory & Mathematics

Every architecture (Perceptron, MLP, CNN, RNN, LSTM, Transformer, LLM, LoRA, RAG, DQN) is examined through transparent mathematical equations and high-resolution diagrams.

33 Python Labs

Files `01_*.py` through `33_*.py` feature clear inline comments, modern APIs (Keras 3, PyTorch 2, Ollama, LoRA, RAG, MCP & SETI), and verified outputs.

Science & Vision

From GPU compute acceleration, open-source LLMs, and creative AI to the EU AI Act, logical qubits, and human responsibility.

Recommended Sequence for Laboratory Practice

Begin with the standalone NumPy scripts (`02_and_gate_numpy.py` , `06_xor_gate_numpy.py` , `22_transformer_self_attention.py` , and `23_bpe_tokenizer_and_sampling.py` through `29_ai_agent_tool_calling_react.py`), which run immediately on any computer without specialized hardware, and then advance to the deep learning models (MNIST, CIFAR-10, IMDb, Patras Weather, Tic-Tac-Toe DQN, and Creative AI) leveraging CUDA GPU acceleration wherever available.

MONOGRAPH STRUCTURE

Table of Contents

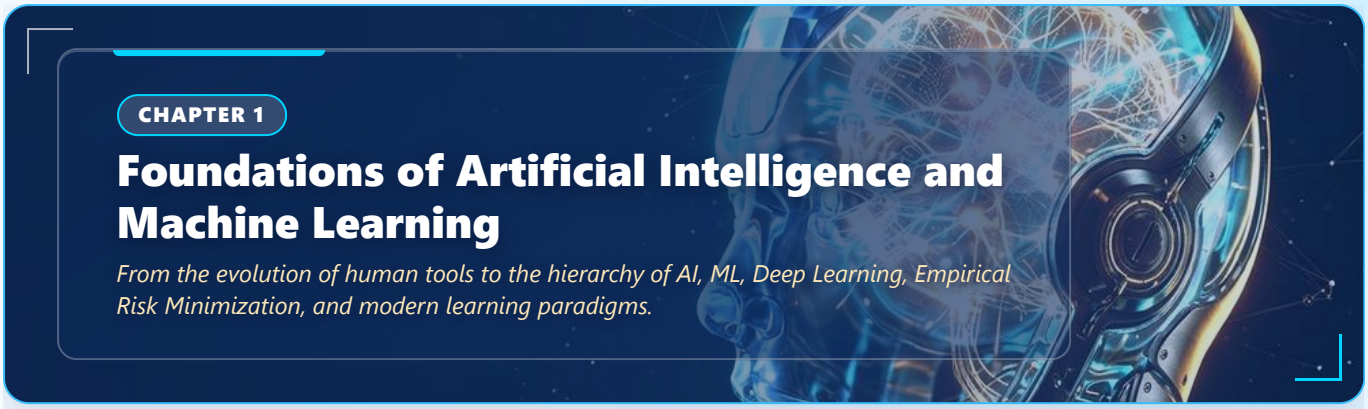
Eighteen thematic modules, 33 executable Python laboratories, and a human epilogue. Click any chapter row to jump directly to its page.

Section	Chapter Title	Page
Preface	Preface, Dedication & Study Guide	2
Chapter 1	Foundations of Artificial Intelligence and Machine Learning	4
Chapter 2	Biological and Artificial Neurons, Activation Functions	9
Chapter 3	Logic Gates, Mathematical Optimization & First Models	14
Chapter 4	Deep Architectures: MLP, CNN, RNN, LSTM, and ConvLSTM2D	26
Chapter 5	Transformer Architecture and Frontier AI Models	32
Chapter 6	Evaluation, Generalization, and Model Lifecycle	37
Chapter 7	Software Ecosystem, Visualization & Model Inspection	43
Chapter 8	GPU Compute Acceleration, CUDA & Development Environment	48
Chapter 9	LLM Internals: BPE, RoPE, KV-Cache & Decoding Sampling	53
Chapter 10	Open-Source AI Models, Ollama & 4-Bit Quantization	59
Chapter 11	Model Specialization, LoRA / QLoRA & Alignment	65
Chapter 12	RAG Systems, Vector Databases & Autonomous AI Agents (MCP)	70
Chapter 13	Applied Artificial Intelligence Labs (Vision, Language, Weather, RL)	76
Chapter 14	SETI Radio Astronomy & Training Astro-LLMs (Labs 30 to 33)	88
Chapter 15	Creative Intelligence: Music, Kaleidoscope & “Starlight 1980”	105
Chapter 16	Ethics, Governance, and the Scientific Revolution of AI	110
Chapter 17	Quantum Computing, Quantum AI & Post-Quantum Cryptography	114
Chapter 18	The Path to Mastery: Mathematics, Code & Research	119
Epilogue	“The Empty Seat” & Humanity and Intelligence Toward the Cosmos	122

Summary Map of Python Laboratories (code.zip)

Laboratory Track	Code Files	Libraries & Technologies
Neurons & Logic Gates	`01` to `08`	Tkinter, NumPy, h5py (H5Web), Keras 3, TensorBoard
Metrics & GPU Acceleration	`09` to `12`	NumPy, Pandas, PyTorch 2 (CUDA/AMP), TensorFlow / Keras 3
LLMs, Open-Source, LoRA, RAG & Agents	`23` to `29`	BPE, Mini-GPT, INT4 GGUF, Ollama, LoRA, Cosine RAG, ReAct/MCP
Applied & Creative AI	`13` to `22`	CNN, Bi-LSTM, ConvLSTM2D, Open-Meteo, DQN, MIDI, Transformers
SETI Radio Astronomy & Astro-LLMs	`30` to `33`	CSIRO simulateSearch, ABACAD, Spectral ViT, RTX 3060 Lab, LoRA & MCP

Quick Start for Python 3.12+ Environment: Extract `code.zip` and run sequentially: `python -m venv .venv` → `pip install -r requirements.txt` → `python 02_and_gate_numpy.py`



CHAPTER 1

Foundations of Artificial Intelligence and Machine Learning

From the evolution of human tools to the hierarchy of AI, ML, Deep Learning, Empirical Risk Minimization, and modern learning paradigms.

Learning Objectives & Theoretical Scope

- Distinguish rigorously among Artificial Intelligence, Machine Learning, Deep Learning, and Generative AI, contrasting hand-crafted features against end-to-end representation learning.
- Formulate any supervised learning task inside the unified mathematical framework of hypothesis classes ($\hat{y} = f_\theta(x)$), Empirical Risk Minimization (ERM), and structural regularization.
- Explain how architectural **inductive bias** constrains the search space to enable statistical generalization from finite datasets.
- Select the appropriate learning paradigm (Supervised, Unsupervised, Self-Supervised, or Reinforcement Learning) from the structure of available data and feedback signals.

1.1 The Evolution of Human Invention: From the Spear to the Cognitive Machine

Throughout human history, progress has been driven by crafting tools that extended our biological limits. From the earliest stone implements of the savanna to mechanical power, electricity, digital computers, and space exploration, every technological leap multiplied our physical or computational capacity. **Artificial Intelligence (AI)** represents the next qualitative leap: engineered systems that do not merely execute rigid, hand-coded instructions, but infer statistical regularities from data, adapt across unseen conditions, and support complex scientific and engineering decisions.

FIRST PRINCIPLES & INTUITION **How Can a Machine Learn Without Explicit Rules?**

Suppose you want a computer program to tell photos of ripe oranges apart from lemons. In **traditional rule-based software**, a programmer writes rigid instructions by hand (`if hue == yellow and aspect_ratio > 1.2: return 'lemon'`), which break immediately whenever lighting, shadows, or camera angles change. In **Machine Learning (ML)**, we flip the workflow completely: we feed the computer hundreds of labeled examples and let an algorithm tune internal numerical knobs called **parameters** (or **weights**) until its predictions agree with reality. **Deep Learning (DL)** stacks multiple layers of these knobs so the network discovers useful visual clues (edges, peel texture, curves) automatically from raw pixels.

- **Feature (x):** An input measurement describing an example (such as apartment area in m^2 or pixel brightness).
- **Label / Target (y):** The known ground-truth answer we want the system to predict.
- **Model (f_θ):** A mathematical function controlled by tunable parameters θ (read "theta") that outputs a prediction $\hat{y}$ (read "y-hat").
- **Training:** The automated process of measuring mistakes with a **loss function** and adjusting θ to make future guesses more accurate.



Figure 1.1a

The dawn of technology as an extension of human intent and observation.



Figure 1.1b

The continuity of exploration from Earth toward cosmic and computational frontiers.

1.2 The Hierarchy: AI, Machine Learning, Deep Learning & Representation Learning

In everyday discourse the terms Artificial Intelligence, Machine Learning, and Deep Learning are often used interchangeably, yet in computer science they describe strictly nested disciplines with distinct mathematical assumptions:

- **Artificial Intelligence (AI):** The overarching discipline dedicated to designing agents that perform tasks associated with human cognition (symbolic reasoning, perception, natural language understanding, and sequential planning).
- **Machine Learning (ML):** A subset of AI in which parameters are estimated statistically from data rather than programmed via explicit domain rules. Classical ML (such as Support Vector Machines or Decision Trees) relies heavily on **hand-crafted features** designed manually by domain experts (for example, SIFT visual keypoints, MFCC acoustic coefficients, or TF-IDF word frequencies).
- **Deep Learning (DL):** A specialization of ML built on multi-layer neural networks that replaces manual feature engineering with **end-to-end representation learning**. Lower layers extract elementary motifs (edges, phonemes, subwords), while deeper layers compose them into high-level semantic abstractions directly from raw pixels, waveforms, or token sequences.
- **Generative AI:** Deep learning architectures (autoregressive LLMs, Diffusion models, and multimodal Transformers) that learn the underlying data distribution $P(x)$ or conditional distribution $P(y \mid x)$ to synthesize original text, source code, scientific hypotheses, imagery, and music.

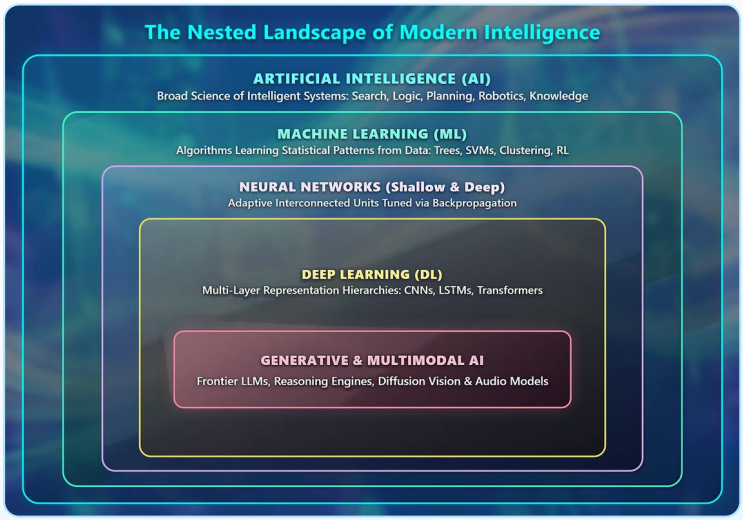


Figure 1.2

The nested architectural relationship among Artificial Intelligence, Machine Learning, Deep Learning, and Generative AI.

1.3 Unified Mathematical Framework: Empirical Risk Minimization & Inductive Bias

Before reading the formal equations below, translate the symbols into everyday engineering terms: x is the input information we observe, y is the true answer, $\hat{y} = f_\theta(x)$ is our model's best guess controlled by all its tunable knobs θ, and the loss $L(\hat{y}, y)$ is a numerical scorekeeper measuring how wrong a guess is (zero means a perfect prediction). Training simply asks: *which knob settings θ^* achieve the lowest average error across our examples without becoming overly complicated?*

Beneath the surface diversity of modern neural networks (from compact 2-input logic gates to trillion-parameter foundation models) lies this single unified mathematical formulation. Given an input space X and target space Y governed by an unknown joint probability distribution $P(x, y)$, we choose a parameterized **hypothesis class** $H = \{ f_\theta : X \to Y \mid \theta \in \Theta \}$ where every candidate model produces predictions $\hat{y} = f_\theta(x)$.

Ideally, we would minimize the **True Risk** (expected loss across all future unseen examples): $R(\theta) = E_{(x, y) \sim P}[L(f_\theta(x), y)]$. Because the true distribution P is inaccessible, we observe a finite training sample $D = \{ (x_i, y_i) \}_{i=1..N}$ and minimize the **Regularized Empirical Risk** instead:

$$\theta^* = \operatorname{argmin}_\theta \left[(1 / N) \sum_{i=1..N} L(f_\theta(x_i), y_i) + \lambda \cdot \Omega(\theta) \right]$$

The first term $R_{emp}(\theta)$ measures data fidelity over the N training samples (MSE, BCE, or Cross-Entropy), while $\lambda \cdot \Omega(\theta)$ penalizes model complexity (L2 weight decay, Dropout) to shrink the generalization gap $R(\theta) - R_{emp}(\theta)$.

STEP-BY-STEP WORKED EXAMPLE **Predicting Apartment Rent: Rule-Based Code vs. ML vs. Deep Learning**

Suppose we want to predict monthly apartment rent y (in EUR) from floor area x (in m^2) using three historical listings: **(40 m^2, 360 EUR)**, **(60 m^2, 480 EUR)**, and **(80 m^2, 600 EUR)**. Hard-coded software brackets (`if x < 50: return 350`) fail smoothly between tiers. Machine Learning instead proposes a simple linear model $\hat{y} = w \cdot x + b$, where w is the price per m^2 and b is a base cost:

1. **Initial untrained guess (w = 4, b = 100):** Predictions are $\hat{y}(40) = 4 \times 40 + 100 = 260$ (error −100), $\hat{y}(60) = 4 \times 60 + 100 = 340$ (error −140), and $\hat{y}(80) = 4 \times 80 + 100 = 420$ (error −180).
2. **Quantifying mismatch via Mean Squared Error (MSE):** Squaring each error prevents positive and negative mistakes from cancelling: $MSE = [(-100)^2 + (-140)^2 + (-180)^2] / 3 = (10{,}000 + 19{,}600 + 32{,}400) / 3 \approx 20{,}666.7$ (typical mismatch $\sqrt{MSE} \approx 143.8$ EUR).
3. **Automated training & generalization:** Because every guess is too low, optimization increases w and b automatically until it converges to **$w^* = 6$, $b^* = 120$** (driving $MSE = 0.0$ on all three listings). On an unseen **70 m^2** apartment, the learned model immediately predicts $\hat{y} = 6 \times 70 + 120 = 540$ EUR.
4. **Leap to Deep Learning:** Whereas this linear ML model uses one hand-chosen feature (x = area), a Deep Learning network can process raw apartment photos and text descriptions directly to learn lighting, renovation quality, and neighborhood appeal automatically.

Why does minimizing regularized loss on finite data generalize to unseen test inputs? The **No Free Lunch Theorem** proves that without structural assumptions, no algorithm outperforms random guessing when averaged over all possible target functions. Successful models embed an explicit **inductive bias**: a set of geometric or probabilistic priors built directly into the hypothesis class H. Convolutional networks encode spatial translation equivariance for imagery; LSTMs encode causal sequential state for time series; Transformers encode global pairwise context across token sets. Every laboratory in this monograph instantiates these exact five pillars: structured data tensors, a parameterized hypothesis class, a differentiable loss, a gradient optimizer, and strict out-of-sample generalization evaluation.

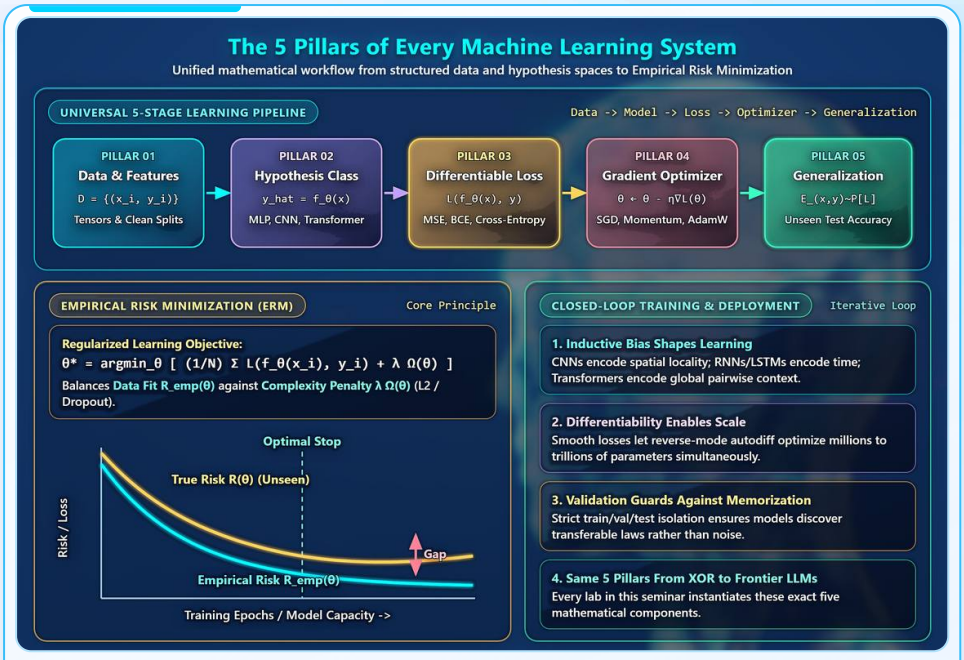


Figure 1.3

The 5 Pillars of Every Machine Learning System: unified mathematical pipeline, Regularized Empirical Risk Minimization (ERM), True vs. Empirical Risk gap, and inductive bias.

1.4 The Core Paradigms of Machine Learning

The structure of the training dataset and the nature of the supervisory signal determine which of the four core machine learning paradigms an engineer selects:

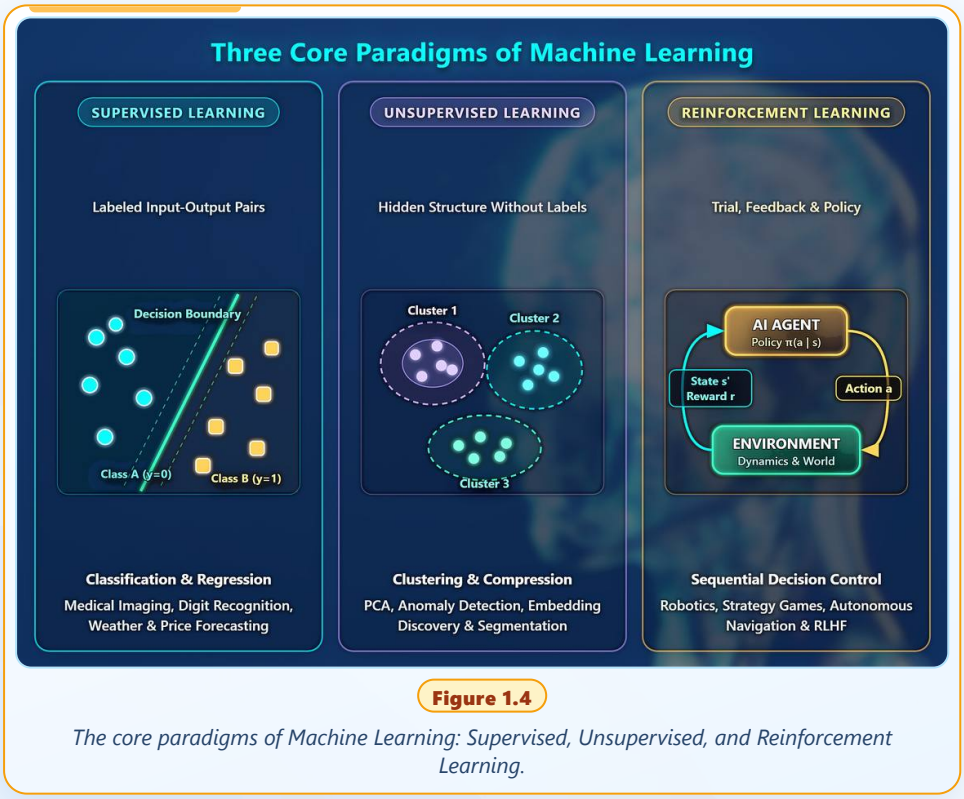


Figure 1.4

The core paradigms of Machine Learning: Supervised, Unsupervised, and Reinforcement Learning.

Learning Paradigm	Input & Feedback Signal	Mathematical Objective	Representative Applications
Supervised	Labeled pairs (x_i, y_i)	Minimize expected loss $L(f_\theta(x), y)$ for classification or regression	MNIST/CIFAR-10 vision, medical diagnosis, Patras weather forecasting

Learning Paradigm	Input & Feedback Signal	Mathematical Objective	Representative Applications
Unsupervised	Unlabeled samples x_i	Discover latent clusters, low-dimensional manifolds, or density $P(x)$	Anomaly detection, PCA dimensionality reduction, k-Means clustering
Self-Supervised	Raw corpora / spectra (auto-derived targets)	Predict masked spans or next tokens $P(x_t \mid x_{<t})$	Pre-training frontier LLMs (Gemma, Llama, GPT) and vision encoders
Reinforcement (RL)	States s, actions a, scalar rewards r	Maximize discounted cumulative return $E[\Sigma\ \gamma^t\ r_t]$	Tic-Tac-Toe DQN, AlphaGo, robotic control, LLM alignment

◆ Key Historical Milestones in the Science of AI

1956 (Dartmouth Conference): Coining of the term Artificial Intelligence • **1986:** Popularization of Error Backpropagation for multi-layer networks • **2012 (AlexNet):** Deep CNN breakthrough powered by CUDA GPU acceleration • **2017 (Transformers):** Parallel Self-Attention replaces sequential recurrence • **Modern Era:** Test-time reasoning models, sparse MoE architectures, open-weight ecosystems, and tool-using scientific AI agents.

Across every chapter that follows, we bridge these formal mathematical principles directly with verified Python 3.12+ code and reproducible laboratory experiments.

CHAPTER 2

Biological and Artificial Neurons, Activation Functions

From synaptic integration and linear hyperplane geometry to exact activation derivatives, vanishing gradients, and Lab 01.

Learning Objectives & Theoretical Scope

- Map biological neurophysiology (dendritic integration, soma threshold, axonal firing) onto the mathematical artificial neuron $y = f(w^T x + b)$.
- Interpret the equation $w^T x + b = 0$ geometrically as a separating hyperplane whose normal vector w sets orientation and whose bias b sets offset.
- Derive exact analytical derivatives for Sigmoid, Tanh, ReLU, Leaky ReLU, GELU, and Softmax, diagnosing **Vanishing Gradients** and **Dying ReLUs**.
- Run controlled parameter sweeps in **Lab 01** to observe saturation and smooth nonlinear transitions in real time.

2.1 Biological Synaptic Integration & Linear Hyperplane Geometry

The human brain contains roughly 86 billion neurons linked by hundreds of trillions of synapses. Each biological neuron receives excitatory and inhibitory postsynaptic potentials through its *dendrites*, performs spatial and temporal summation across the *cell body (soma)*, and whenever membrane voltage crosses the firing threshold at the *axon hillock*, propagates an all-or-none electrochemical action potential down its *axon*. Learning corresponds biologically to modifying synaptic efficacy (long-term potentiation and depression).

Inspired by this mechanism, the artificial neuron (evolving from the 1943 McCulloch-Pitts threshold logic unit and Frank Rosenblatt's 1958 Perceptron to modern differentiable units) decomposes computation into two canonical operations: an **affine inner product** with synaptic weights $w \in R^n$ and bias $b \in R$, followed by a scalar **nonlinear activation function** $f(\cdot)$.

FIRST PRINCIPLES & INTUITION **An Artificial Neuron as a Weighted Jury Vote with a Dimmer Switch**

Imagine deciding whether to pack an umbrella before leaving for campus. You weigh two everyday clues: how dark the clouds look (x_1) and how humid the air feels (x_2). Each clue receives a **weight** (w_1, w_2) measuring its importance: a positive weight votes "yes," a negative weight votes "no," and a weight near zero ignores that clue. You also add a personal **bias** (b) reflecting your baseline habit (for example, a cautious student carries an umbrella even on mildly overcast days).

Combining those weighted votes yields one raw score called the **pre-activation** or **logit** (z). Passing z through an **activation function** $f(z)$ acts like a smooth dimmer switch or valve: instead of returning unbounded raw scores, it converts z into a clean output (such as a firing strength or probability between 0 and 1) that lets multi-layer networks bend decision boundaries around curved, real-world patterns.

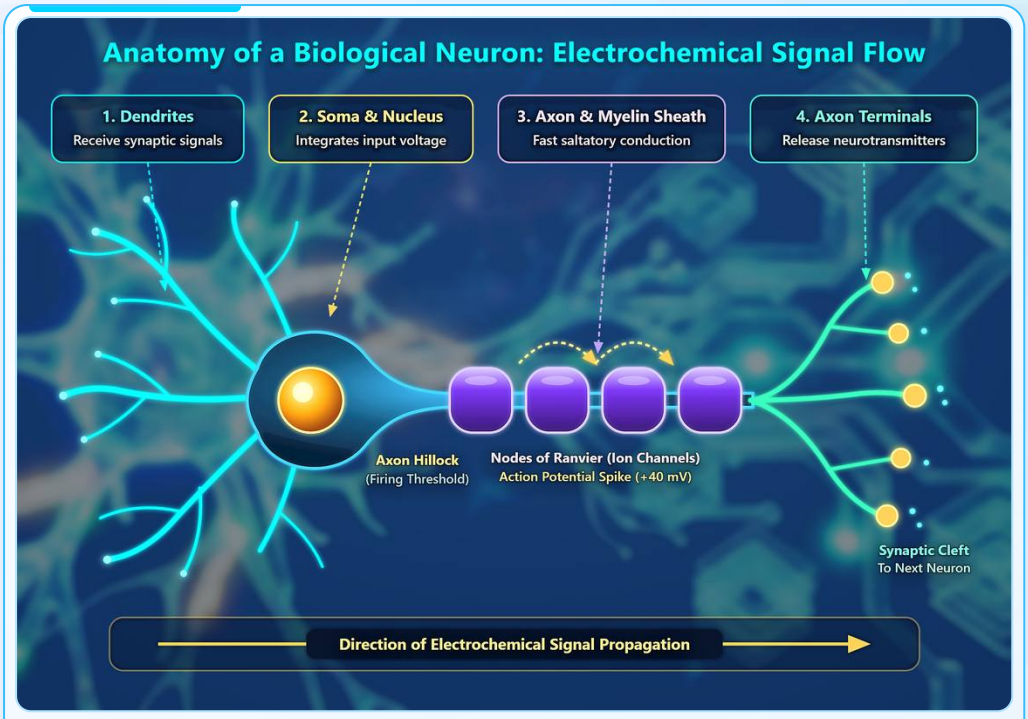


Figure 2.1a *Anatomy of a biological neuron (dendrites, soma, axon, myelin sheath, and synapses).*

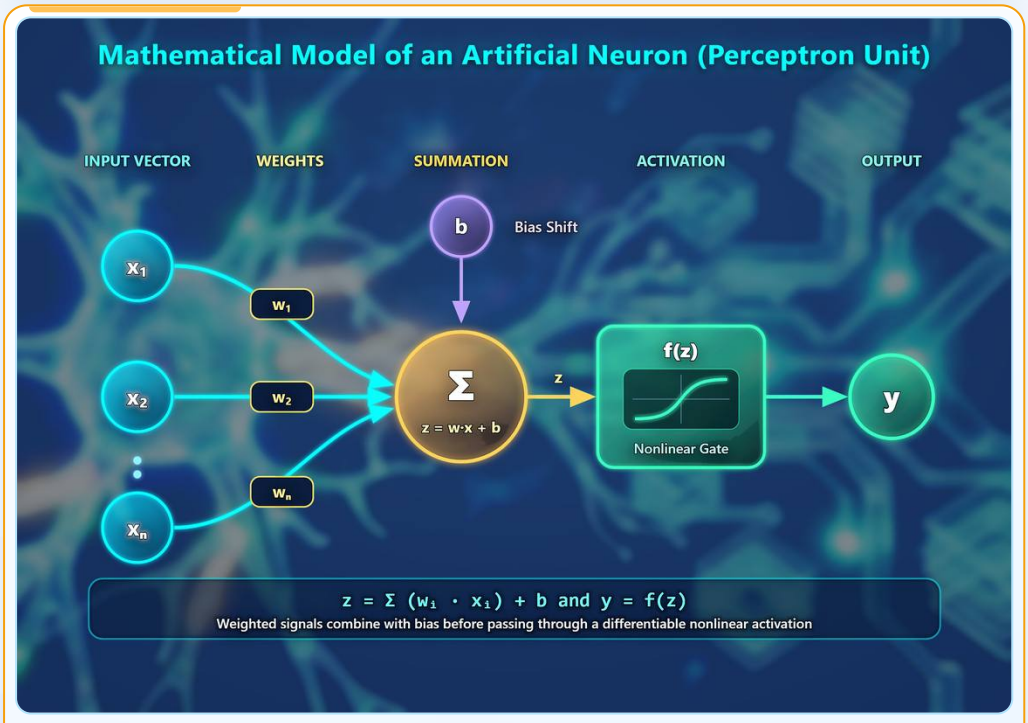


Figure 2.1b

Mathematical diagram of an artificial neuron (Perceptron) with weights w, bias b, and activation f(z).

$$z = w^T x + b = \Sigma_{i=1..n} (w_i \cdot x_i) + b \quad \Rightarrow \quad y = f(z)$$

Where $x \in R^n$ is the input vector, $w \in R^n$ holds the trainable synaptic weights, b is the bias offset, z is the pre-activation logit, and $f(\cdot)$ is a differentiable nonlinearity.

Geometric Interpretation of the Decision Hyperplane: Setting the pre-activation equal to zero, $w^T x + b = 0$, defines an $(n - 1)$-dimensional **hyperplane** slicing R^n into a positive half-space ($z > 0$) and a negative half-space ($z < 0$). In 2D this boundary is a straight line; in 3D it is a flat plane. The weight vector w is strictly orthogonal (normal) to the hyper-

plane, controlling both its angular orientation and the steepness of transition via its Euclidean norm $\|w\|_2$. Meanwhile, the signed perpendicular distance from the origin to the boundary equals $-b / \|w\|_2$: adjusting the bias b translates the decision boundary across space without rotating it, acting identically to a shiftable biological firing threshold.

2.2 Nonlinear Activation Functions, Exact Derivatives & Gradient Pathologies

Without a nonlinear activation function between layers, composing any depth L of affine transformations collapses algebraically into a single linear operator: $W_2(W_1x + b_1) + b_2 = W_{eq}x + b_{eq}$. Depth adds expressive power only when interleaved with nonlinear activations. For readers new to calculus, the **derivative** $f'(z)$ listed in the table below simply measures the **slope** of the activation curve at z: during learning, that slope controls how strongly a small tweak to a weight changes the neuron's output. Whenever a curve flattens out ($slope \approx 0$), error feedback can no longer travel backward through that neuron.

Activation	Formula $f(z)$	Exact Derivative $f'(z)$	Range	Gradient Behavior & Primary Role
Sigmoid	$1 / (1 + e^{-z})$	$\sigma(z) \cdot (1 - \sigma(z)) \leq 0.25$	(0, 1)	Saturates for $\|z\| > 4$ causing severe **vanishing gradients**; binary output probabilities & LSTM gates
Tanh	$(e^{z} - e^{-z}) / (e^{z} + e^{-z})$	$1 - \tanh^2(z) \in (0, 1]$	(−1, +1)	Zero-centered (4× stronger peak slope than Sigmoid), yet still saturates at extremes; RNN/LSTM states
ReLU	$\max(0, z)$	1 if $z > 0$, else 0	[0, +∞)	Unit positive slope eliminates positive vanishing gradients, but negative bias shifts cause **Dying ReLUs**
Leaky ReLU	$\max(\alpha z, z)$, $\alpha = 0.01$	1 if $z > 0$, else α	(−∞, +∞)	Preserves non-zero slope α on the negative axis so inactive units can recover gradient flow
GELU	$0.5 \cdot z \cdot (1 + \text{erf}(z / \sqrt{2}))$	$\Phi(z) + z \cdot \varphi(z)$	(−0.17, +∞)	Smooth, non-monotonic curvature near zero (Φ, φ are normal CDF/PDF); standard in Transformers & LLMs
Softmax	$e^{z_i} / \sum_j e^{z_j}$	$\partial p_i/\partial z_j = p_i(\delta_{ij} - p_j)$	(0, 1), Σ=1	Couples all logits into a valid probability simplex for multi-class cross-entropy classification

Two failure modes dominate activation selection in deep architectures. First, the **Vanishing Gradient Problem** occurs in deep Sigmoid or Tanh networks because backpropagation multiplies local derivatives along every layer: since $\sigma'(z) \leq 0.25$ everywhere, propagating across just 10 layers attenuates the gradient by at least $0.25^{10} \approx 9.5 \times 10^{-7}$, freezing early feature detectors. Second, the **Dying ReLU Problem** arises when an aggressive learning rate pushes a neuron's bias deeply negative so $z < 0$ for every training example; because $ReLU'(z) = 0$, that unit receives zero gradient forever. Leaky ReLU and smooth probabilistic activations like **GELU** resolve both pathologies simultaneously.

STEP-BY-STEP WORKED EXAMPLE **Computing a 2-Input Neuron by Hand (Active Logit vs. Deep Saturation)**

Consider a 2-input neuron with inputs $x_1 = 1.0$, $x_2 = 0.5$, weights $w_1 = 0.8$, $w_2 = -0.4$, and bias $b = 0.2$:

1. **Step 1 (Pre-activation logit):** $z = w_1x_1 + w_2x_2 + b = (0.8 \times 1.0) + (-0.4 \times 0.5) + 0.2 = 0.80 - 0.20 + 0.20 =$ ***0.80***.
2. **Step 2 (Compare activations at z = +0.80): ReLU(0.80) = 0.800** (slope *1.000*, unattenuated gradient); **Sigmoid(0.80) = 1 / (1 + $e^{-0.80}$) = 0.690** (69.0% probability, slope $0.690 \times 0.310 = 0.214$); **Tanh(0.80) = ($e^{0.80}$ − $e^{-0.80}$) / ($e^{0.80}$ + $e^{-0.80}$) = 0.664** (slope $1 - 0.664^2 = 0.559$).
3. **Step 3 (What happens if z = −5.0?): ReLU(−5.0) = 0.000** with slope *0.000* (shut off completely; if this holds for all inputs, the unit becomes a **Dying ReLU**, whereas *Leaky ReLU(−5.0) = −0.050* keeps slope *0.01*). **Sigmoid(−5.0) = 0.0067** with tiny slope *0.0066* (99.3% of the learning signal vanishes), and **Tanh(−5.0) = −0.9999** with slope *0.00018* (deep saturation).

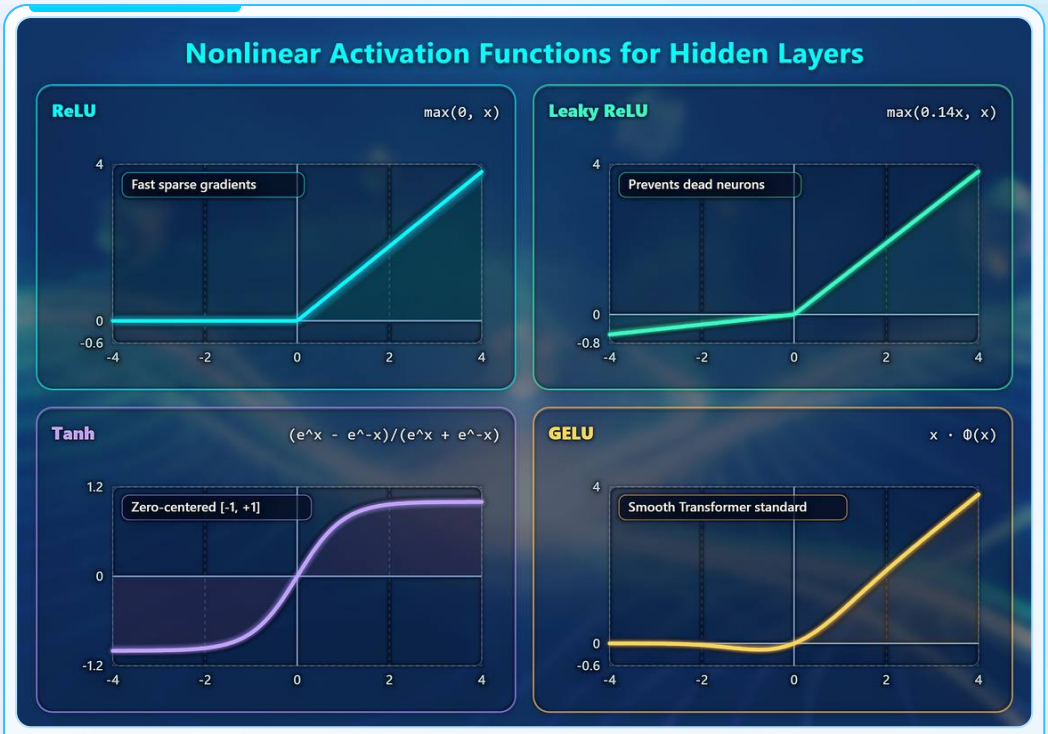


Figure 2.2a *Primary nonlinear hidden-layer activation functions: ReLU, Leaky ReLU, Tanh, and GELU.*

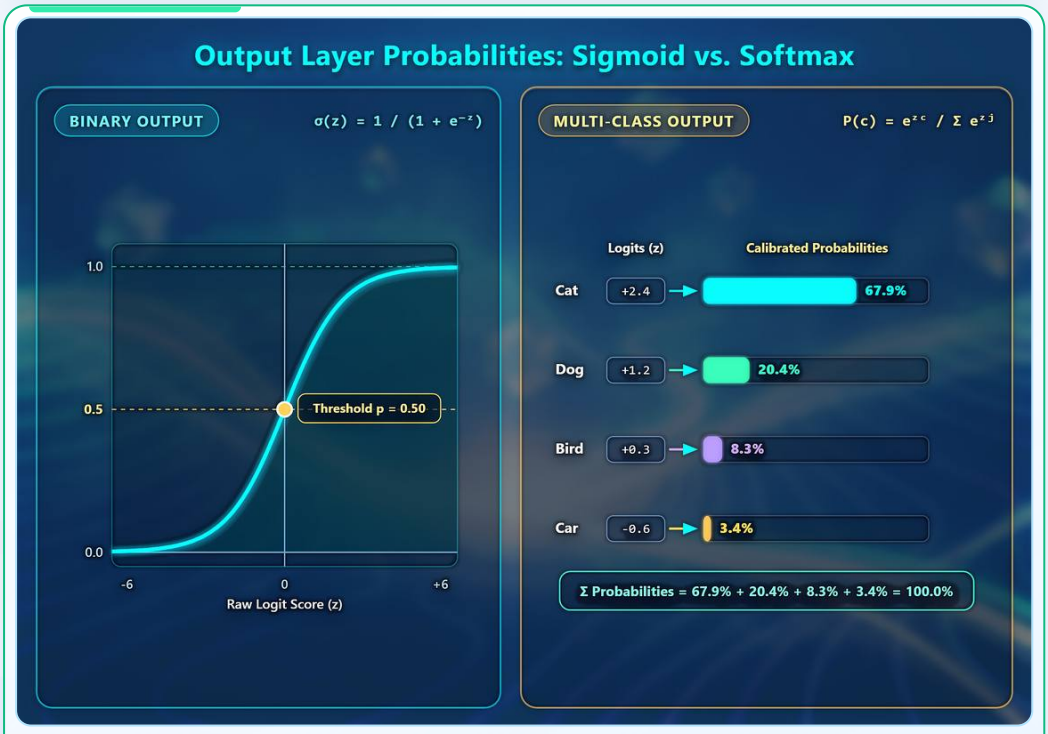


Figure 2.2b

Output-layer activation functions: Sigmoid (binary probability) and Softmax (multi-class distribution).

2.3 Lab 01: Interactive Neuron Simulator in Python

LAB 01 **Interactive Artificial Neuron & Activation Explorer** `code/01_basic_neuron_gui.py`

Implements the exact mathematical equation $z = w_1x_1 + w_2x_2 + b$ and $y = f(z)$ inside a desktop Tkinter simulator. Includes a numerically overflow-safe Sigmoid branch alongside ReLU, Tanh, and exact Gaussian Error Linear Unit (GELU) via `math.erf`, exposing how pre-activation magnitude governs output saturation.

Guided Experiment: Set inputs and weights to produce extreme logits (`z = +10` and `z = -10`) and switch across all four activations: verify that Sigmoid and Tanh flatten into zero-gradient saturation, ReLU strictly clips the negative half-axis to 0.0, and GELU exhibits a smooth, slightly negative dip around `z = -0.75` before smoothly decaying toward zero.

PYTHON 3.12+ `code/01_basic_neuron_gui.py` — Numerically stable activation core & Tkinter GUI

```python
import math
import tkinter as tk
from tkinter import messagebox, ttk


def activate(z: float, func_name: str) -> float:
    """Applies the selected nonlinear activation function to z."""
    if func_name == "Sigmoid":
        # Numerically stable sigmoid implementation: 1 / (1 + e^(-z))
        if z >= 0:
            return 1.0 / (1.0 + math.exp(-z))
        exp_z = math.exp(z)
        return exp_z / (1.0 + exp_z)

    if func_name == "ReLU":
        # Rectified Linear Unit: zeroes negative values, keeps positive ones
        return max(0.0, z)

    if func_name == "Tanh":
        # Hyperbolic tangent with zero-centered output range (-1, 1)
        return math.tanh(z)

    if func_name == "GELU":
        # Gaussian Error Linear Unit: smooth activation used in Transformers
        return 0.5 * z * (1.0 + math.erf(z / math.sqrt(2.0)))

    raise ValueError(f"Unknown activation function: {func_name}")


class NeuronSimulatorApp:
    """Graphical simulator application for a single artificial neuron."""

    def __init__(self, root: tk.Tk) -> None:
        self.root = root
        self.root.title("Artificial Neuron Simulator | AI, ML & DL Seminar")
        self.root.geometry("540x430")
        self.root.resizable(False, False)

        # Main padded container frame for clean visual layout
        frame = ttk.Frame(root, padding=18)
        frame.pack(fill=tk.BOTH, expand=True)
# ... (full executable code in code/01_basic_neuron_gui.py inside code.zip)
```

Practical Engineering Rule for Activation Selection

Use **ReLU** or **Leaky ReLU** for fast convolutional vision backbones, **GELU** (or SwiGLU) inside Transformer feed-forward blocks and modern MLPs, **Sigmoid** strictly at binary classification outputs or recurrent control gates, and **Softmax** at mutually exclusive multi-class output layers (or raw logits with `from_logits=True` for maximal numerical precision).

CHAPTER 3

Logic Gates, Mathematical Optimization & First Models

How neural networks learn from scratch: AND separability, Backpropagation chain rule, AdamW, Keras 3 APIs, Universal Approximation for XOR, and Labs 02 to 08.

Learning Objectives & Theoretical Scope

- Contrast linearly separable problems (AND gate) with non-linearly separable problems (XOR gate) using explicit hyperplane geometry and the **Universal Approximation** intuition.
- Derive every partial derivative of **reverse-mode Backpropagation** step-by-step via the multivariable chain rule and verify why it computes all parameter gradients in a single backward sweep.
- Write and apply the full bias-corrected moment updates and decoupled weight decay equations of **Adam** and **AdamW**.
- Build, train, compare, serialize, and reload neural networks across pure NumPy and Keras 3 (Sequential vs. Functional API with Residual Skip Connections) in **Labs 02, 03, 05, 06, 07, and 08**.

3.1 The AND Logic Gate as a Linearly Separable Problem

The **AND** logic gate provides the cleanest possible laboratory benchmark for binary classification: four input pairs `(0,0), (0,1), (1,0), (1,1)` with target `1` exclusively at `(1,1)`. Geometrically, a single linear hyperplane such as $x_1 + x_2 - 1.5 = 0$ cleanly separates class 1 from class 0 with positive margin.

FIRST PRINCIPLES & INTUITION **Descending a Foggy Mountain Valley: Gradient Descent & Backpropagation**

How does a neural network actually discover good weights without a human typing them in? Imagine standing blindfolded on a foggy mountain hillside (the **loss landscape**), where your height represents the network's prediction error and your map position represents its **weights**. You cannot see the entire valley, so you feel the slope right under your boots (the **gradient**): if the ground rises to the right, you step downhill to the left! The length of each stride is the **learning rate (η)**: tiny steps crawl too slowly, while oversized leaps overshoot the valley floor entirely.

- **Backpropagation:** Works like tracing responsibility backward along a relay team, using calculus's chain rule to compute every weight's exact share of the final error in one sweep.
- **Epoch:** One complete pass through the entire training dataset (for the AND gate, all 4 input pairs).
- **Linearly Separable:** A problem whose two classes can be cleanly separated by a single straight line (like AND), in contrast to diagonal puzzles like XOR that require hidden layers.

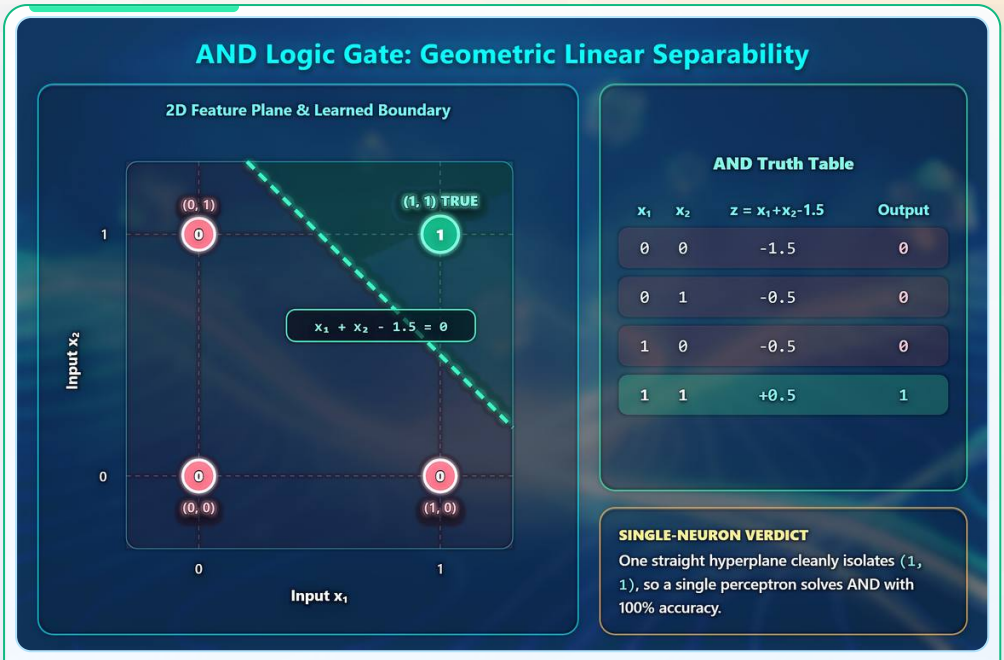


Figure 3.1

Truth table, geometric decision boundary, and neural implementation of the AND logic gate.

LAB 02 **Pure-NumPy Two-Layer Network, Backpropagation & HDF5 Export** `code/02_and_gate_numpy.py`

Trains a 2 → 3 → 1 neural network purely in NumPy with Xavier/Glorot weight initialization, explicit matrix forward pass, manual chain-rule backpropagation, and HDF5 (`.h5`) serialization so every learned weight matrix can be inspected directly inside VS Code via H5Web.

Guided Experiment: Run the script with `seed=42` and verify RMSE drops from ~0.49 to 0.000640 across 10,000 epochs; then test changing `learning_rate` from 0.5 to 0.01 and 5.0 to observe slow convergence versus oscillation.

PYTHON 3.12+ `code/02_and_gate_numpy.py` Pure-NumPy initialization, forward pass & HDF5 export

```python
from pathlib import Path
import numpy as np

try:
    import h5py
except ImportError:
    h5py = None


def sigmoid(z: np.ndarray) -> np.ndarray:
    """Sigmoid activation function: maps every real value into (0, 1)."""
    return 1.0 / (1.0 + np.exp(-np.clip(z, -500.0, 500.0)))


def sigmoid_derivative_from_output(a: np.ndarray) -> np.ndarray:
    """Derivative of sigmoid expressed directly from its output a."""
    return a * (1.0 - a)


def main() -> None:
    # Fixed random seed for 100% reproducible training results
    rng = np.random.default_rng(seed=42)

    # Truth table for the AND logic gate: inputs X (4x2) and targets y (4x1)
    X = np.array(
        [
            [0.0, 0.0],
            [0.0, 1.0],
            [1.0, 0.0],
```

```
            [1.0, 1.0],
        ],
        dtype=np.float64,
    )
    y = np.array([[0.0], [0.0], [0.0], [1.0]], dtype=np.float64)

    # Xavier/Glorot weight initialization and zero initialization for biases
    input_dim, hidden_dim, output_dim = 2, 3, 1
    W1 = rng.normal(
        0.0, np.sqrt(2.0 / (input_dim + hidden_dim)), size=(input_dim, hidden_dim)
    )
    b1 = np.zeros((1, hidden_dim), dtype=np.float64)
    W2 = rng.normal(
        0.0, np.sqrt(2.0 / (hidden_dim + output_dim)), size=(hidden_dim, output_dim)
    )
# ... (full executable code in code/02_and_gate_numpy.py inside code.zip)
```

VERIFIED OUTPUT

```
Epoch 1 | RMSE: 0.491903   ->   Epoch 10000 | RMSE: 0.000640
(0,0)->0.000003 [0]  |  (0,1)->0.000461 [0]  |  (1,0)->0.000421 [0]  |  (1,1)->0.998882 [1]
```

3.2 Step-by-Step Backpropagation Chain Rule & AdamW Optimization

Why do we measure errors with a smooth formula instead of simply counting right and wrong answers? Because a raw right-or-wrong step count forms a flat staircase with zero slope almost everywhere, leaving gradient descent blind to which direction improves the model. Training any neural network is therefore formulated as continuous **mathematical optimization**: minimizing a smooth, differentiable loss $L(\theta)$ over parameters θ. For binary probabilities $p_i = \sigma(z_i)$, maximizing the Bernoulli log-likelihood yields **Binary Cross-Entropy (BCE)**, which imposes an unbounded logarithmic penalty on overconfident wrong predictions:

$$\mathbf{L_{BCE} = -(1/N)\,\Sigma_{i=1..N}\,[\,y_i \cdot \ln(p_i) + (1 - y_i) \cdot \ln(1 - p_i)\,]}$$

When target y = 1 and predicted probability p → 0, the penalty −ln(p) → +∞, creating a strong restorative gradient.

Step-by-Step Backpropagation Derivation: Consider a single sigmoid neuron processing input x with target $y \in \{0, 1\}$: forward pass $z = w^T x + b$, activation $a = \sigma(z)$, and scalar loss $L = -[y \ln(a) + (1 - y) \ln(1 - a)]$. Applying the multivariable **chain rule** in reverse order from loss back to each weight produces:

1. **Loss gradient w.r.t. activation:** $\partial L / \partial a = -(y / a) + (1 - y) / (1 - a) = (a - y) / [a(1 - a)]$.
2. **Local Sigmoid derivative:** $\partial a / \partial z = a(1 - a)$.
3. **Exact cancellation miracle:** Multiplying factors (1) and (2) cancels the denominator identically, yielding the ultra-clean upstream error signal $\delta = \partial L / \partial z = a - y$!
4. **Weight and bias gradients:** Since $\partial z / \partial w_i = x_i$ and $\partial z / \partial b = 1$, we obtain $\partial L / \partial w_i = (a - y) \cdot x_i$ and $\partial L / \partial b = a - y$.

In the worked numerical graph of Figure 3.2 (x_1=1.0, w_1=0.80, x_2=0.5, w_2=−0.40, b=0.20, y=1.0), the forward pass computes $z = 0.80$, $a = \sigma(0.80) = 0.690$, and $L = 0.371$. In reverse mode, $\partial L/\partial a = -1.449$ multiplies $\partial a/\partial z = 0.214$ to give $\partial L/\partial z = -0.310$, immediately supplying $\partial L/\partial w_1 = -0.310$, $\partial L/\partial w_2 = -0.155$, and $\partial L/\partial b = -0.310$. Because neural networks map millions or billions of parameters N to a single scalar loss, **reverse-mode automatic differentiation** computes all N gradients in **O(1) sweep** (1 forward + 1 backward pass), whereas finite differences would require an intractable $N + 1$ forward passes!

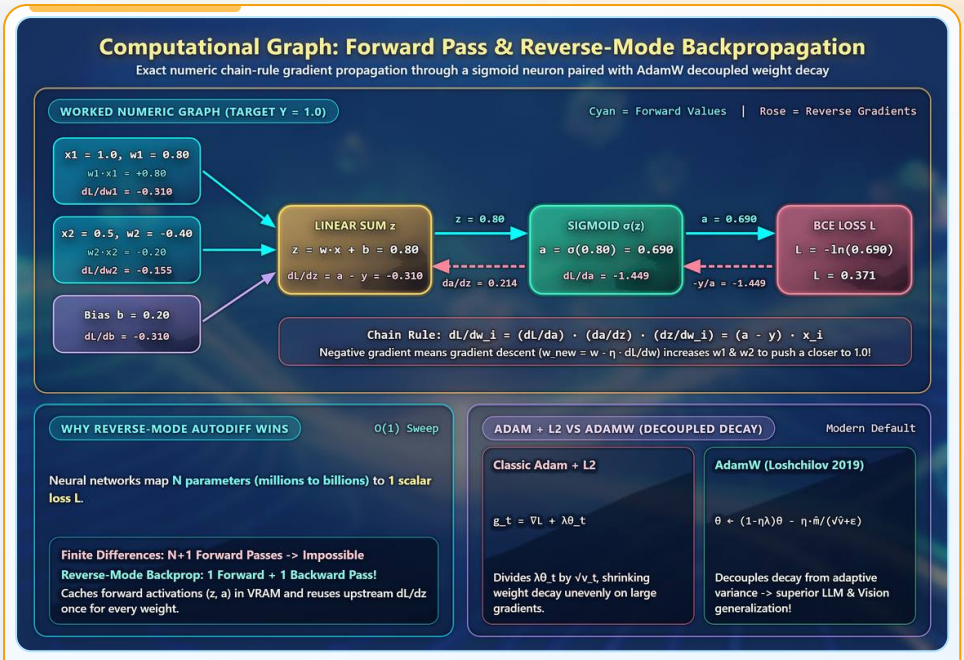


Figure 3.2

Worked numeric computational graph for Forward Pass and Reverse-Mode Backpropagation, paired with O(1) autodiff efficiency and Adam + L2 vs. AdamW decoupled weight decay.

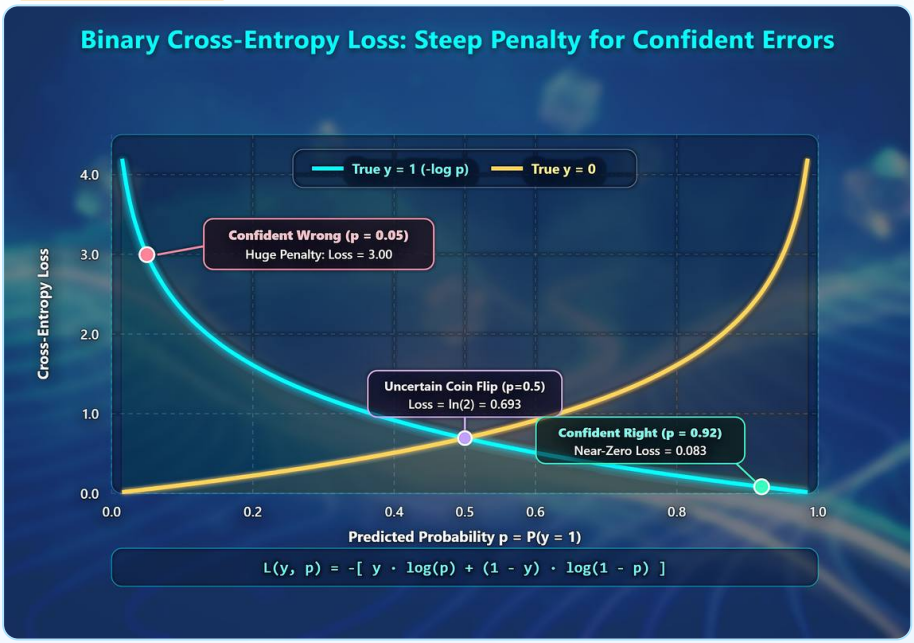


Figure 3.3a

Behavior of the Binary Cross-Entropy (BCE) loss function for incorrect and correct predictions.

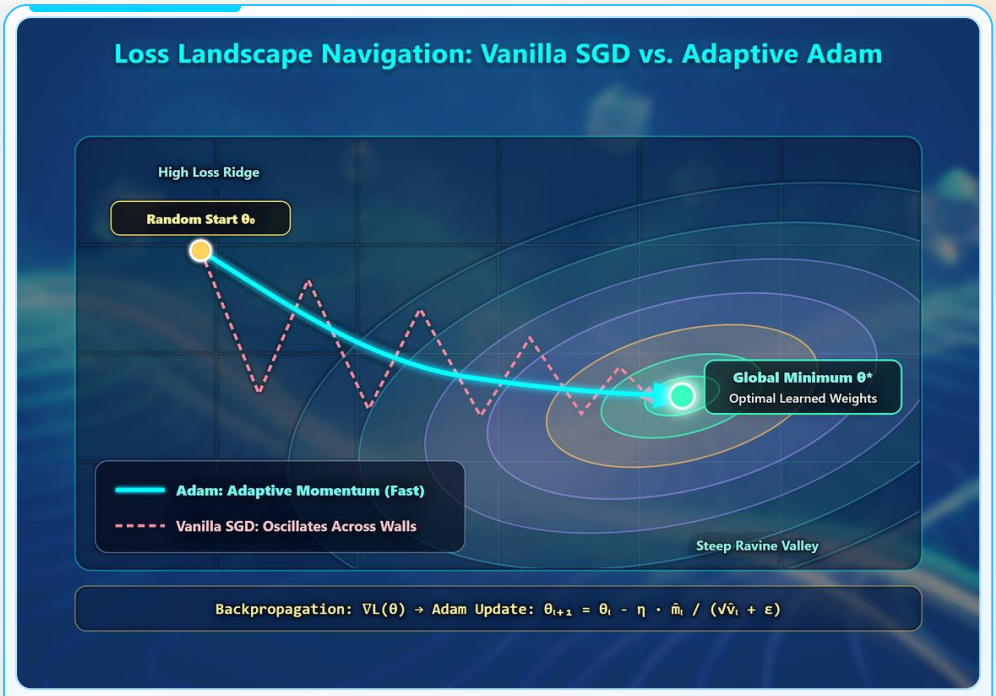


Figure 3.3b

Loss landscape, vanilla stochastic gradient descent (SGD), and adaptive momentum trajectories of the Adam optimizer.

From Vanilla SGD to Adam and AdamW (Decoupled Weight Decay): Standard Gradient Descent ($\theta_{t+1} = \theta_t - \eta\, g_t$, where $g_t = \nabla_\theta L(\theta_t)$) oscillates across steep ravines and stalls on flat saddle ridges. **Adam** (Kingma & Ba, 2015) tracks exponentially weighted moving averages of both the gradient mean (first moment m_t, acting as physical momentum) and the uncentered gradient variance (second moment v_t, normalizing step sizes per parameter). Because $m_0 = v_0 = 0$ biases early estimates toward zero, Adam applies exact bias corrections $\hat{m}_t$ and $\hat{v}_t$. Finally, **AdamW** (Loshchilov & Hutter, 2019) fixes a subtle flaw in classic Adam+L2: rather than adding $\lambda\theta_t$ into g_t (where it gets rescaled unevenly by $\sqrt{\hat{v}_t}$), AdamW applies **decoupled weight decay** directly to the parameters:

$$\mathbf{m_t = \beta_1\, m_{t-1} + (1-\beta_1)\, g_t} \quad | \quad \mathbf{v_t = \beta_2\, v_{t-1} + (1-\beta_2)\, g_t^2}$$

$$\mathbf{\hat{m}_t = m_t \,/\, (1-\beta_1^{\,t})} \quad | \quad \mathbf{\hat{v}_t = v_t \,/\, (1-\beta_2^{\,t})}$$

$$\mathbf{\theta_{t+1} = (1-\eta\lambda)\, \theta_t - \eta \cdot \hat{m}_t \,/\, (\sqrt{\hat{v}_t} + \varepsilon)}$$

Standard hyperparameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$, weight decay $\lambda \in [10^{-4}, 10^{-1}]$. Decoupled decay $(1 - \eta\lambda)\theta_t$ delivers markedly superior generalization in Vision and LLM training.

STEP-BY-STEP WORKED EXAMPLE **Gradient Descent by Hand on L(w) = (w - 3)^2: Tuning the Learning Rate**

Consider a simple bowl-shaped loss $\mathbf{L(w) = (w-3)^2}$ whose exact minimum is $w^* = 3$ (where $L(3) = 0$). Its derivative (slope) is $\mathbf{g = dL/dw = 2(w-3)}$. Starting at $\mathbf{w_0 = 0}$ (error $L = 9.00$), apply the update rule $w_{t+1} = w_t - \eta \cdot g_t$ across three choices of learning rate η:

1. **Well-tuned (η = 0.1):** Step 1: $g_0 = 2(0-3) = -6.0 \rightarrow w_1 = 0 - 0.1(-6.0) = \mathbf{0.60}$ ($L = 5.76$). Step 2: $g_1 = 2(0.60-3) = -4.8 \rightarrow w_2 = 0.60 - 0.1(-4.8) = \mathbf{1.08}$ ($L = 3.69$). Step 3: $g_2 = -3.84 \rightarrow w_3 = \mathbf{1.464}$ ($L = 2.36$). Each step glides cleanly 20% closer to $w^* = 3$.
2. **Too small (η = 0.001):** Step 1 moves only to $w_1 = 0 - 0.001(-6.0) = \mathbf{0.006}$ ($L = 8.964$), requiring thousands of iterations just to cross the valley.
3. **Too large (η = 1.2, diverges!):** Step 1 leaps past the minimum to $w_1 = 0 - 1.2(-6.0) = \mathbf{7.20}$ ($L = 17.64$, worse than where we started!). Step 2 rebounds even harder: $g_1 = +8.40 \rightarrow w_2 = 7.20 - 1.2(8.40) = \mathbf{-2.88}$ ($L = 34.57$), exploding rapidly to numerical overflow (`NaN`).

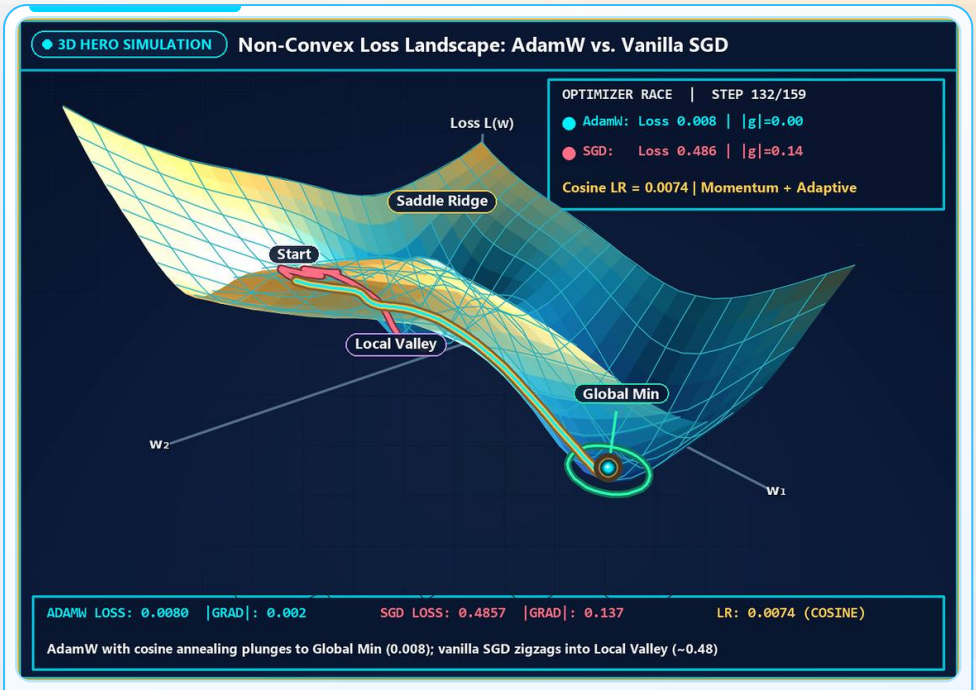


Figure 3.4

3D non-convex loss landscape simulation: AdamW with cosine learning-rate annealing smoothly escapes saddle ridges to reach the Global Min (Loss 0.0080), while vanilla SGD zigzags into a suboptimal Local Valley (Loss 0.4857).

3.3 Keras 3: Sequential API versus Functional API (Labs 03 & 05)

Keras 3 operates natively across TensorFlow, PyTorch, and JAX backends with one clean Python interface. Simple single-input linear layer stacks use the **Sequential API** (Lab 03), whereas modern deep architectures featuring branching pathways, multiple inputs/outputs, and **Residual Skip Connections** ($x + F(x)$) require the directed-acyclic-graph **Functional API** (Lab 05).

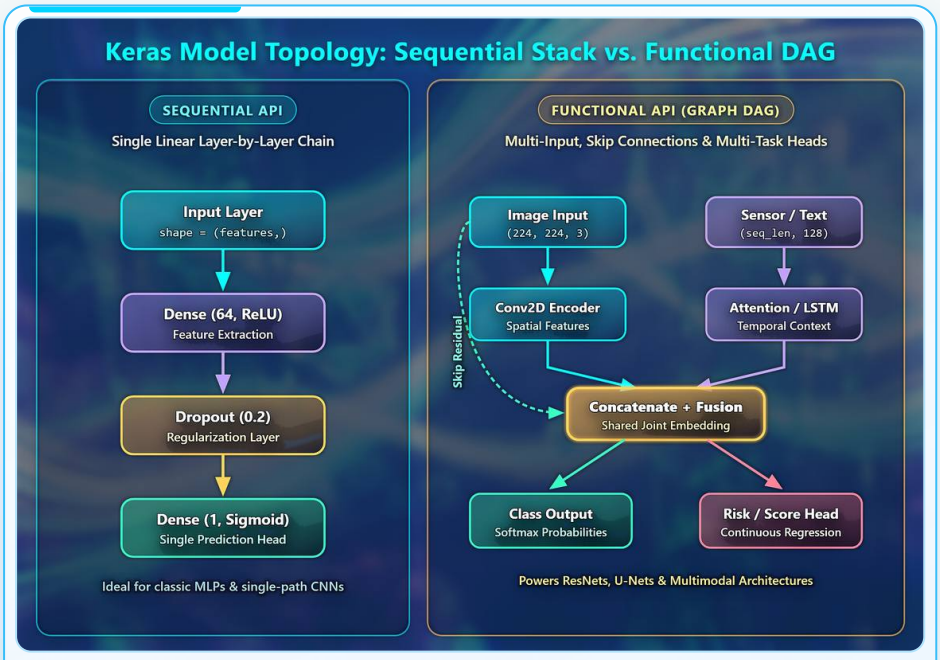


Figure 3.5

Architectural comparison between the Keras Sequential API (linear flow) and Functional API (directed graph with Skip Connections).

LAB 03 **AND Gate with Keras 3, AdamW & Native .keras Persistence** `code/03_and_gate_keras.py`

Rebuilds the AND classifier in Keras 3 using an explicit `keras.Input`, `AdamW` optimizer, `EarlyStopping(restore_best_weights=True)`, and secure native `.keras` serialization, verifying exact prediction parity after reloading from disk.

Guided Experiment: Inspect `model.summary()` to verify the exact parameter count (2×4 + 4 = 12 in the hidden layer, 4×1 + 1 = 5 in the output layer, total 17 parameters) and open `gate_and.keras` in Netron.

PYTHON 3.12+ `code/03_and_gate_keras.py` AND gate with Keras 3, AdamW & EarlyStopping

```python
from pathlib import Path
import numpy as np
import keras
from keras import layers


def build_and_model() -> keras.Model:
    """Creates a compact Sequential model for the AND logic gate."""
    model = keras.Sequential(
        [
            keras.Input(shape=(2,), name="binary_inputs"),
            layers.Dense(4, activation="relu", name="hidden_dense"),
            layers.Dense(1, activation="sigmoid", name="probability_output"),
        ],
        name="gate_and_classifier",
    )

    model.compile(
        optimizer=keras.optimizers.AdamW(learning_rate=0.05),
        loss="binary_crossentropy",
        metrics=["accuracy"],
    )
    return model


def main() -> None:
    keras.utils.set_random_seed(42)

    # Training dataset for the AND logic gate
    X_train = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float32)
    y_train = np.array([[0], [0], [0], [1]], dtype=np.float32)

    model = build_and_model()
    model.summary()

    # EarlyStopping automatically halts training once loss converges
    early_stop = keras.callbacks.EarlyStopping(
        monitor="loss", patience=40, min_delta=1e-4, restore_best_weights=True
    )
# ... (full executable code in code/03_and_gate_keras.py inside code.zip)
```

VERIFIED OUTPUT

```
Saved gate_and.keras | Predictions: ['[0, 0]->0.002', '[0, 1]->0.029', '[1, 0]->0.029', '[1, 1]->0.967']
```

LAB 05 **Sequential Stack vs. Functional DAG with Residual Skip Connection**

`code/05_sequential_vs_functional_api.py`

Directly contrasts a linear 3-class `keras.Sequential` classifier (8 inputs → 32 GELU → 32 GELU → 3 Softmax) against a `Functional API` directed graph combining an identity skip path `layers.Add()([projected, branch])` with `LayerNormalization`, mirroring the core residual block of ResNets and Transformers.

Guided Experiment: Execute the script and compare parameter totals and Softmax row sums (Σ = 1.000000) across both topologies; try stacking five residual blocks in the Functional model and confirm gradient flow remains unobstructed.

PYTHON 3.12+ `code/05_sequential_vs_functional_api.py` Sequential API vs. Functional API Residual Skip Block

```python
import numpy as np
import keras
from keras import layers


def build_sequential_model(
    input_features: int = 8, num_classes: int = 3
) -> keras.Model:
    """Builds a linear stack of dense layers (GELU -> Softmax) with Sequential API."""
    return keras.Sequential(
        [
            keras.Input(shape=(input_features,), name="seq_input"),
            layers.Dense(32, activation="gelu", name="seq_dense_1"),
            layers.Dense(32, activation="gelu", name="seq_dense_2"),
            layers.Dense(num_classes, activation="softmax", name="seq_output"),
        ],
        name="sequential_classifier",
    )


def build_functional_residual_model(
    input_features: int = 8, num_classes: int = 3
) -> keras.Model:
    """Builds a nonlinear graph with a Residual Connection via the Functional API."""
    inputs = keras.Input(shape=(input_features,), name="func_input")

    # Initial projection of the input vector into a 32-dimensional hidden space
    projected = layers.Dense(32, activation="gelu", name="projection")(inputs)

    # Main residual branch F(x) composed of two dense layers
    branch = layers.Dense(32, activation="gelu", name="residual_dense_1")(projected)
    branch = layers.Dense(32, activation=None, name="residual_dense_2")(branch)

    # Residual skip addition (x + F(x)) for smooth gradient flow in deep networks
    merged = layers.Add(name="skip_add")([projected, branch])
    merged = layers.LayerNormalization(name="layer_norm")(merged)
    merged = layers.Activation("gelu", name="post_norm_gelu")(merged)

    # Final projection to a Softmax probability distribution over num_classes
    outputs = layers.Dense(
        num_classes, activation="softmax", name="func_output"
    )(merged)
# ... (full executable code in code/05_sequential_vs_functional_api.py inside code.zip)
```

3.4 The Historic XOR Problem, Universal Approximation & Inference (Labs 06, 07 & 08)

In 1969, Marvin Minsky and Seymour Papert demonstrated that a single-layer Perceptron cannot solve the **XOR** (exclusive OR) logic gate because its positive points `(0,1), (1,0)` and negative points `(0,0), (1,1)` lie on opposite diagonals of the unit square: no single straight line can divide them. The resolution comes from the **Universal Approxima-**

tion Theorem (Cybenko, 1989; Hornik, 1991): a Multi-Layer Perceptron with just **one hidden layer** of nonlinear neurons and finite width can approximate any continuous function on a compact subset of R^n to arbitrary precision.

Intuitively, hidden neurons solve XOR by learning two linear sub-features in parallel (for instance, an *OR* boundary and a *NAND* boundary) and mapping the four original 2D corners into a warped hidden representation space where `(0,0)` and `(1,1)` collapse onto the same side of a single linear output hyperplane. Thus, hidden layers act as learned geometric coordinate transformations!

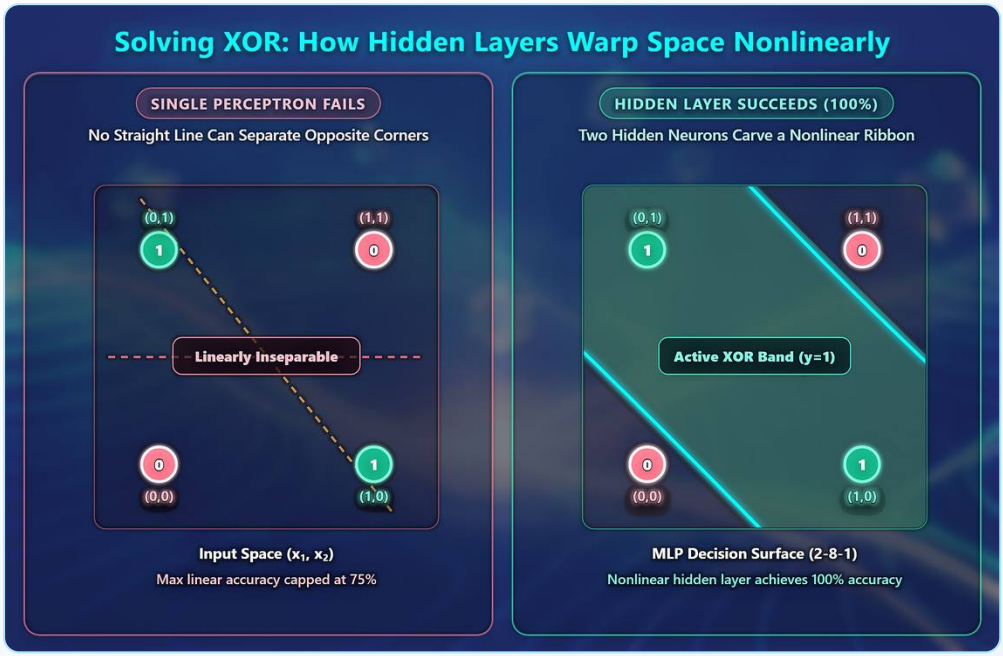


Figure 3.6a

Nonlinear decision boundary of the XOR gate achieved with a Multi-Layer Perceptron (MLP).

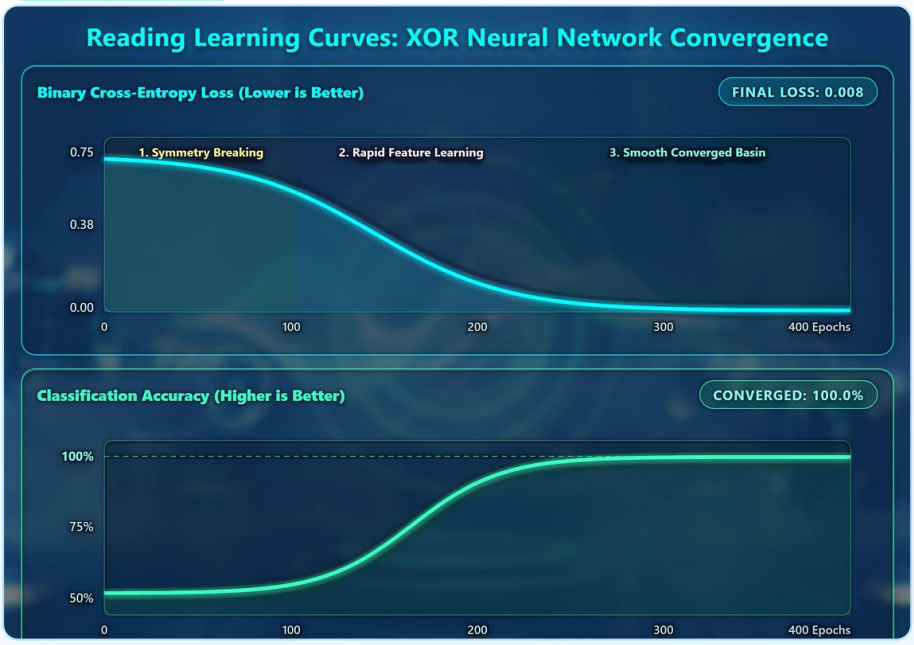


Figure 3.6b

Binary Cross-Entropy (BCE) loss and accuracy convergence curves during training of the XOR gate.

LAB 06 **Solving Nonlinear XOR from Scratch in Pure NumPy** `code/06_xor_gate_numpy.py`

Constructs a 2 → 4 → 1 MLP purely in NumPy using zero-centered `np.tanh` in the hidden layer (exact derivative $1 - a_1^2$) and Sigmoid + Binary Cross-Entropy at the output, driving BCE loss down from 0.6976 to 0.0017 in 6,000 transparent gradient-descent steps.

Guided Experiment: Replace `np.tanh(z1)` with pure linear identity `a1 = z1` (and derivative 1.0) to verify experimentally that without hidden nonlinearity the network stalls permanently at BCE ≈ 0.693 (50% coin flip).

PYTHON 3.12+ `code/06_xor_gate_numpy.py` Pure-NumPy 2-4-1 MLP with Tanh, BCE & exact backpropagation

```python
import numpy as np


def sigmoid(z: np.ndarray) -> np.ndarray:
    """Sigmoid activation function producing probabilities in (0, 1)."""
    return 1.0 / (1.0 + np.exp(-np.clip(z, -500.0, 500.0)))


def binary_cross_entropy(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """Computes binary cross-entropy (BCE) with clipping to avoid log(0)."""
    eps = 1e-12
    p = np.clip(y_pred, eps, 1.0 - eps)
    return float(-np.mean(y_true * np.log(p) + (1.0 - y_true) * np.log(1.0 - p)))


def main() -> None:
    """Trains a pure-NumPy MLP from scratch on the XOR logic gate."""
    # === 1. Prepare XOR Truth Table ===
    rng = np.random.default_rng(seed=2026)

    X = np.array(
        [
            [0.0, 0.0],
            [0.0, 1.0],
            [1.0, 0.0],
            [1.0, 1.0],
        ],
        dtype=np.float64,
    )
    y = np.array([[0.0], [1.0], [1.0], [0.0]], dtype=np.float64)

    # === 2. Initialize MLP Parameters (2 -> 4 -> 1) ===
    # Scale initial weights by 1 / sqrt(fan_in) for well-behaved Tanh inputs
    input_dim, hidden_dim, output_dim = 2, 4, 1
    W1 = rng.normal(0.0, np.sqrt(1.0 / input_dim), size=(input_dim, hidden_dim))
    b1 = np.zeros((1, hidden_dim), dtype=np.float64)
    W2 = rng.normal(0.0, np.sqrt(1.0 / hidden_dim), size=(hidden_dim, output_dim))
    b2 = np.zeros((1, output_dim), dtype=np.float64)

    learning_rate = 0.25
    epochs = 6_000
    n_samples = X.shape[0]

# ... (full executable code in code/06_xor_gate_numpy.py inside code.zip)
```

VERIFIED OUTPUT

```
Epoch 1 | BCE Loss: 0.697610   ->   Epoch 6000 | BCE Loss: 0.001720
(0,0)->0.000171 [0]  |  (0,1)->0.998002 [1]  |  (1,0)->0.997959 [1]  |  (1,1)->0.002662 [0]
```

LAB 07 **Training XOR in Keras 3 with GELU, EarlyStopping & Persistence** `code/07_xor_gate_keras.py`

Trains an 8-unit `GELU` hidden layer with Adam (`lr=0.05`) and `EarlyStopping`, saves the converged checkpoint to `xor_gate_model.keras`, and plots paired loss and accuracy learning curves.

Guided Experiment: Compare convergence epochs when switching hidden width from 8 down to 2 neurons (the theoretical minimum for XOR) and observe how extra width smooths the non-convex optimization landscape.

PYTHON 3.12+ `code/07_xor_gate_keras.py` — XOR solver in Keras 3 (GELU, Adam, EarlyStopping & .keras save)

```python
from pathlib import Path
import matplotlib.pyplot as plt
import numpy as np
import keras
from keras import layers


def main() -> None:
    keras.utils.set_random_seed(42)

    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float32)
    y = np.array([[0], [1], [1], [0]], dtype=np.float32)

    model = keras.Sequential(
        [
            keras.Input(shape=(2,), name="xor_input"),
            layers.Dense(8, activation="gelu", name="hidden_representation"),
            layers.Dense(1, activation="sigmoid", name="xor_output"),
        ],
        name="xor_solver_keras",
    )

    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=0.05),
        loss="binary_crossentropy",
        metrics=["accuracy"],
    )

    early_stop = keras.callbacks.EarlyStopping(
        monitor="loss", patience=60, min_delta=1e-4, restore_best_weights=True
    )

    history = model.fit(
        X,
# ... (full executable code in code/07_xor_gate_keras.py inside code.zip)
```

LAB 08 **Decoupled Production Inference: Reloading a Saved .keras Model** `code/08_xor_load_and_predict.py`

Demonstrates the canonical production separation between offline training and low-latency inference: loads `xor_gate_model.keras` created by Lab 07 via `keras.models.load_model` and executes deterministic forward passes with `training=False` in an interactive prompt loop.

Guided Experiment: Test not only exact binary corners (`0 1`, `1 1`) but also noisy intermediate coordinates such as `0.1 0.9` and `0.5 0.5` to probe the continuous probability surface learned by the GELU hidden layer.

PYTHON 3.12+ `code/08_xor_load_and_predict.py` — Loading saved xor_gate_model.keras & interactive inference loop

```python
from pathlib import Path
import numpy as np
import keras


def main() -> None:
    """Loads the saved XOR model and runs an interactive inference loop."""
    # === 1. Verify File Existence and Load Saved Model ===
    model_path = Path("xor_gate_model.keras")
```

```
    if not model_path.exists():
        print(
            f"Model file {model_path.name} was not found. "
            "Run 07_xor_gate_keras.py first to create it."
        )
        return

    model = keras.models.load_model(model_path)
    print(f"Successfully loaded model: {model_path.resolve()}")
    print("Enter two values (0 or 1) separated by a space, or 'q' to quit.\n")

    # === 2. Interactive Inference Loop ===
    while True:
        raw = input("Inputs (x1 x2): ").strip()
        if raw.lower() in ("q", "quit", "exit"):
            print("Exiting program.")
            break

        parts = raw.split()
        if len(parts) != 2:
            print("Please enter exactly two numbers, e.g. 1 0")
            continue

        try:
            x1, x2 = float(parts[0]), float(parts[1])
        except ValueError:
# ... (full executable code in code/08_xor_load_and_predict.py inside code.zip)
```

★ Complete Chapter 3 Laboratory Coverage (Labs 02 to 08)

Together, Labs `02`, `03`, `05`, `06`, `07`, and `08` trace the full mathematical and engineering progression from raw NumPy chain-rule gradients to multi-backend Keras 3 Residual DAGs and decoupled model serving. Companion Lab `04_and_gate_tensorboard.py` extends the AND workflow with live TensorBoard weight histograms and is covered in detail in Chapter 7.

CHAPTER 4

Deep Architectures: MLP, CNN, RNN, LSTM, and ConvLSTM2D

Matching network topology to data geometry: depth vs. width, 2D convolution arithmetic, receptive fields, full 4-gate LSTM equations, and spatiotemporal ConvLSTM2D.

Learning Objectives & Theoretical Scope

- Explain why architectural **depth** provides exponential representational efficiency over flat width, and how **BatchNormalization** and **Dropout** stabilize deep MLPs.
- Compute exact 2D convolution output dimensions $O = \lfloor (W - K + 2P) / S \rfloor + 1$, parameter counts, and hierarchical receptive-field growth across stacked 3×3 kernels.
- Derive all four gate equations of the **LSTM** cell (f_t, i_t, $\tilde{C}_t$, C_t, o_t, h_t) and explain how its additive cell-state highway solves vanishing gradients in BPTT.
- Generalize temporal recurrence to 5D spatiotemporal tensors *(B, T, H, W, C)* via **ConvLSTM2D** convolutional gates.

4.1 Deep Multi-Layer Perceptrons: Depth vs. Width, BatchNorm & Dropout

Although a single wide hidden layer is theoretically a universal approximator, approximating compositional real-world functions with a shallow network requires an exponentially growing number of hidden neurons. A **Deep Multi-Layer Perceptron (Deep MLP)** trades flat width for hierarchical **depth**: each successive dense layer reuses intermediate abstractions computed by earlier layers, representing complex curved decision regions with orders of magnitude fewer parameters.

FIRST PRINCIPLES & INTUITION **Matching the Network to the Signal: Stencils, Notebooks & Weather Maps**

Why do deep learning engineers use four distinct architectures instead of one giant MLP for everything? Because real-world data carries physical structure (space, time, or both), and matching that geometry makes learning dramatically faster and more reliable:

- **Deep MLP (Spreadsheet Committee):** Connects every input column directly to every hidden neuron, ideal for independent tabular measurements where column order has no geometric meaning.
- **CNN (Sliding Visual Stencil):** Slides a tiny reusable magnifying filter (called a **kernel**) over an image to detect the same edge or shape anywhere in the frame (**spatial weight sharing**).
- **LSTM (Detective's Notebook):** Reads sequences step by step while carrying a dedicated long-term **notebook** (the cell state C_t) controlled by three intuitive gates: a **Forget Gate** that erases outdated clues when a new topic begins, an **Input Gate** that writes down fresh evidence, and an **Output Gate** that decides which notes to read aloud right now.
- **ConvLSTM2D (Animated Weather Notebook):** Replaces flat list updates inside the LSTM notebook with 2D visual stencils so the network tracks spatial storms moving across maps over time.

Training deep stacks stably requires two core architectural regularizers. **BatchNormalization** normalizes mini-batch activations to zero mean and unit variance ($\hat{x} = (x - \mu_B) / \sqrt{(\sigma_B^2 + \varepsilon)}$) before applying learnable scale γ and shift β parameters, smoothing the loss landscape and permitting faster learning rates. **Inverted Dropout** stochastically zeroes a fraction p of hidden activations during training while scaling surviving units by $1 / (1 - p)$, preventing brittle neuron co-adaptation and approximating an exponential ensemble of thinned subnetworks (automatically disabled during inference when `training=False`).

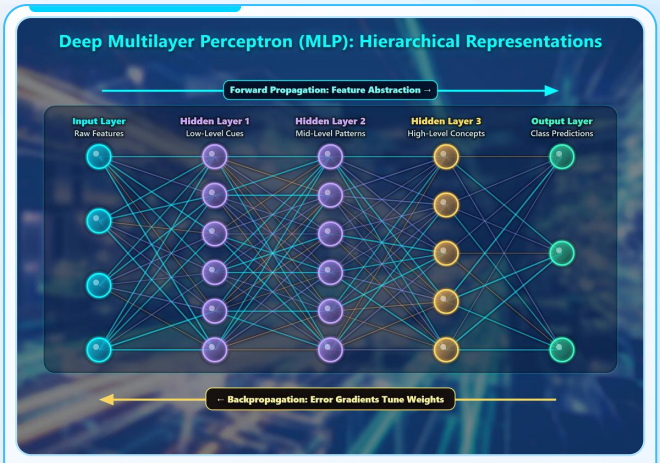


Figure 4.1

Internal anatomy of a deep Multi-Layer Perceptron (input vector, hierarchical hidden representation layers, weighted dense projections, and output).

4.2 Convolutional Neural Networks: 2D Arithmetic, Weight Sharing & Receptive Fields

Flattening a standard 224×224 RGB image into a dense MLP layer with 64 feature maps would require roughly **9.6 billion parameters** in the first layer alone, destroying 2D spatial locality and overfitting immediately. **Convolutional Neural Networks (CNNs)** eliminate this explosion via **local receptive fields** and **spatial weight sharing**: a single compact *3×3×3* filter scans every location of the image, requiring only *(3 × 3 × 3 + 1) × 64 = 1,792* parameters (over **5.3 million times fewer**) while guaranteeing translation equivariance. Two simple geometric dials control the scan: **zero-padding (*P*)** adds a thin border of zeros around the image so border pixels are not trimmed away, while **stride (*S*)** sets how many pixels the stencil jumps at each step.

$$O = \lfloor (W - K + 2P) / S \rfloor + 1 \quad | \quad \text{Params}_{\text{Conv2D}} = (K_h \cdot K_w \cdot C_{in} + 1) \cdot C_{out}$$

Where W is input spatial width, K is kernel size, P is zero-padding, and S is stride. Example (W=5, K=3): Valid (P=0, S=1) → O=3; Same (P=1, S=1) → O=5; Strided (P=0, S=2) → O=2.

STEP-BY-STEP WORKED EXAMPLE **Applying a 3x3 Vertical Edge Kernel to a 5x5 Pixel Patch by Hand**

Consider a tiny **5×5** grayscale image patch whose left two columns are dark (`0`) and right three columns are bright (`100`), so every row reads `[0, 0, 100, 100, 100]` (a crisp vertical boundary between columns 1 and 2). Slide the classic **3×3 vertical edge kernel** `[[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]]` across it with padding *P = 0* and stride *S = 1*:

1. **Output size check:** *O = ⌊(5 − 3 + 0) / 1⌋ + 1 =* ***3***, producing a **3×3** output feature map.
2. **Position (0, 0) covering columns 0, 1, 2:** Each of the 3 rows contains `[0, 0, 100]`. Multiply element-by-element with `[-1, 0, +1]` and sum across all 3 rows: *3 × [(−1)(0) + (0)(0) + (+1)(100)] =* ***+300*** (strong edge detection!).
3. **Position (0, 1) covering columns 1, 2, 3:** Each row contains `[0, 100, 100]`. Weighted sum across 3 rows: *3 × [(−1)(0) + (0)(100) + (+1)(100)] =* ***+300*** (edge sits directly under center column!).
4. **Position (0, 2) covering columns 2, 3, 4 (uniformly bright patch):** Each row contains `[100, 100, 100]`. Weighted sum: *3 × [(−1)(100) + (0)(100) + (+1)(100)] =* ***0*** (flat regions cancel identically!).

The complete 3×3 output map has rows `[300, 300, 0]`: the 9-weight kernel fires strongly wherever brightness jumps vertically and stays silent on flat surfaces.

Hierarchical Receptive Field Stacking: Rather than deploying heavy 5×5 or 7×7 kernels, modern vision architectures (VGG, ResNet, ConvNeXt) stack compact 3×3 convolutions. Each stacked layer expands the effective spatial receptive field according to $RF_l = RF_{l-1} + (K - 1) \cdot \Pi_{i<l} S_i$: two consecutive 3×3 layers capture the exact same 5×5 spatial context

as one 5×5 filter using **28% fewer parameters** (18 vs. 25 per channel) while inserting an extra nonlinear activation! Deeper layers combine MaxPooling downsampling with wider channel counts to progress from raw pixel edges (RF 3×3) to textures (RF 5×5) and whole semantic objects.

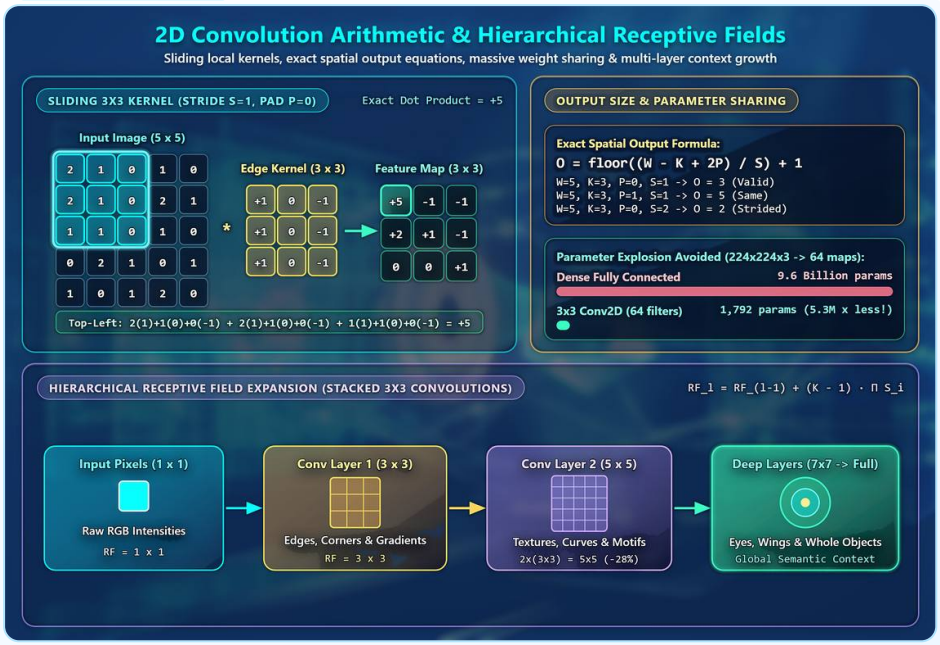


Figure 4.2

*2D Convolution arithmetic (worked 5x5 * 3x3 edge kernel dot product), exact spatial output formula, 5.3Mx parameter-sharing reduction, and hierarchical receptive-field growth.*

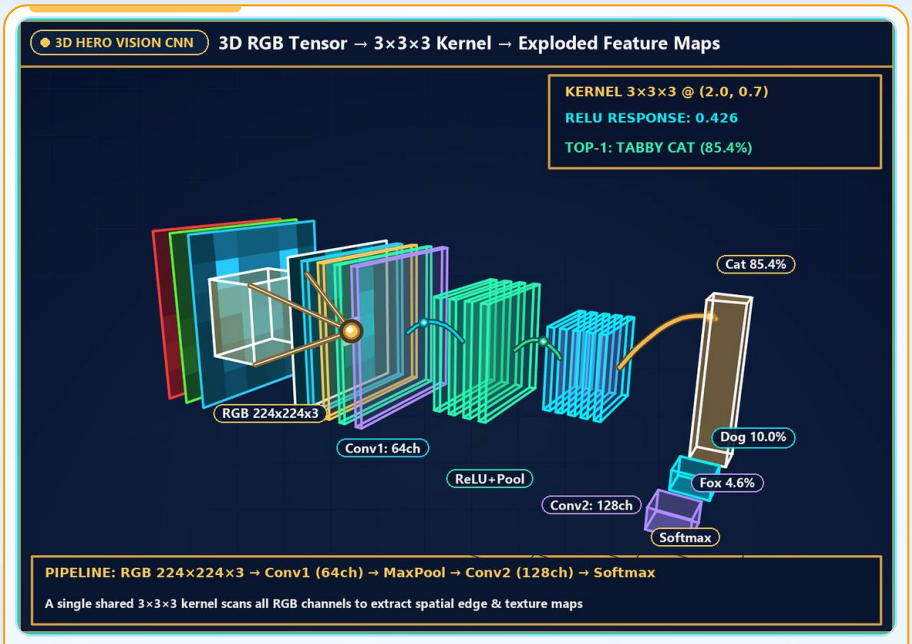


Figure 4.3

3D volumetric CNN visualization: a shared 3x3x3 convolutional kernel slides across an RGB 224x224x3 tensor to produce multi-channel feature maps, ReLU+Pool stages, and Softmax class probabilities.

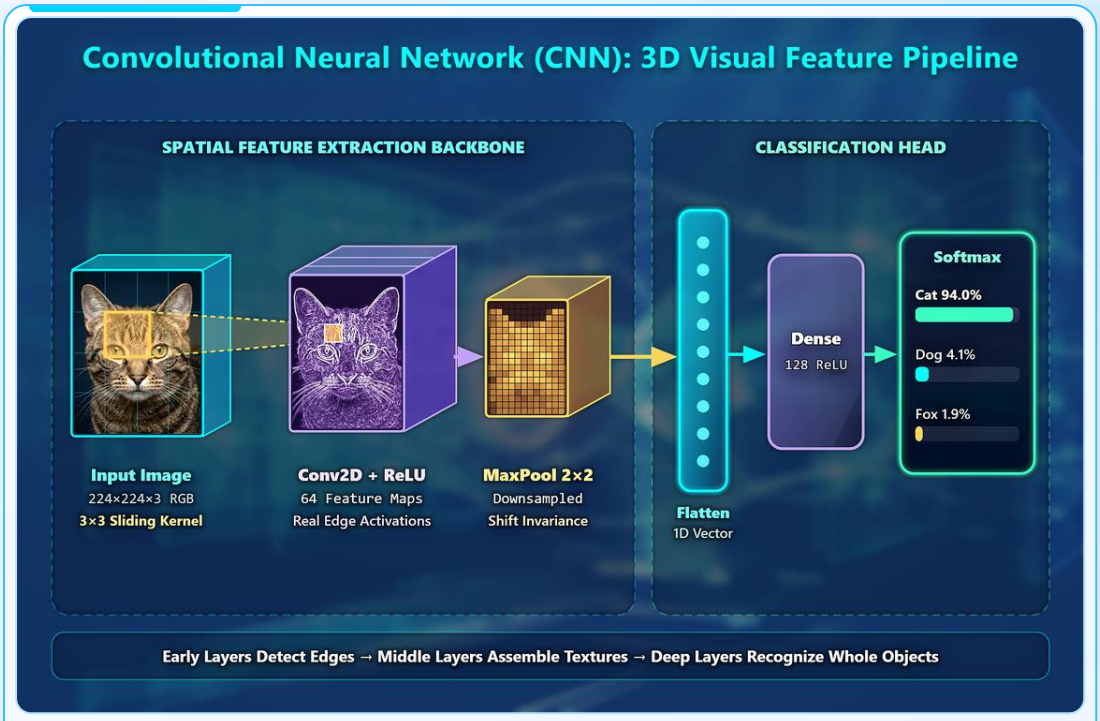


Figure 4.4

End-to-end CNN classification pipeline from input image through Conv2D + ReLU feature maps, MaxPooling, and Softmax decision layer.

4.3 Recurrent Networks (RNN), Full LSTM 4-Gate Equations & ConvLSTM2D

Sequential signals (language, audio, meteorological time series, electrocardiograms) carry critical temporal ordering. A vanilla **Recurrent Neural Network (RNN)** updates a shared hidden state $h_t = tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$ at every timestep. During **Backpropagation Through Time (BPTT)**, gradients flow backward by repeatedly multiplying by W_{hh}^T across T steps: if the spectral radius is below 1 gradients vanish exponentially, erasing long-term memory; if above 1 gradients explode (necessitating gradient norm clipping).

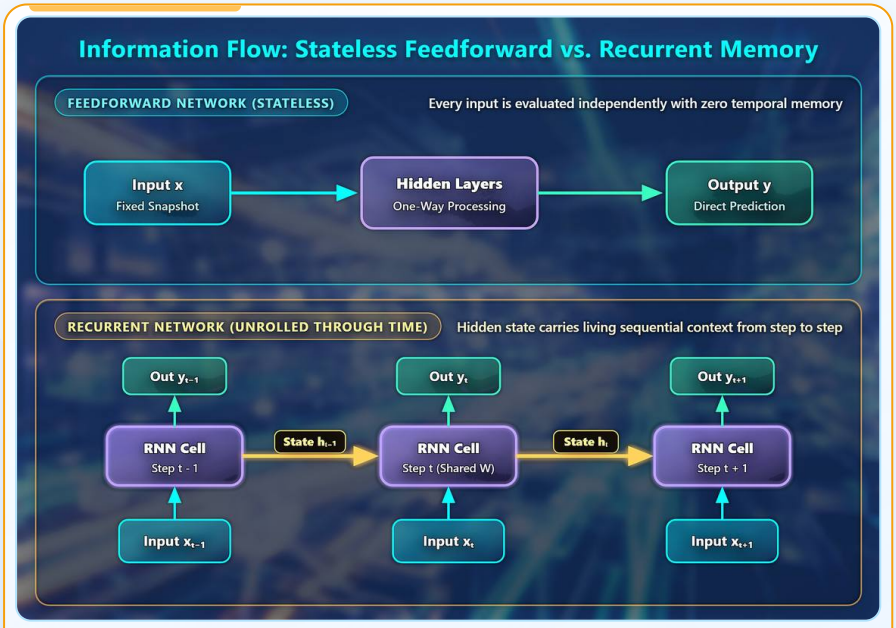


Figure 4.5a

Feedforward networks versus Recurrent Neural Networks (RNNs) unrolled across time steps.

Long Short-Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) overcomes vanishing temporal gradients by translating the detective's notebook intuition into exact vector algebra: it decouples long-term memory into a dedicated **cell-state highway** C_t regulated by four synchronized neural transformations. Here $\odot$ simply denotes element-by-element multiplication of two equal-length vectors, and every Sigmoid gate outputs values between *0* (close the valve completely) and *1* (let 100% of the signal pass through):

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad | \quad i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad | \quad C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad | \quad h_t = o_t \odot \tanh(C_t)$$

Forget gate f_t selects which prior memories to retain; Input gate i_t writes candidate $\tilde{C}_t$ into the additive cell highway C_t ($\partial C_t/\partial C_{t-1} = f_t$, preserving gradients); Output gate o_t exposes filtered state h_t.

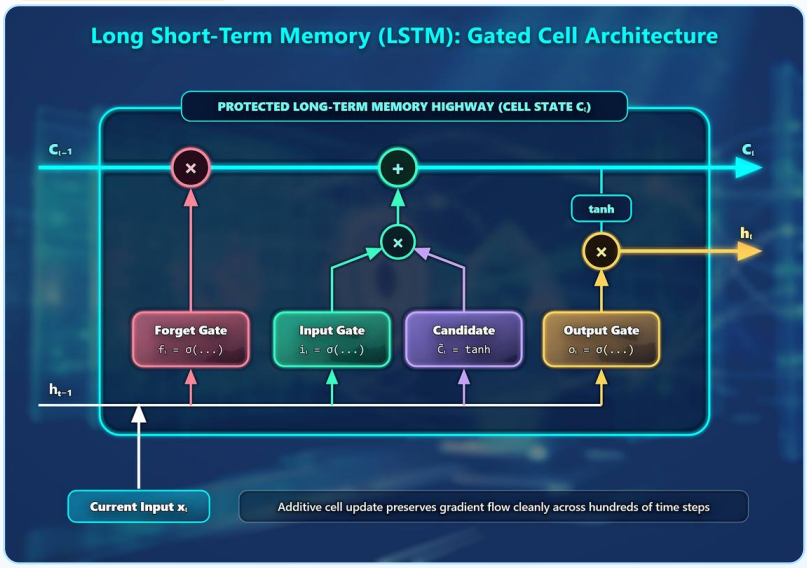


Figure 4.5b

Internal wiring of an LSTM memory cell showing the Forget, Input, and Output gates governing cell state Ct and hidden state ht.

Spatiotemporal ConvLSTM2D Gates: Standard LSTMs require 1D input vectors, destroying 2D geometry if applied to radar maps, satellite cloud frames, or gridded weather fields. **ConvLSTM2D** (Shi et al., 2015) preserves full spatial topology by replacing every dense matrix multiplication $W \cdot x$ and $W \cdot h$ inside all four LSTM gates with two-dimensional spatial convolutions $W *_{2D} X_t$ over 5D tensors *(Batch, Time, Height, Width, Channels)*. Consequently, each recurrent gate learns localized spatial dynamics such as storm-front advection and coupled atmospheric gradients (implemented directly in Chapter 13, Lab 17).

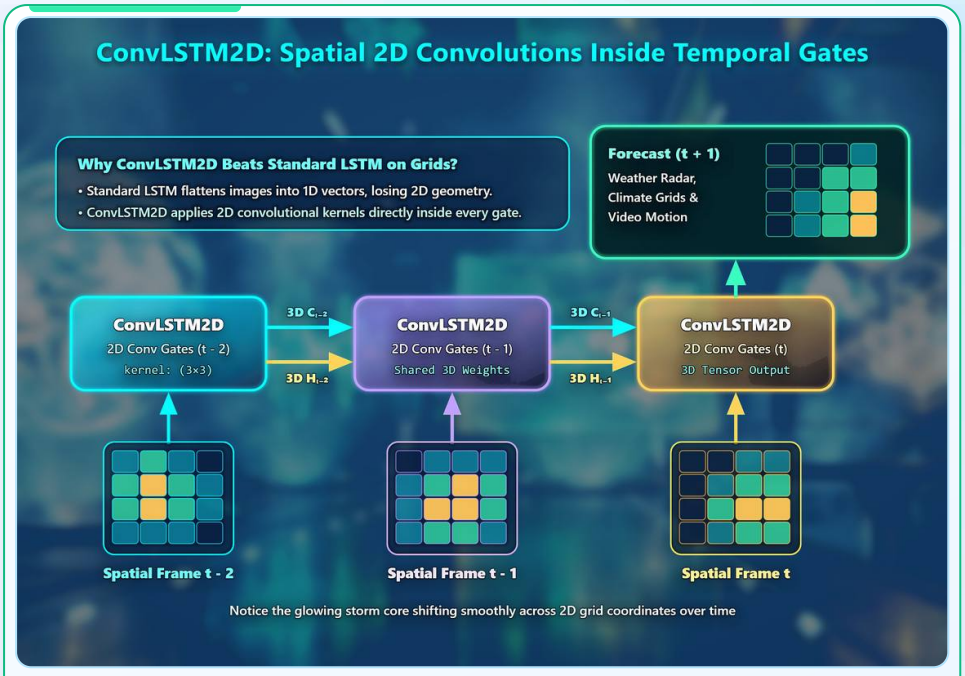


Figure 4.6

ConvLSTM2D architecture replacing dense gate projections with 2D spatial convolutions for spatiotemporal forecasting.

Architecture	Input Tensor Shape	Core Mechanism & Inductive Bias	Premier Application Domain
Deep MLP	2D tensor (B × F)	Dense projections + BatchNorm + Dropout + ReLU/GELU	Tabular data, logic gates, physical regression
CNN (2D)	4D tensor (B × H × W × C)	Local weight-shared 2D kernels + translation equivariance	Image recognition (MNIST, CIFAR-10), object detection
RNN / LSTM	3D tensor (B × T × F)	4-gate memory cell + additive gradient highway C_t	Time series, Patras meteorology, sentiment analysis (Bi-LSTM)
ConvLSTM2D	5D tensor (B × T × H × W × C)	2D convolutional gates inside a temporal LSTM recurrence	Spatiotemporal weather grids, precipitation radar, video

◆ **Practical Rule for Architecture Selection (Inductive Bias)**

Architecture selection always begins with the **geometry of the data**: for independent feature vectors choose a **Deep MLP**, for images and spatial spectra choose a **CNN**, for ordered sequences and time series choose **LSTM / Bi-LSTM** (or **Transformers** for long-range parallel context), and for spatial maps evolving across time choose **ConvLSTM2D**. Matching the inductive bias to the physics of the signal dramatically lowers sample complexity and strengthens generalization.

CHAPTER 5

Transformer Architecture and Frontier AI Models

Scaled Dot-Product and Multi-Head Self-Attention, variance stabilization, Pre-LN residuals, Transformer families (Encoder, Decoder, MoE), and Lab 22.

Learning Objectives & Theoretical Scope

- Derive **Scaled Dot-Product** and **Multi-Head Self-Attention**, explaining mathematically why dividing by $\sqrt{d_k}$ stabilizes Softmax gradient variance.
- Trace exact four-dimensional tensor shapes *[B, h, N, d_k]* alongside **Pre-LN Residual Connections** and Positional Encodings (Sinusoidal & RoPE).
- Contrast **Encoder-Only**, **Decoder-Only**, and **Sparse Mixture of Experts (MoE)** Transformer families and evaluate frontier capabilities (Test-Time Compute, Multimodality, AI Agents).
- Implement Multi-Head Self-Attention from scratch in pure NumPy and inspect live attention probability rows in **Lab 22**.

5.1 Scaled Dot-Product, Multi-Head Self-Attention & Pre-LN Residuals

Prior to the landmark paper "*Attention Is All You Need*" (Vaswani et al., 2017), sequence modeling depended on recurrent networks (RNNs and LSTMs) that processed tokens strictly one by one, creating an *O(N)* sequential bottleneck on GPUs and eroding long-range dependencies. The **Transformer** eliminated recurrence altogether, introducing **Scaled Dot-Product Self-Attention** so that every token interacts directly with every other token in parallel across an *O(1)* path length.

FIRST PRINCIPLES & INTUITION **Self-Attention as a Smart Library Search: Queries, Keys & Values**

Think about reading the sentence *"The river bank flooded after heavy rain."* How do you instantly know *bank* means a shoreline rather than a financial institution? Your eyes glance directly at nearby context words like *river* and *flooded*. A Transformer converts each word or subword piece (a **token**) into a numeric list (an **embedding**) and projects it into three simple library-search roles:

- **Query (Q):** "What contextual clue am I searching for?" (like your question at the library desk).
- **Key (K):** "What topic label do I advertise to other words?" (like the title printed on every book spine).
- **Value (V):** "What actual meaning should I share if you decide to pay attention to me?" (like the knowledge inside the book).

Each token compares its Query against every token's Key using a dot-product similarity score, converts those scores into attention percentages with **Softmax**, and builds a weighted blend of the Values so meaning flows globally across the sentence in one parallel step.

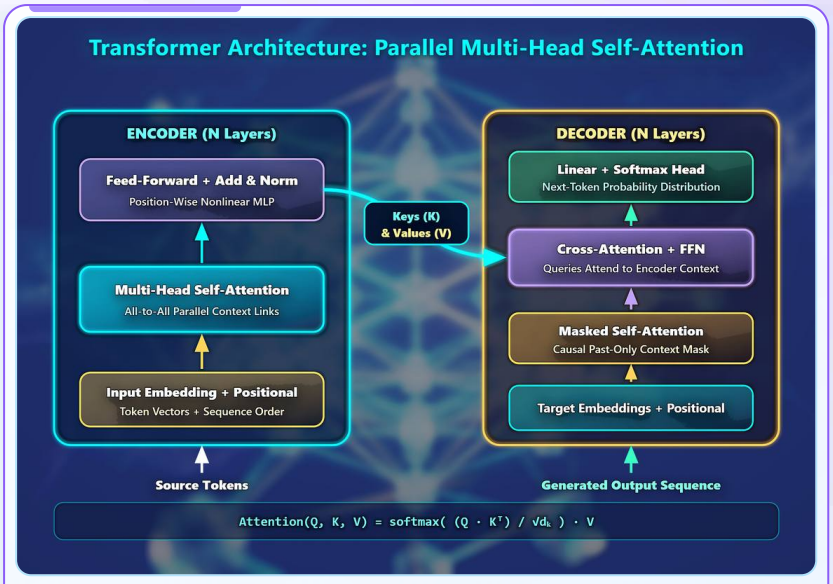


Figure 5.1

Transformer Architecture: Positional Encoding, Multi-Head Self-Attention, Residual Connections (Add & Norm), and Feed-Forward Network.

Why the Scaling Factor 1 / $\sqrt{d_k}$ Matters: Suppose the projected Query vector q and Key vector k have independent components with zero mean and unit variance. Their inner product $q \cdot k = \Sigma_{i=1..d_k} q_i k_i$ has mean 0 and variance $\mathbf{d_k}$ (standard deviation $\sqrt{d_k}$). For realistic head dimensions such as $d_k = 128$, unscaled dot products swing wildly into extreme positive or negative values, pushing Softmax into flat saturation regions where gradients vanish to near zero. Dividing by $\sqrt{d_k}$ restores unit variance identically, keeping attention weights smooth and trainable across hundreds of layers.

$$\mathbf{Attention(Q, K, V) = softmax((Q \cdot K^T) / \sqrt{d_k}) \cdot V}$$

$$\mathbf{MultiHead(Q, K, V) = Concat(head_1, ..., head_h) \cdot W^O \quad | \quad head_i = Attention(Q\, W_i^Q, K\, W_i^K, V\, W_i^V)}$$

Input tensor X [B, N, d_{model}] splits into h parallel heads of dimension $d_k = d_{model} / h$ (e.g., $d_{model} = 4096$, $h = 32 \Rightarrow d_k = 128$), projecting tensors to [B, h, N, d_k] before recombining through W^O.

STEP-BY-STEP WORKED EXAMPLE **Self-Attention by Hand on a 3-Word Sentence: ['The', 'dog', 'runs']**

Trace how the verb **"runs"** gathers context from the 3-word sentence `['The', 'dog', 'runs']` inside one attention head with dimension $d_k = 4$ (so $\sqrt{d_k} = 2.0$):

1. **Raw Query-Key dot-product scores:** Matching "runs"'s Query against the Keys of *[The, dog, runs]* produces raw similarity scores **[1.0, 2.0, 0.5]** ("dog" scores highest because it is the active subject).
2. **Scale by 1 / $\sqrt{d_k}$ = 1 / 2.0:** *[1.0 / 2, 2.0 / 2, 0.5 / 2]* = ***[0.50, 1.00, 0.25]***.
3. **Convert to Softmax attention probabilities:** Exponentiate ($e^{0.50} = 1.649$, $e^{1.00} = 2.718$, $e^{0.25} = 1.284$; sum = *5.651*) and divide: **α = [0.292, 0.481, 0.227]** (29.2% on "The", **48.1%** on "dog", 22.7% on "runs"; exact sum = 1.000).
4. **Weighted average of Value vectors:** Given 2D Value vectors $V_{The} = [0.0, 1.0]$, $V_{dog} = [2.0, 0.0]$, and $V_{runs} = [1.0, 2.0]$, the updated contextual vector for "runs" is: *0.292[0, 1] + 0.481[2, 0] + 0.227[1, 2]* = ***[1.189, 0.746]***. Nearly half of "runs"'s new representation now carries the identity of "dog," showing concretely how contextual meaning flows!

Through **Multi-Head Attention**, distinct subspaces specialize in parallel: one head resolves pronoun coreference, another captures local syntactic dependencies, and another links distant semantic concepts. Think of a **residual connection** (*x + Sublayer(x)*) as an express bypass lane: even if a deep layer initially learns small adjustments, the original token meaning and learning gradients travel intact across dozens of blocks. Two additional components complete every modern Transformer block:

- **Pre-LN Residual Connections:** Whereas the original 2017 Transformer applied LayerNorm after addition (*Post-LN*), virtually all modern large-scale models adopt **Pre-LN** ($x_{l+1} = x_l + Sublayer(Norm(x_l))$, typically with RMSNorm), preserving a clean identity gradient highway directly from the output loss back to token embeddings without fragile warmup instability.
- **Positional Encodings (Sinusoidal & RoPE):** Because pure dot-product attention is permutation-equivariant (a bag of tokens without order), sequence order is injected either via additive multi-frequency sinusoidal waves or via **Rotary Positional Embeddings (RoPE)**, which rotate Query and Key subspaces so inner products depend cleanly on exact relative token distance (explored deeply in Chapter 9).

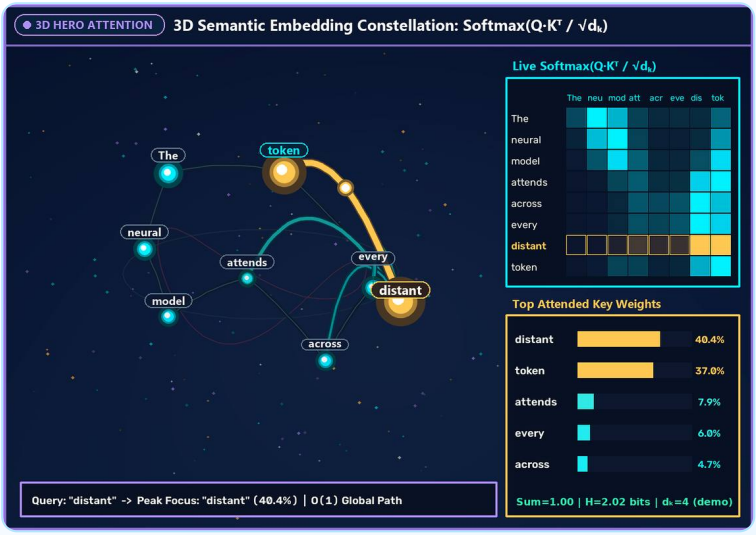


Figure 5.2

3D Semantic Embedding Constellation illustrating live Softmax(Q K^T / sqrt(d_k)) coupling, O(1) direct global paths across distant tokens, and row-normalized attention entropy.

5.2 Transformer Families (Encoder, Decoder, MoE) & Frontier AI Capabilities

Varying the attention mask and feed-forward routing yields three dominant architectural families deployed across modern research and production:

- **Encoder-Only (BERT, RoBERTa, BGE):** Uses an unmasked bidirectional $N \times N$ attention matrix pretrained via Masked Language Modeling (MLM); every token attends both left and right, making encoders ideal for dense RAG retrieval embeddings, classification, and Named Entity Recognition (NER).
- **Decoder-Only (GPT, Gemma, Llama, Qwen):** Applies a lower-triangular **causal mask** (setting upper-triangular logits to $-\infty$) pretrained via Next-Token Prediction; ideal for open-ended dialogue, code synthesis, and chain-of-thought reasoning.
- **Sparse Mixture of Experts (MoE: DeepSeek, Mixtral, Llama 4):** Replaces the dense feed-forward network with multiple specialized expert subnetworks guided by a learned router that activates only Top-K experts per token (typically ~20% active weights), decoupling massive total knowledge capacity from per-token FLOPs.

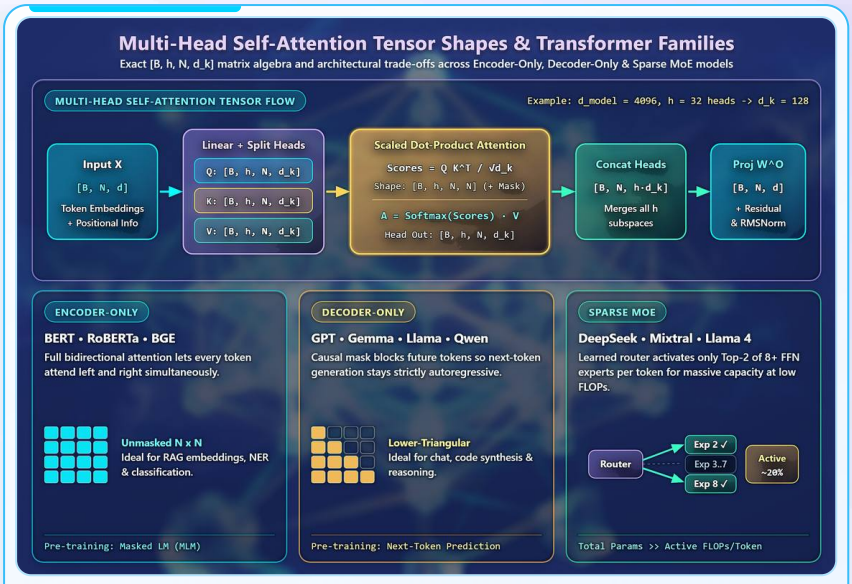


Figure 5.3

Multi-Head Self-Attention tensor flow [B, h, N, d_k] and architectural comparison of Encoder-Only (bidirectional), Decoder-Only (causal), and Sparse MoE Transformers.

Scaling these Transformer families ushered in today's generation of **Foundation Models**, anchored by four frontier capabilities:

Reasoning Models (Test-Time Compute)

Trained via reinforcement learning on verifiable problems in mathematics, science, and code, these models allocate extra inference compute (structured chains of thought and self-verification) to solve multi-step derivations reliably.

Sparse Mixture of Experts (MoE)

Dynamic Top-K routing activates only a fraction of parameters per token, delivering the knowledge capacity of hundreds of billions of weights at a fraction of dense inference latency and energy cost.

Native Multimodality & Long Context

Unified token streams process text, images, spectrograms, audio, video, and source code across multi-million-token windows, paired with **RAG** for grounded, auditable scientific answers.

Autonomous AI Agents

Language models coupled to external tools (terminals, databases, Python sandboxes, and Model Context Protocol servers) that plan, execute, inspect, and verify multi-step engineering workflows under human oversight.

5.3 Lab 22: Multi-Head Self-Attention in Pure NumPy

LAB 22 **Scaled Dot-Product & Multi-Head Self-Attention from Scratch** `code/22_transformer_self_attention.py`

Implements numerically stable row-wise Softmax, `scaled_dot_product_attention(Q, K, V)`, and parallel multi-head projection matrices `W_Q, W_K, W_V, W_O` in pure NumPy over a 6-token sentence (`d_model=16, num_heads=4, d_head=4`), printing every attention weight row and verifying exact probability normalization (Σ = 1.000).

Guided Experiment: Modify `scaled_dot_product_attention` to omit dividing by `np.sqrt(d_k)` and increase `d_model` to 512: observe how attention weights collapse toward rigid one-hot peaks (near-zero Shannon entropy), proving experimentally why variance scaling is indispensable.

PYTHON 3.12+ `code/22_transformer_self_attention.py` Scaled Dot-Product & Multi-Head Self-Attention in pure NumPy

```python
import numpy as np


def softmax(x: np.ndarray, axis: int = -1) -> np.ndarray:
    """Numerically stable Softmax along the specified axis."""
    shifted = x - np.max(x, axis=axis, keepdims=True)
    exp_vals = np.exp(shifted)
```

```
    return exp_vals / np.sum(exp_vals, axis=axis, keepdims=True)


def scaled_dot_product_attention(
    Q: np.ndarray, K: np.ndarray, V: np.ndarray
) -> tuple[np.ndarray, np.ndarray]:
    """
    Computes Scaled Dot-Product Attention:
        scores = (Q @ K^T) / sqrt(d_k)
        weights = softmax(scores)
        output = weights @ V
    Dividing by sqrt(d_k) prevents Softmax saturation when d_k is large.
    """
    d_k = Q.shape[-1]
    scores = (Q @ K.T) / np.sqrt(float(d_k))
    weights = softmax(scores, axis=-1)
    context = weights @ V
    return context, weights


def multi_head_attention(
    X: np.ndarray, num_heads: int, rng: np.random.Generator
) -> tuple[np.ndarray, list[np.ndarray]]:
    """Applies Multi-Head Self-Attention across parallel projection heads."""
    seq_len, d_model = X.shape
    if d_model % num_heads != 0:
        raise ValueError("d_model must be evenly divisible by num_heads.")

    d_head = d_model // num_heads
    head_outputs = []
    head_weights = []
# ... (full executable code in code/22_transformer_self_attention.py inside code.zip)
```

VERIFIED OUTPUT

```
Sequence length: 6 tokens | d_model = 16 | Heads = 4 | Output: (6, 16)
intelligence | 0.133  0.133  0.198  0.309  0.176  0.051  (Σ = 1.000)
learns       | 0.111  0.131  0.185  0.273  0.221  0.078  (Σ = 1.000)
```

Architectural Dimension	Recurrent Networks (RNN / LSTM)	Transformer Architecture
Sequential Processing	Serial step-by-step recurrence O(N)	Fully parallel across all N tokens during training
Long-Range Path Length	O(N) hops; gradients decay across long contexts	Direct O(1) pairwise connection between any two tokens
Attention Complexity	$O(N \cdot d^2)$ time, O(1) inference state	$O(N^2 \cdot d)$ self-attention; O(N) per step with KV-Cache
GPU Utilization & Scaling	Limited tensor-core parallelism	Exceptional scaling across trillions of training tokens

◈ Key Takeaway of Chapter 5

Every row of the self-attention matrix sums to exactly **1.000** (a Softmax probability distribution), dynamically weighting how strongly each query token draws context from every key token. Paired with **Pre-LN residuals**, **KV-Cache** memory, and Rotary Positional Embeddings (**RoPE**), this single matrix equation powers virtually all modern reasoning, multimodal, and scientific foundation models.

CHAPTER 6

Evaluation, Generalization, and the Model Lifecycle

Bias-Variance decomposition, regularization, master classification & regression metrics, leak-free validation splitting (K-Fold vs. Walk-Forward), and Labs 09 & 10.

Learning Objectives & Theoretical Scope

- Execute a disciplined end-to-end ML lifecycle with strict isolation among **Training**, **Validation**, and held-out **Test** splits.
- Decompose expected prediction error into **Bias^2**, **Variance**, and **Irreducible Noise (σ^2)**, pairing diagnostic curves with L1/L2 decay, Dropout, and EarlyStopping.
- Select and interpret the complete suite of classification and regression metrics (Accuracy, Precision, Recall, F1, ROC-AUC, PR-AUC, MAE, MSE, RMSE, R^2).
- Distinguish shuffled **K-Fold Cross-Validation** from chronological **Walk-Forward Validation** to eliminate future data leakage, and verify convergence laws in **Labs 09 and 10**.

6.1 The End-to-End Machine Learning Project Lifecycle

The scientific reliability of an AI system depends less on exotic architectures than on engineering discipline across the complete iterative workflow: clear problem formulation, auditable data ingestion and cleaning, strict separation into **Training**, **Validation**, and **Test** sets, controlled hyperparameter tuning, reproducible artifact versioning, and post-deployment drift monitoring.

FIRST PRINCIPLES & INTUITION **Studying for a University Exam: Memorization vs. True Understanding**

Why do data scientists guard three separate datasets so strictly? Compare training a neural network to preparing for a tough university final exam:

- **Training Set:** The homework exercises you practice repeatedly to build core skill.
- **Validation Set:** A practice mock exam you take midway to tune your study habits (your **hyperparameters**, such as learning rate or network size).
- **Test Set:** The sealed final exam opened only *once* at the very end to measure honest real-world **generalization**.

Skimming the textbook too lightly causes **Underfitting (High Bias)**, where you miss even basic homework questions. Memorizing homework answers verbatim without grasping the principles causes **Overfitting (High Variance)**: you score 99% on practice problems yet fail unfamiliar exam questions. Finally, peeking at the final exam while studying ruins the fairness of your score completely (**Data Leakage**).

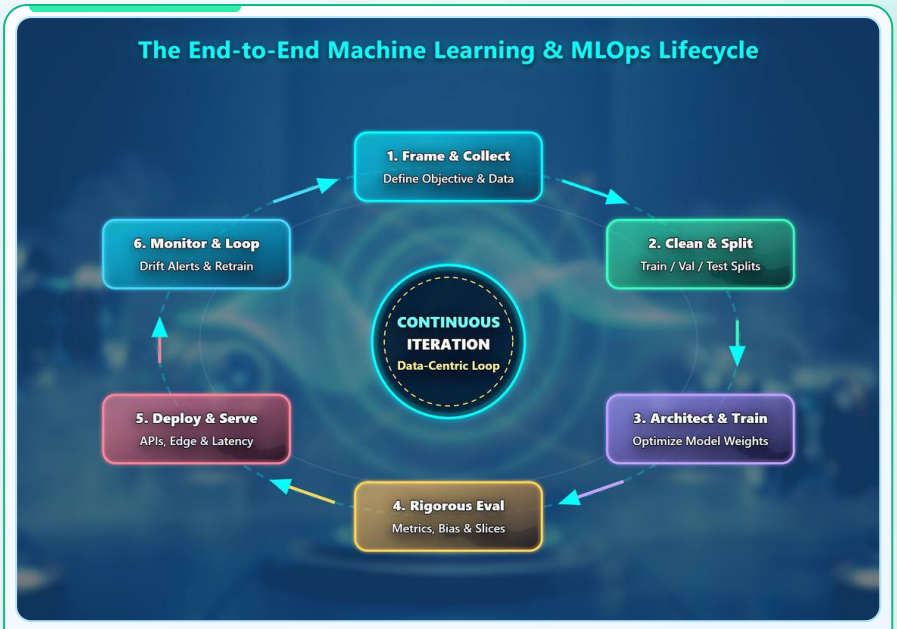


Figure 6.1

The complete iterative Machine Learning model lifecycle (data curation, training, validation, deployment, and continuous monitoring).

6.2 Bias-Variance Decomposition, Regularization & Validation Strategies

Hyperparameters (*learning rate*, *batch size*, *epochs*, network depth, and regularization strength) are fixed before training and govern model capacity. Mathematically, we can split any model's unseen test error into three clear ingredients: systematic oversimplification (**Bias^2**), oversensitivity to random quirks of the training sample (**Variance**), and unavoidable real-world measurement randomness (**Irreducible Noise σ^2**). For any squared-error estimator $\hat{y} = f_\theta(x)$ targeting noisy observations $y = f^*(x) + \varepsilon$ (where $Var(\varepsilon) = \sigma^2$), expected unseen test error decomposes algebraically into these three orthogonal components:

$$\mathbf{E[\,(y - \hat{y})^2\,] = (\,E[\hat{y}] - f^*(x)\,)^2 + E[\,(\hat{y} - E[\hat{y}])^2\,] + \sigma^2 \;=\; Bias^2 + Variance + \sigma^2}$$

High Bias produces Underfitting (systematic error from an overly rigid model); High Variance produces Overfitting (memorizing finite training noise); σ^2 is the irreducible aleatoric noise floor.

Taming variance without inflating bias relies on four complementary regularizers: **L1 regularization (Lasso)** adds $\lambda \Sigma |w_i|$ to induce sparse zero weights; **L2 regularization (Ridge / Weight Decay)** adds $\lambda \Sigma w_i^2$ to shrink large norms smoothly; **Dropout** and **Data Augmentation** inject stochastic invariances; and **Early Stopping** halts training the moment validation loss stops improving while restoring the lowest-loss checkpoint (`restore_best_weights=True`).

Equally critical is choosing the right validation protocol to prevent **data leakage**:

- **Shuffled K-Fold Cross-Validation (i.i.d. Tabular & Vision Data):** Partitions data into K equal folds (typically $K = 5$), rotating each fold once as validation and averaging performance to reduce evaluation variance across small or medium datasets.
- **Chronological Walk-Forward Validation (Time Series & Meteorology):** Shuffling temporal observations leaks future climate or market states backward into the training split, producing wildly optimistic yet useless test scores. Time-series models MUST train strictly on past intervals, validate on subsequent future blocks, and fit every normalization scaler (`MinMaxScaler`, `StandardScaler`) **exclusively on the past training partition**.

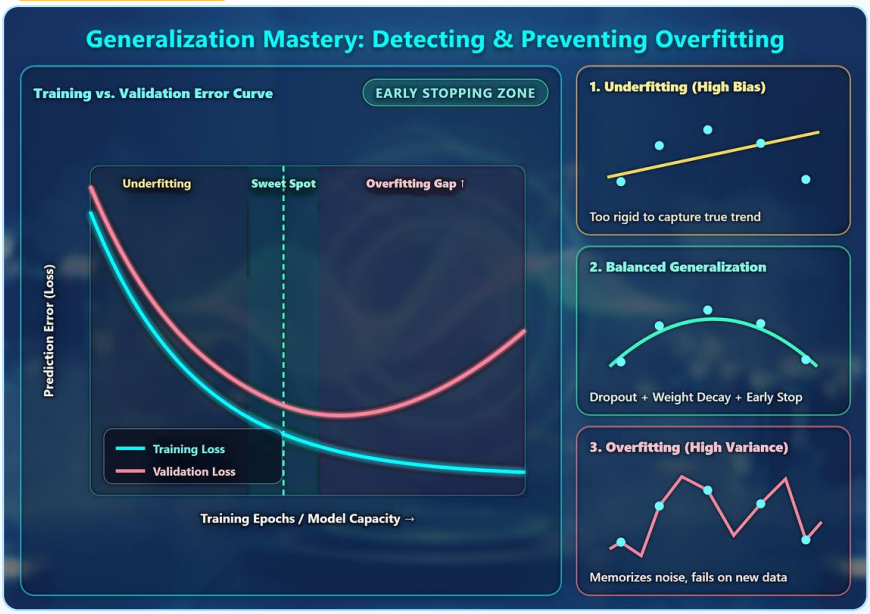


Figure 6.2a

Diagnostic comparison of Underfitting (High Bias), Optimal Generalization, and Overfitting (High Variance).

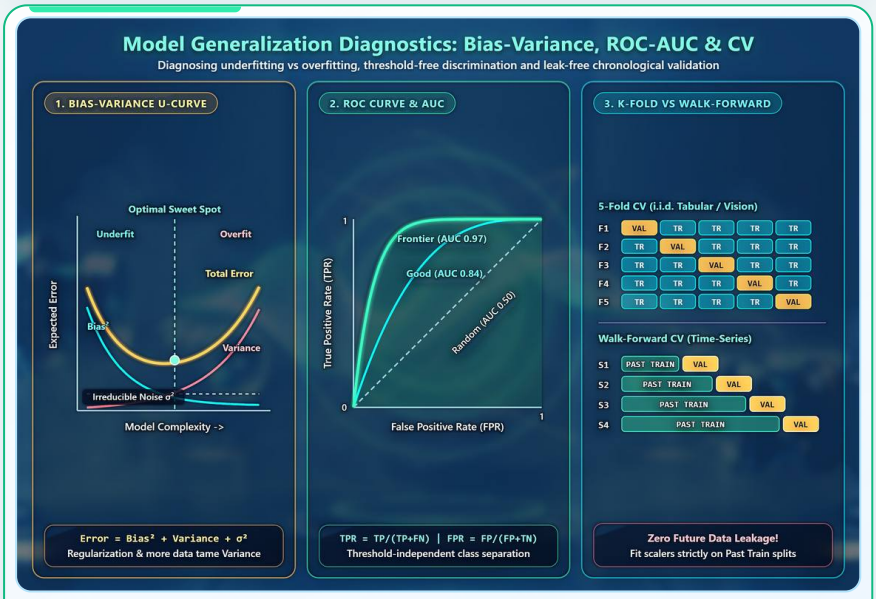


Figure 6.2b

Model Generalization Diagnostics: Bias-Variance U-Curve, threshold-free ROC-AUC discrimination, and K-Fold vs. leak-free Walk-Forward Cross-Validation.

6.3 Master Classification & Regression Metrics Framework

Selecting an unsuitable metric can disguise a broken model: on a rare-event dataset where 99% of samples are negative (such as medical pathology screening or SETI technosignature detection), a trivial model predicting "negative" everywhere scores 99% Accuracy while achieving 0% Recall. To see exactly where a classifier succeeds or fails, engineers sort every prediction into the four quadrants of a **Confusion Matrix**: **True Positives (TP)** (correctly caught cases), **True Negatives (TN)** (correctly cleared cases), **False Positives (FP)** (false alarms), and **False Negatives (FN)** (missed real cases). The master reference table below pairs every standard metric with its exact mathematical definition and engineering use case.

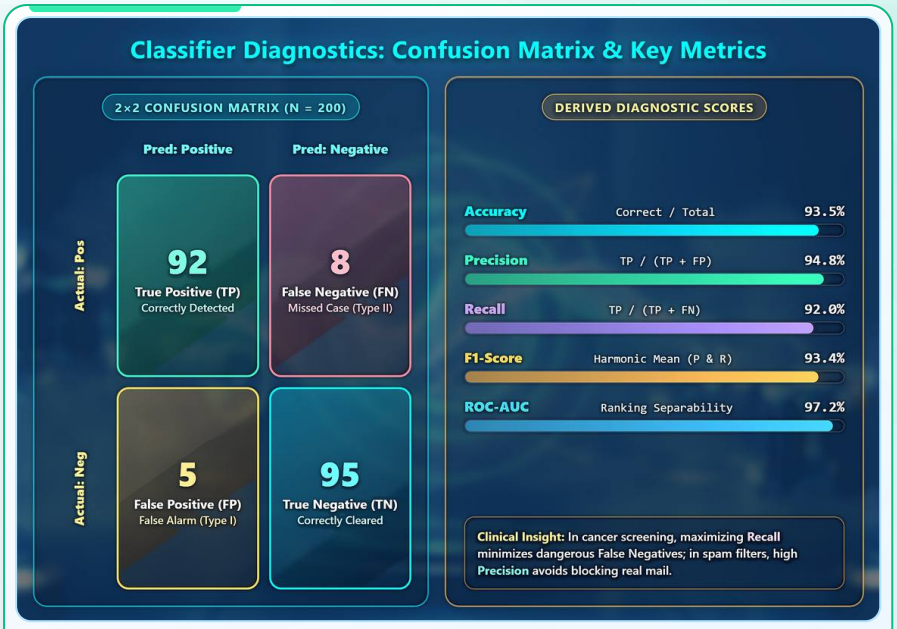


Figure 6.3

Confusion Matrix (TP, TN, FP, FN) and geometric derivation of Accuracy, Precision, Recall, F1-Score, and ROC-AUC.

$$\text{RMSE} = \sqrt{[\,(1/N)\,\Sigma\,(y_i - \hat{y}_i)^2\,]} \;\;|\;\; \text{Precision} = TP/(TP+FP) \;\;|\;\; \text{Recall} = TP/(TP+FN) \;\;|\;\; F1 = 2{\cdot}P{\cdot}R/(P+R)$$

RMSE reports regression error in the physical units of the target (e.g., °C or hPa), while F1 is the harmonic mean of Precision and Recall.

STEP-BY-STEP WORKED EXAMPLE **The Accuracy Paradox: Medical Screening on 100 Patients**

Suppose a clinic screens **100 patients** for a rare condition: **10 patients are truly positive** and **90 patients are healthy**. Compare two competing models on the exact same cohort:

1. **Naive "Always Healthy" Baseline:** Predicts negative for all 100 people without examining any test data. Counts: *TP = 0, FN = 10, FP = 0, TN = 90*. **Accuracy = (0 + 90) / 100 = 90.0%** (deceptively high!), yet **Recall = 0 / (0 + 10) = 0.0%** (every single sick patient goes undiagnosed, a catastrophic clinical failure!).
2. **Real Diagnostic Classifier:** Flags 14 patients (9 genuinely positive plus 5 false alarms) and clears 86 patients (85 healthy plus 1 missed case). Counts: *TP = 9, FN = 1, FP = 5, TN = 85*.
3. **Exact Metrics for the Real Detector: Accuracy** = (9 + 85) / 100 = **94.0%**; **Precision** = 9 / (9 + 5) = 9 / 14 = **64.3%** (reliability of positive alarms); **Recall** = 9 / (9 + 1) = 9 / 10 = **90.0%** (9 of 10 true cases caught!); **F1-Score** = 2 × (9/14) × (9/10) / (9/14 + 9/10) = 18 / 24 = **75.0%**.

Takeaway: Whenever positive events are rare (medicine, fraud, or radio-astronomy signals), never evaluate by Accuracy alone: always report Confusion Matrix counts, Precision, Recall, F1, and PR-AUC.

Metric	Task	Formula	Engineering Interpretation & Best Use Case
Accuracy	Classif.	(TP + TN) / N	Overall correctness; valid ONLY on balanced classes (e.g., MNIST, CIFAR-10)
Precision (P)	Classif.	TP / (TP + FP)	Fraction of predicted positives truly positive; critical when false alarms are costly
Recall (TPR)	Classif.	TP / (TP + FN)	Fraction of actual positives detected; critical in medical triage & rare-signal search
F1-Score	Classif.	2 · P · R / (P + R)	Harmonic mean penalizing extreme imbalance between Precision and Recall

Metric	Task	Formula	Engineering Interpretation & Best Use Case
ROC-AUC	Classif.	$\int$ TPR d(FPR)	Threshold-free ranking quality (0.5 random, 1.0 perfect separation across thresholds)
PR-AUC	Classif.	$\int$ P d(R)	Preferred over ROC-AUC under severe class imbalance (ignores overwhelming TN count)
MAE	Regress.	$(1/N)\ \Sigma\ \lvert y_i - \hat{y}_i \rvert$	Mean absolute error in physical units; robust against isolated extreme outliers
MSE	Regress.	$(1/N)\ \Sigma\ (y_i - \hat{y}_i)^2$	Smooth differentiable training loss heavily penalizing large prediction spikes
RMSE	Regress.	$\sqrt{\text{MSE}}$	Standard physical-unit reporting metric sensitive to tail errors (e.g., °C in Lab 16)
R^2	Regress.	$1 - SS_{res} / SS_{tot}$	Proportion of target variance explained by the model (1.0 ideal, 0.0 mean baseline)

6.4 Labs 09 & 10: Statistical Metrics & Monte Carlo Convergence (π)

LAB 09 **Regression & Binary Classification Metrics Pipeline with Pandas**

`code/09_metrics_rmse_and_dataframe.py`

Computes `MSE`, `RMSE`, `MAE`, and R^2 alongside the full Confusion Matrix (`TP, TN, FP, FN`) and Accuracy/Precision/Recall/F1 from first-principles NumPy equations, assembling structured Pandas DataFrames for transparent model auditing.

Guided Experiment: Inject a single large outlier (+25 °C error) into the synthetic regression predictions and measure how sharply RMSE jumps compared to MAE, demonstrating quadratic outlier sensitivity.

PYTHON 3.12+ `code/09_metrics_rmse_and_dataframe.py` First-principles MSE, RMSE, MAE, R2 & Confusion Matrix

```python
import numpy as np

try:
    import pandas as pd
except ImportError:
    pd = None


def compute_regression_metrics(
    y_true: np.ndarray, y_pred: np.ndarray
) -> dict[str, float]:
    """Computes core error and goodness-of-fit metrics for regression."""
    errors = y_true - y_pred
    mse = float(np.mean(errors ** 2))
    rmse = float(np.sqrt(mse))
    mae = float(np.mean(np.abs(errors)))
    ss_res = float(np.sum(errors ** 2))
    ss_tot = float(np.sum((y_true - np.mean(y_true)) ** 2))
    r2 = 1.0 - (ss_res / ss_tot) if ss_tot > 0 else 0.0
    return {"MSE": mse, "RMSE": rmse, "MAE": mae, "R2": r2}


def compute_binary_classification_metrics(
    y_true: np.ndarray, y_pred: np.ndarray
) -> dict[str, float]:
    """Computes TP, TN, FP, FN alongside Accuracy, Precision, Recall, F1."""
    tp = int(np.sum((y_true == 1) & (y_pred == 1)))
    tn = int(np.sum((y_true == 0) & (y_pred == 0)))
    fp = int(np.sum((y_true == 0) & (y_pred == 1)))
    fn = int(np.sum((y_true == 1) & (y_pred == 0)))
```

```
    total = max(tp + tn + fp + fn, 1)
# ... (full executable code in code/09_metrics_rmse_and_dataframe.py inside code.zip)
```

LAB 10 Vectorized Monte Carlo Estimation of Pi & O(N^-1/2) Convergence

`code/10_calc_pi_monte_carlo_nn.py`

Samples up to 5,000,000 uniform 2D coordinates inside the unit square to estimate π via the quarter-circle area ratio $\pi/4$, averaging 5 trials per sample size and fitting a log-log linear regression (`np.polyfit`) to verify the universal stochastic convergence law $Error \propto N^{-1/2}$ (slope ≈ -0.5).

Guided Experiment: Increase trial averaging from 5 to 25 runs per sample size and verify that the empirical log-log slope tightens even closer to the exact theoretical −0.5000 Central Limit Theorem rate.

PYTHON 3.12+ `code/10_calc_pi_monte_carlo_nn.py` Vectorized Monte Carlo π estimator & log-log slope verification

```python
import math
import numpy as np


def estimate_pi_monte_carlo(num_samples: int, rng: np.random.Generator) -> float:
    """Estimates pi from num_samples uniform random points in [0, 1]^2."""
    # Vectorized sampling of (x, y) coordinates for high throughput
    x = rng.random(num_samples, dtype=np.float64)
    y = rng.random(num_samples, dtype=np.float64)

    # Check distance from the origin: x^2 + y^2 <= 1
    inside_quarter_circle = np.sum((x * x + y * y) <= 1.0)

    # Area ratio (quarter circle / unit square) equals pi / 4
    return 4.0 * float(inside_quarter_circle) / float(num_samples)


def main() -> None:
    rng = np.random.default_rng(seed=42)
    sample_sizes = [
        1_000, 5_000, 10_000, 50_000, 100_000, 500_000, 1_000_000, 5_000_000
    ]

    print(f"True value of pi (math.pi): {math.pi:.10f}\n")
    print(f"{'Samples N':>12s} | {'Pi Estimate':>14s} | {'Absolute Error':>16s}")
    print("=" * 50)

    mean_errors = []
    for n in sample_sizes:
        # Average 5 independent trials per N for a stable error estimate
# ... (full executable code in code/10_calc_pi_monte_carlo_nn.py inside code.zip)
```

VERIFIED OUTPUT

```
N = 1,000 -> π ≈ 3.1496 (error 0.0413)  |  N = 5,000,000 -> π ≈ 3.1422 (error 0.00084)
Empirical log-log slope: -0.4231 (theoretical value: -0.5000, O(N^-1/2) law)
```

CHAPTER 7

Software Ecosystem, Visualization & Model Inspection

The modern Python 3.12+ AI engineering stack, live TensorBoard telemetry, Netron graph analysis, H5Web weight inspection, a systematic debugging protocol, and Lab 04.

Learning Objectives & Theoretical Scope

- Select the right layer of the modern **AI Engineering Stack** across NumPy 2, Pandas 2, Keras 3, PyTorch 2, Hugging Face, ONNX, TensorBoard, Netron, and H5Web.
- Transform opaque neural networks into observable scientific instruments by auditing scalar curves, weight histograms, computational graphs, and raw HDF5 tensors.
- Apply a repeatable **5-Step Model Debugging & Observability Protocol** before committing compute to multi-hour training runs.
- Instrument timestamped TensorBoard callbacks and inspect live histograms inside Visual Studio Code in **Lab 04**.

7.1 Complete AI Engineering Stack in Python 3.12+

Production-grade machine learning is built on a modular, interoperable stack spanning numerical array foundations, multi-backend deep learning compilers, open model hubs, cross-runtime serialization, and visual telemetry tools:

FIRST PRINCIPLES & INTUITION **Why AI Engineers Use a Cockpit of Tools (Not Just One Script)**

Training a deep neural network without visual inspection tools is like flying an aircraft at night with the cockpit gauges covered: you cannot tell whether the model is climbing smoothly, stalling, or drifting off course until it crashes. Every layer of the Python AI ecosystem acts as one specialized cockpit instrument:

- **Tensors (NumPy & Pandas):** A *tensor* is simply a multi-dimensional grid of numbers: a 1D vector is a list, a 2D matrix is a table, a 3D tensor is one color image, and a 4D tensor is a **batch** of images processed together.
- **Training Engines (Keras 3 & PyTorch 2):** Build layers and automatically compute every backpropagation derivative (**autograd**) without manual calculus.
- **Live Telemetry (TensorBoard):** Your real-time dashboard plotting training/validation loss curves and weight histograms while the model learns.
- **X-Ray Scanners (Netron & H5Web):** Open saved model files (`.keras`, `.onnx`, `.h5`) after training to inspect every connection and numerical weight matrix directly.

Tool / Framework	Stack Layer	Core Engineering Capabilities	Primary Artifact / Role
NumPy 2	Array Compute	Contiguous C-speed N-dimensional arrays, BLAS/LAPACK matrix math	Pure-math backprop, attention & signal processing (Labs 02, 06, 22)
Pandas 2	Tabular & Time	Labeled DataFrames, datetime indexing, leakage-safe slicing & CSV I/O	Metric tables & meteorological series (Labs 09, 16, 17)
Keras 3	High-Level DL	Multi-backend API running identically on TensorFlow, PyTorch, or JAX	Rapid Sequential/Functional models & safe `.keras` files
PyTorch 2	Research & LLM	Dynamic autograd, `torch.compile`, native CUDA & `torch.amp`	Custom kernels, AMP benchmarks, LoRA & frontier LLM stacks

Tool / Framework	Stack Layer	Core Engineering Capabilities	Primary Artifact / Role
Hugging Face	Pretrained Hub	Standardized tokenizers, MarianMT/LLM weights, PEFT & `pipelines`	Zero-friction transfer learning & neural translation (Lab 15)
ONNX / ORT	Interchange	Open graph specification decoupling training framework from C++/edge runtime	Cross-platform `.onnx` deployment on CPU, GPU & embedded targets
TensorBoard	Live Telemetry	Real-time loss/metric curves, layer weight histograms, graph & profiler traces	Timestamped `logs/run_*` comparison inside VS Code
Netron	Graph Viewer	Interactive topology & tensor-dimension visualizer for saved model archives	Audits `.keras`, `.h5`, `.onnx`, `.pt` & `.gguf`
H5Web & Neuroph	Weight Inspector	VS Code HDF5 matrix/heatmap explorer (H5Web); visual Java sandbox (Neuroph)	Direct numeric inspection of `.h5` weight matrices (Lab 02)

7.2 Training Telemetry, Architecture Inspection & Debugging Protocol

A neural network should never be treated as an opaque black box. Combining live metric logging with post-training graph and tensor inspection turns deep learning from trial and error into transparent scientific measurement. In everyday practice, two beginner habits save hours of frustration: verifying exact tensor shapes through `model.summary()` before starting training, and relying on clean **model serialization** (saving both the layer blueprint and learned weights into a portable `.keras` or `.onnx` file so models reload reliably without retraining):

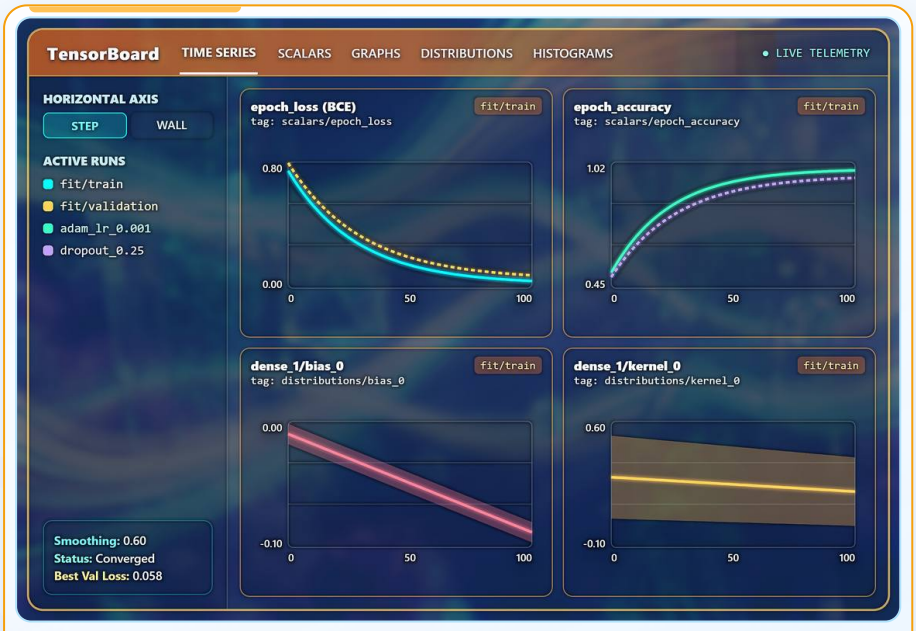


Figure 7.1

TensorBoard: live monitoring of Loss and Accuracy curves alongside per-epoch synaptic weight histograms.



Figure 7.2

Netron: interactive visualization of layer topology, kernel attributes, and tensor dimensions inside .keras, .h5, and .onnx model files.

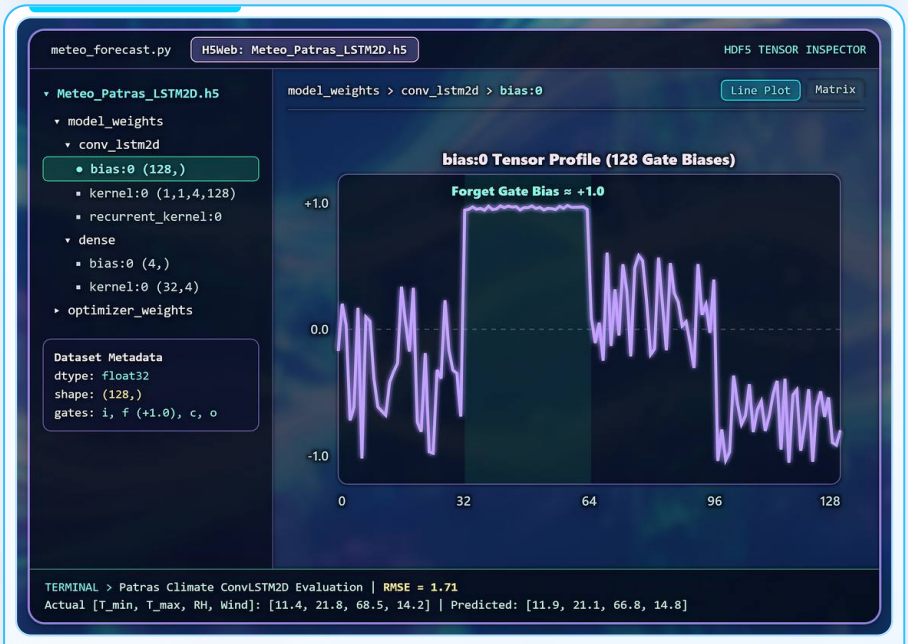


Figure 7.3

H5Web (Visual Studio Code extension): direct numerical and heatmap inspection of synaptic weight matrices and biases inside HDF5 (.h5) files.

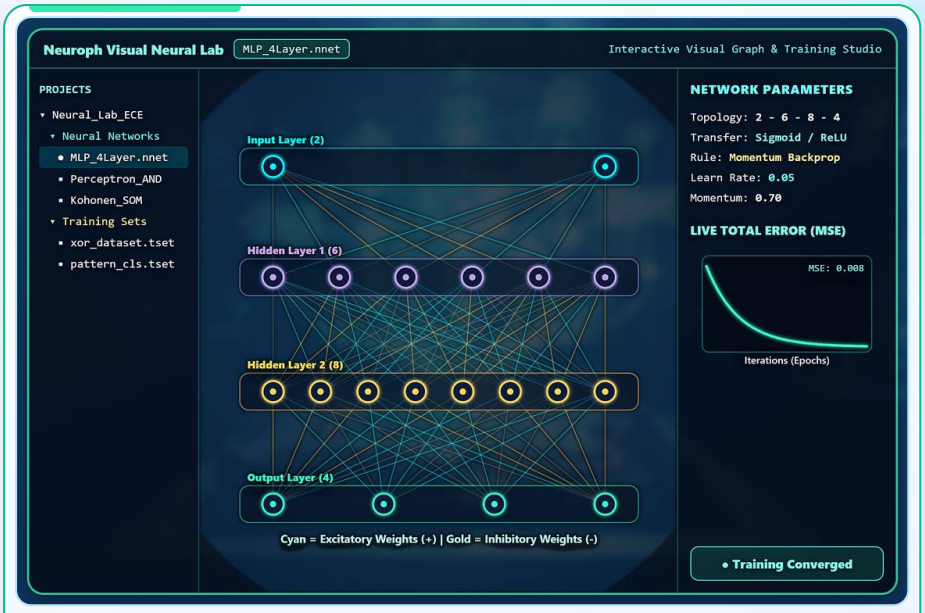


Figure 7.4

Neuroph Studio: educational graphical environment (Java) for hands-on experimentation with compact neural networks and synaptic weights.

STEP-BY-STEP WORKED EXAMPLE

Day-1 Tensor Shape & Debugging Walkthrough (Computing model.summary() by Hand)

Over half of all beginner deep learning bugs are simple **shape mismatches**. Trace a mini-batch of **32 grayscale images** of size 28×28 pixels through a classic digit classifier to verify every output tensor shape and trainable parameter count by hand:

1. **Input Tensor:** Shape `(32, 28, 28, 1)` = *(Batch=32, Height=28, Width=28, Channels=1)*. Forgetting the trailing channel axis `1` is the #1 Day-1 `ValueError` in vision labs!
2. **Conv2D(32 filters, 3×3 kernel, padding='valid'):** Spatial size becomes *(28 − 3) / 1 + 1 = 26* → output shape `(32, 26, 26, 32)`. Trainable weights: each of 32 filters has *3×3×1* weights plus 1 bias → **(3 × 3 × 1 + 1) × 32 = 320 parameters**.
3. **MaxPooling2D(pool_size=2):** Halves spatial dimensions (*26 / 2 = 13*) → output shape `(32, 13, 13, 32)` with **0 parameters** (pure max selection, no learned weights).
4. **Flatten():** Unrolls each 13×13×32 cube into a 1D feature vector of length *13 × 13 × 32 = 5,408* → shape `(32, 5408)`, **0 parameters**.
5. **Dense(10, activation='softmax'):** Maps 5,408 features to 10 class probabilities → shape `(32, 10)` with **(5,408 + 1) × 10 = 54,090 parameters**. Grand total printed by `model.summary()`: **320 + 54,090 = 54,410 trainable parameters**!

◆ Systematic 5-Step Model Debugging & Observability Protocol

1. **Audit Tensors & Scaling First:** Before calling `fit()`, verify exact tensor shapes, dtypes (`float32`), input ranges (`[0, 1]` or standardized), label alignment, and zero NaN/Inf entries.
2. **Single-Batch Overfit Sanity Check:** Train briefly on 4 to 16 examples without regularization; if training loss fails to plunge toward zero, inspect loss-function pairing (e.g., `from_logits` mismatch) or broken gradient wiring before scaling up.
3. **Watch Train vs. Val Curves & Weight Histograms:** Use TensorBoard to spot diverging validation loss (overfitting), stalled flat histograms (vanishing gradients / dying ReLUs), or exploding weight norms early.
4. **Verify Serialized Topology & Weights:** Open exported `.keras` or `.onnx` graphs in Netron and `.h5` weights in H5Web to confirm layer connectivity and non-degenerate parameter distributions.
5. **Isolate Every Ablation Run:** Pin random seeds (`seed=42`) and write each experiment to its own timestamped directory (`run_YYYYMMDD_HHMMSS`) so learning-rate or architecture sweeps overlay cleanly.

7.3 Lab 04: Logging Training Telemetry & Histograms in TensorBoard

LAB 04 **Live Scalar Curves, Graph & Weight Histograms in TensorBoard** `code/04_and_gate_tensorboard.py`

Attaches `keras.callbacks.TensorBoard(log_dir=..., histogram_freq=10, write_graph=True)` to a Keras 3 network using unique timestamped run folders (`logs/gate_and/run_YYYYMMDD_HHMMSS`), recording epoch-by-epoch loss, accuracy, computational graph, and weight distribution evolution.

Guided Experiment: Run Lab 04 twice: once with `learning_rate=0.04` and once with `learning_rate=0.002`. Launch TensorBoard in VS Code (`Ctrl+Shift+P → Python: Launch TensorBoard`), overlay both runs on the Scalars tab, and inspect how hidden-layer weight histograms widen as the decision boundary sharpens.

PYTHON 3.12+ `code/04_and_gate_tensorboard.py` Timestamped TensorBoard Callback & Weight Histograms in Keras 3

```python
from datetime import datetime
from pathlib import Path
import numpy as np
import keras
from keras import layers


def main() -> None:
    keras.utils.set_random_seed(42)

    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float32)
    y = np.array([[0], [0], [0], [1]], dtype=np.float32)

    model = keras.Sequential(
        [
            keras.Input(shape=(2,), name="inputs"),
            layers.Dense(4, activation="relu", name="dense_hidden"),
            layers.Dense(1, activation="sigmoid", name="dense_output"),
        ],
        name="and_gate_tensorboard",
    )

    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=0.04),
        loss="binary_crossentropy",
        metrics=["accuracy"],
    )

    # Create a unique timestamped log folder to compare multiple runs
# ... (full executable code in code/04_and_gate_tensorboard.py inside code.zip)
```

CHAPTER 8

GPU Compute Acceleration, CUDA & Development Environment

SIMD/SIMT warps, Tensor Core systolic arrays, the Roofline Model, FP32 to FP8 numerical precision, Automatic Mixed Precision (AMP), and Labs 11 & 12.

Learning Objectives & Theoretical Scope

- Contrast latency-optimized CPU cores against throughput-optimized GPU **SIMT warps** and **Tensor Core systolic arrays**.
- Apply the **Roofline Model** (Arithmetic Intensity vs. Hardware Ridge Point) to diagnose Compute-Bound versus Memory-Bandwidth-Bound deep learning kernels.
- Compare IEEE and AI floating-point representations (**FP32, TF32, BF16, FP16, FP8**) and explain the 6-step **Automatic Mixed Precision (AMP)** loop with FP32 Master Weights and Dynamic Loss Scaling.
- Configure clean Python 3.12+ virtual environments and benchmark CPU vs. GPU execution across PyTorch 2 and Keras 3 / TensorFlow in **Labs 11 and 12**.

8.1 CPU vs. GPU Architecture: SIMT Warps, Tensor Cores & the Roofline Model

Deep learning workloads are overwhelmingly dominated by dense General Matrix Multiplications (**GEMM**) and tensor convolutions. A modern server **CPU** dedicates most of its silicon budget to large caches, branch predictors, and out-of-order execution for a modest number of high-frequency cores. On our seminar laboratory workstation, dual **Intel Xeon E5-2600 v3** processors supply 20 physical cores, 40 hardware threads, and 32 GB of DDR4 memory (~128 GB/s host bandwidth).

FIRST PRINCIPLES & INTUITION **Sports Cars vs. Fleet of Cargo Vans: Why GPUs Accelerate Neural Networks**

Suppose you need to haul 10,000 identical boxes across a city. A **CPU** is like a small team of 20 ultra-fast Formula 1 sports cars: unbeatable at complex, unpredictable city routing (running your operating system, spreadsheets, or complex branching logic), yet forced to make thousands of trips when moving bulk cargo. A **GPU** is like a synchronized fleet of **3,584 cargo vans** moving down parallel highway lanes: each individual core runs slightly slower than a CPU core, yet multiplying millions of uniform neural weights simultaneously finishes in a fraction of the time!

- **CUDA:** NVIDIA's parallel computing platform that lets Python libraries (PyTorch, TensorFlow) dispatch matrix math directly onto thousands of GPU cores.
- **VRAM (Video RAM):** Ultra-fast memory wired directly beside the GPU chip (e.g., 12 GB on an RTX 3060) that holds model weights, gradients, and mini-batches.
- **FLOPs & Numerical Precision:** *FLOPs* count floating-point multiply/add steps; lowering storage precision from 32 bits (**FP32**) to 16 bits (**FP16/BF16**) cuts memory traffic in half while keeping neural accuracy intact.

A **GPU** flips this silicon trade-off completely: it packs thousands of lightweight arithmetic cores organized into Streaming Multiprocessors (SMs) executing in **SIMT (Single Instruction, Multiple Threads)** lockstep groups called **warps** (32 threads per warp on NVIDIA CUDA). Whenever one warp stalls waiting for VRAM data, the hardware warp scheduler instantaneously switches to another resident warp with zero context-switch penalty, hiding memory latency through massive concurrency. Modern RTX and datacenter GPUs further embed specialized **Tensor Core systolic arrays** that fuse an entire matrix tile multiply-accumulate ($D = A \times B + C$) per clock cycle with on-chip register reuse, achieving over **100×** the raw matrix TFLOPS of a dual-socket CPU! Even our compact workstation **NVIDIA GeForce GTX 1050 Ti** (768 CUDA Cores, 4 GB GDDR5) delivers a verified **5.4×** epoch speedup on MNIST (11.64 s vs. 63.18 s at equivalent ~96% accuracy).

$$I = \text{FLOPs} / \text{Bytes}_{\text{transferred}} \quad \Rightarrow \quad P_{att} = \min(P_{peak}, B_{mem} \cdot I)$$

The Roofline Model (Williams et al., 2009): kernels whose Arithmetic Intensity I exceeds the hardware ridge point P_{peak} / B_{mem} are Compute-Bound (large GEMM / Conv2D); kernels below it are Memory-Bandwidth-Bound (element-wise activations, LayerNorm, batch-1 LLM decoding).

Intuitively, **Arithmetic Intensity (*I*)** measures how much useful math we squeeze out of every byte carried from memory (much like cooking a full dinner per grocery trip rather than driving back to the market for every single ingredient). Understanding the **Roofline Model** immediately clarifies modern AI systems engineering: dense matrix training with large batch sizes sits in the **Compute-Bound** regime (accelerated by Tensor Cores and lower-bit arithmetic), whereas single-user LLM token generation reloads gigabytes of weights per generated token and sits deeply in the **Memory-Bandwidth-Bound** regime (accelerated by 4-bit weight quantization in Chapter 10 and operator fusion such as FlashAttention and `torch.compile`).

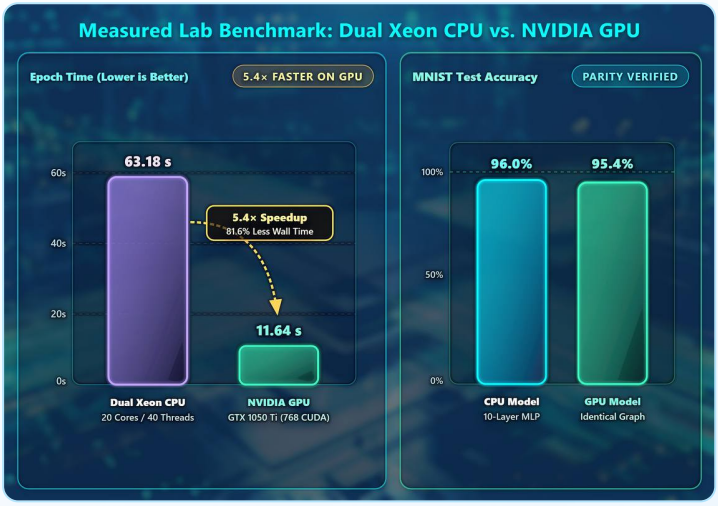


Figure 8.1

Experimental seminar epoch-time benchmark (63.18 s on Dual Xeon CPU vs. 11.64 s on NVIDIA GTX 1050 Ti GPU) confirming identical classification accuracy.

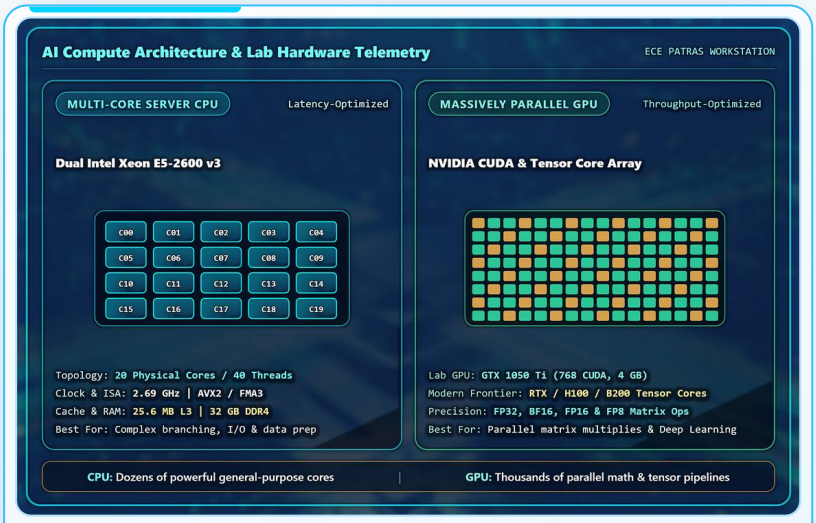


Figure 8.2

Laboratory workstation hardware telemetry: Dual Xeon CPU topology (20C/40T) contrasted with GPU CUDA & Tensor Core array architecture.

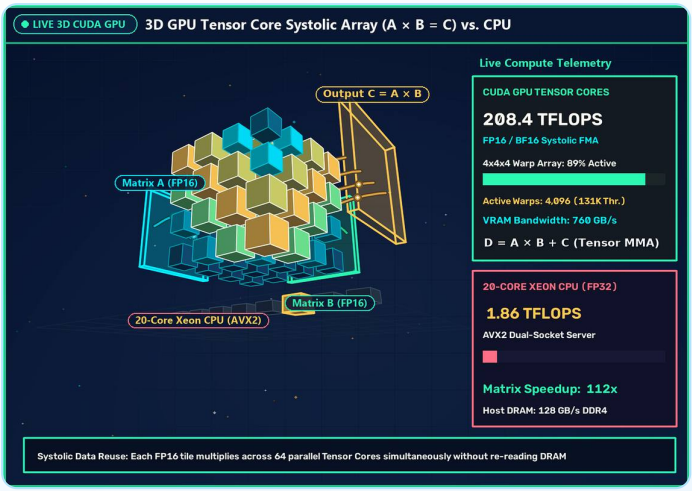


Figure 8.3

3D GPU Tensor Core Systolic Array (D = A x B + C) illustrating on-chip FP16 tile reuse across parallel warps versus host DDR4 AVX2 CPU throughput.

8.2 Numerical Precision Formats & Automatic Mixed Precision (AMP)

Deep neural networks tolerate reduced mantissa precision remarkably well during forward and backward matrix multiplications, provided dynamic exponent range is safeguarded against overflow and gradient underflow. The table below compares all five IEEE and AI floating-point formats across modern GPU architectures:

Format	Bits (S : E : M)	Max Value	Relative Speed	Deep Learning Engineering Role
FP32	32 (1 : 8 : 23)	3.4×10^{38}	1.0×	Master weights, AdamW moments, loss accumulation & numerical baseline
TF32	19 (1 : 8 : 10)	3.4×10^{38}	~2.0×	Default Tensor Core matmul format on Ampere/Ada/Hopper (FP32 range, FP16 precision)
BF16	16 (1 : 8 : 7)	3.4×10^{38}	2× to 4×	Modern LLM training gold standard; matches FP32 exponent so **no Loss Scaling** is needed
FP16	16 (1 : 5 : 10)	65,504	2× to 4×	Higher mantissa precision than BF16, but narrow range requires **Dynamic Loss Scaling**
FP8 (E4M3)	8 (1 : 4 : 3)	448.0	4× to 6×	Hopper/Blackwell ultra-fast inference & block-scaled Transformer training

Automatic Mixed Precision (AMP) works like keeping an archival high-precision master ledger while doing fast everyday arithmetic on a scratchpad: it delivers 2× to 4× throughput gains and up to 50% activation VRAM savings without degrading convergence accuracy through a disciplined six-stage training loop: (1) store authoritative **FP32 Master Weights** so tiny parameter updates never round to zero; (2) autocast activations and weight copies into **FP16 or BF16**, halving HBM memory traffic; (3) execute Tensor Core GEMM with **FP32 internal accumulation**; (4) compute the scalar loss in FP32 and, when using FP16, multiply by a dynamic scale factor S ($L_{scaled} = L \cdot S$) so tiny backpropagated gradients stay safely above the FP16 underflow floor; (5) backpropagate scaled gradients; and (6) unscale ($/ S$), check for Inf/NaN (skipping the step and halving S if overflow occurred), clip norms, and apply the exact AdamW update back to the FP32 Master Weights.

STEP-BY-STEP WORKED EXAMPLE

Parallel Compute & VRAM Footprint by Hand: CPU vs. RTX 3060 & FP32 vs. AMP

Use simple arithmetic to see why modern GPUs and Mixed Precision transform deep learning budgets:

1. **Parallelism on 1,000,000 Multiply-Adds (2,000,000 FLOPs):** On a **20-core CPU** processing one scalar element per core per wave, the task requires *1,000,000 / 20* = ***50,000 waves***. On an NVIDIA **RTX 3060 GPU (3,584 CUDA cores)**, the exact same workload divides into just $\lceil 1{,}000{,}000 / 3{,}584 \rceil$ = ***280 parallel waves*** (~**179×** more concurrent execution lanes, further accelerated when Tensor Cores fuse 4×4 matrix tiles per clock!).
2. **VRAM Footprint of a 10M-Parameter Model in Pure FP32 (4 bytes/number):** Model weights occupy *10,000,000 × 4 B* = ***40 MB***. During training with AdamW, storing FP32 gradients (*40 MB*), first-moment moving averages m_t (*40 MB*), second-moment averages v_t (*40 MB*), and optimizer working buffers (*40 MB*) adds **160 MB of gradient & AdamW state** (*200 MB* static footprint before batch activations).
3. **Savings with FP16 / AMP (2 bytes/number):** For inference, FP16 cuts weights in half to just **20 MB**. During AMP training, forward working weights (*20 MB*), FP16 gradients (*20 MB*), and **mini-batch activation tensors shrink by 50%**, letting dramatically larger batches fit inside a 12 GB RTX 3060 at 2× to 4× higher Tensor Core speed!

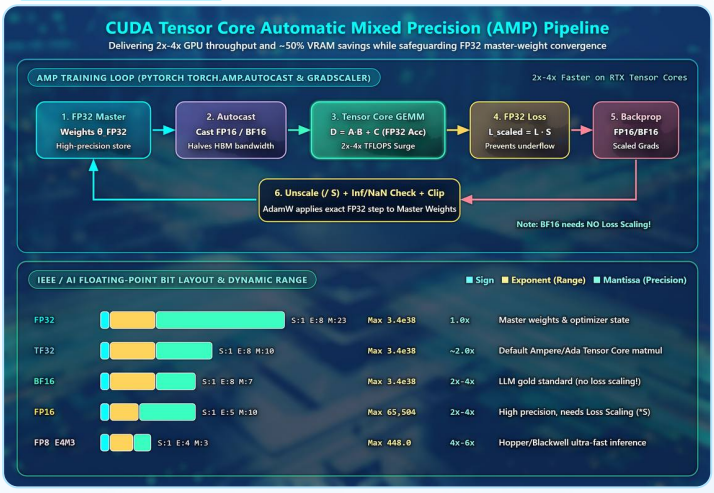


Figure 8.4

CUDA Tensor Core Automatic Mixed Precision (AMP) 6-stage training loop (FP32 master weights, autocast, dynamic loss scaling) and IEEE/AI floating-point bit layouts from FP32 to FP8.

8.3 Clean Python Environment Setup & GPU Best Practices

For stable, reproducible deep learning engineering in **Python 3.12+**, we enforce four practical workstation rules:

1. **Isolated virtual environments:** Create `python -m venv .venv` (or `uv venv`) and install pinned packages from `requirements.txt` so projects never collide with system libraries.
2. **Driver & CUDA alignment:** On Windows, PyTorch 2 provides native CUDA wheels out of the box, whereas GPU acceleration in TensorFlow is best deployed inside Linux or WSL2 with NVIDIA CUDA libraries.
3. **Dynamic VRAM allocation:** In TensorFlow / Keras 3, call `tf.config.experimental.set_memory_growth(gpu, True)` so the runtime allocates VRAM incrementally rather than locking 100% of card memory on startup.
4. **Accurate asynchronous GPU timing:** Because CUDA kernel launches return immediately to the host CPU, always call `torch.cuda.synchronize()` before reading wall-clock timers in performance benchmarks.

8.4 Labs 11 & 12: CPU vs. GPU Benchmarks in PyTorch 2 & Keras 3

LAB 11 **PyTorch 2 CPU vs. CUDA GPU Benchmark with AMP & Synchronization**

`code/11_gpu_vs_cpu_pytorch.py`

Trains an identical seeded two-stage `MnistCNN` first on CPU and immediately afterward on CUDA GPU using `torch.amp.autocast('cuda')`, `GradScaler`, and strict `torch.cuda.synchronize()` barriers before computing the exact wall-clock speedup ratio and saving comparative loss/accuracy charts.

Guided Experiment: Compare execution time with `batch_size=32` versus `batch_size=256` on GPU: notice how larger batches raise Arithmetic Intensity and warp occupancy, pushing the workload further up the Roofline curve.

PYTHON 3.12+ `code/11_gpu_vs_cpu_pytorch.py` PyTorch 2 MnistCNN & CUDA AMP Benchmark Setup

```python
import time
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms


class MnistCNN(nn.Module):
    """Two-stage CNN classifying 28x28 grayscale digits into 10 classes."""

    def __init__(self) -> None:
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
# ... (full executable code in code/11_gpu_vs_cpu_pytorch.py inside code.zip)
```

LAB 12 **Keras 3 / TensorFlow Dynamic VRAM Growth & Explicit Device Placement**

`code/12_gpu_vs_cpu_keras.py`

Demonstrates safe TensorFlow GPU initialization via `set_memory_growth(gpu, True)` and explicit hardware placement using `with tf.device('/CPU:0')` versus `with tf.device('/GPU:0')` over a seeded MNIST CNN, exporting a comparative timing bar chart (`keras_cpu_vs_gpu_benchmark.png`).

Guided Experiment: Run Lab 12 on native Windows (observing CPU fallback notice) and, if available, inside WSL2 or Linux with CUDA enabled to measure end-to-end Keras 3 GPU acceleration directly.

PYTHON 3.12+ `code/12_gpu_vs_cpu_keras.py` Dynamic VRAM Memory Growth & Explicit tf.device Benchmark

```python
import time
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import keras
from keras import layers


def configure_gpu_memory_growth() -> list:
    """Enables incremental VRAM allocation on all available physical GPUs."""
    gpus = tf.config.list_physical_devices("GPU")
    for gpu in gpus:
        # Prevent TensorFlow from claiming all VRAM immediately at startup
        tf.config.experimental.set_memory_growth(gpu, True)
    return gpus

# ... (full executable code in code/12_gpu_vs_cpu_keras.py inside code.zip)
```

CHAPTER 9

Internal Architecture of Large Language Models (LLMs)

From BPE subword tokenization, Rotary Positional Embeddings (RoPE), and KV-Cache VRAM engineering to Temperature, Top-P, Min-P sampling, and Test-Time Compute reasoning.

Learning Objectives & Theoretical Scope

- Trace the autoregressive next-token generation loop and construct subword vocabularies step by step using the greedy **Byte-Pair Encoding (BPE)** merge algorithm.
- Derive the 2D complex rotation geometry of **Rotary Positional Embeddings (RoPE)** and verify why inner products depend strictly on relative token distance $(m - n)\theta_i$.
- Calculate exact GPU memory requirements using the **KV-Cache VRAM byte formula** and compare **MHA**, **GQA**, and **MQA** attention head-sharing schemes across 8K to 128K contexts.
- Contrast deterministic and stochastic decoding strategies (Greedy, Beam Search, Temperature, Top-K, Top-P Nucleus, Min-P, and **Test-Time Compute**) through Shannon entropy.
- Implement a BPE tokenizer, probability sampler, causal **Mini-GPT** decoder, and KV-Cache estimator from scratch in **Labs 23 & 24**.

9.1 Autoregressive Factorization & Byte-Pair Encoding (BPE)

FIRST PRINCIPLES & INTUITION **How a Language Model Reads & Writes One Subword at a Time**

At its heart, a Large Language Model (LLM) plays one repeating game: *given every word written so far, predict the single most plausible next puzzle piece (subword token)*. That loop operates in two distinct phases that mirror how a student takes a written exam:

- **Prefill (Reading Phase):** Like scanning an entire prompt in one glance, the GPU processes every input token simultaneously in parallel and saves compact attention notes (the **KV-Cache**) inside high-speed GPU memory (**VRAM**).
- **Decode (Writing Phase):** Like writing an essay word by word, the model predicts one subword, appends it to the page, reuses its saved KV-Cache notes instead of re-reading from scratch, and repeats until emitting a stop token.

Plain-English Glossary: A *token* is a word or frequent fragment (roughly three-quarters of an English word); a *logit* is a raw numerical preference score before conversion into probabilities; and the *context window* is the model's total working desk capacity measured in tokens.

Having examined the **Transformer** architecture (Chapter 5) and the parallel throughput of **GPUs & CUDA** (Chapter 8), we now see how both pillars converge inside modern **Large Language Models (LLMs)**. Virtually every frontier decoder-only model (Gemma, Llama, Qwen, DeepSeek, Claude, and Gemini) models language through *autoregressive factorization*: the joint probability of a sequence of tokens $(x_1, x_2, \ldots, x_N)$ is decomposed via the chain rule of probability into a product of conditional next-token distributions:

$$P(x_1, x_2, \ldots, x_N) = \prod_{t=1}^{N} P_\theta(x_t \mid x_1, x_2, \ldots, x_{t-1})$$

At every decoding step t, the Transformer maps the preceding context $x_{<t}$ to a probability distribution over the entire vocabulary V, samples one token x_t, and appends it to the input window.

Before any neural layer executes, raw text must be mapped into integer token IDs. Word-level dictionaries fail on unseen words (Out-of-Vocabulary errors) and explode across morphologically rich languages, whereas raw character sequences triple or quadruple context length. Modern LLMs solve this trade-off with **Byte-Pair Encoding (BPE)**, a greedy compression algorithm that builds a subword vocabulary of 32,000 to 256,000 units:

1. **Initialize Base Symbols:** Split every training word into UTF-8 bytes or characters and append an end-of-word boundary marker `</w>` (for example, `n e u r a l </w>` and `n e u r o n </w>`).
2. **Count Adjacent Bigrams:** Tally the corpus-weighted frequency of every adjacent symbol pair `(a, b)` across all distinct words.
3. **Greedy Merge:** Select the single most frequent pair `(a*, b*)`, fuse every occurrence into a new symbol `a*b*` (for example, `('n','e') → 'ne'`, then `('ne','u') → 'neu'`, then `('neu','r') → 'neur'`), and record the ordered merge rule.
4. **Iterate to Target Vocabulary Size:** Repeat steps 2 and 3 for M merges until reaching the target vocabulary size `|V|`. Common words survive as single tokens, while rare scientific terms, Greek words, equations, and Python identifiers decompose cleanly into reusable roots and suffixes with zero OOV failures.

9.2 Long-Context Geometry: Rotary Positional Embeddings (RoPE)

Because Self-Attention treats input tokens like an unordered bag of words by default (*permutation-equivariant*), a language model needs an explicit encoding of token order. Early Transformers added fixed sinusoidal or learned absolute vectors to input embeddings, which degraded sharply when extrapolated beyond the training context length. Modern open and frontier LLMs instead adopt **Rotary Positional Embeddings (RoPE, Su et al.)**. Intuitively, RoPE attaches a bank of spinning clock hands to every token's vector so the angular gap between two hands depends strictly on how many positions apart the two words sit. Formally, RoPE groups each head's d_{head} coordinates into $d_{head} / 2$ consecutive 2D planes and rotates each pair (x_{2i}, x_{2i+1}) of the Query vector at position m and Key vector at position n by position-proportional angles $m\theta_i$ and $n\theta_i$:

$$\mathbf{R}(m, \theta_i)\,[x_{2i}, x_{2i+1}]^{T} = [x_{2i}\cos(m\theta_i) - x_{2i+1}\sin(m\theta_i),\;\; x_{2i}\sin(m\theta_i) + x_{2i+1}\cos(m\theta_i)]^{T}$$

$$\theta_i = \text{base}^{-2i\,/\,d_{head}} \;\;|\;\; (R_m\,q)^{T}\,(R_n\,k) = q^{T}\,R_{(n-m)}\,k$$

By the orthogonality of 2D rotation matrices, the Query-Key inner product depends strictly on the relative angular displacement $(m - n)\theta_i$. High-frequency dimensions (small i) resolve local syntax, while low-frequency dimensions track long-range discourse across 128K to 1M tokens (extended via NTK or YaRN frequency scaling).

9.3 Causal Masking, Exact KV-Cache VRAM Math & MHA vs. GQA vs. MQA

LLM inference divides cleanly into two distinct computational regimes. During **Prefill**, all prompt tokens are processed simultaneously through dense matrix-matrix multiplications, protected by a lower-triangular **causal mask** (setting upper-triangular entries $j > i$ to $-\infty$ before Softmax so no token can attend to the future). During **Autoregressive Decode**, tokens arrive one at a time. Recomputing Keys and Values for all previous tokens at every step would incur quadratic `O(S²)` work per generated token. By caching previously computed K and V tensors inside GPU VRAM (the **KV-Cache**), per-step compute collapses to linear `O(S)`, shifting the primary engineering constraint from FLOPs to GPU memory capacity and bandwidth:

$$\mathbf{M_{KV} = 2 \cdot L \cdot n_{kv} \cdot d_{head} \cdot S \cdot B \cdot bytes}$$

Factor 2 accounts for both Key and Value tensors across L Transformer layers, n_{kv} KV heads, head width d_{head}, sequence length S, batch size B, and numeric precision (2 bytes for FP16/BF16, 1 byte for FP8).

Worked Numerical Example (8B Parameter Model, Single User `B = 1`, FP16 = 2 bytes): Consider a standard 8B architecture with $L = 32$ layers, $n_q = 32$ query heads, and $d_{head} = 128$. With **Grouped-Query Attention (GQA, `n_kv = 8`, 4:1 sharing)**, an **8K context** (`S = 8,192`) requires `2 × 32 × 8 × 128 × 8,192 × 1 × 2 = 1,073,741,824 bytes = 1.00 GiB` (1.07 GB). Scaling up to **32K context** (`S = 32,768`) needs **4.00 GiB** (4.29 GB), fitting comfortably inside a 12 GB consumer GPU alongside 4-bit weights. Pushing to a massive **128K context** (`S = 131,072`) expands the KV-Cache

to **16.00 GiB** (17.18 GB), exceeding a 12 GB GPU unless FP8 KV-Cache quantization halves it to 8.00 GiB. By contrast, classical **Multi-Head Attention (MHA, $n_{kv} = 32$)** multiplies every figure by 4×, consuming 16.00 GiB already at 32K and a prohibitive **64.00 GiB** at 128K!

Attention Sharing Scheme	Query : KV Ratio	KV-Cache (8B, 32K, FP16)	Decode Bandwidth Cost	Accuracy & Frontier Adoption
MHA (Multi-Head)	1 : 1 (32 Q, 32 KV)	16.00 GiB (OOM on 12 GB)	Baseline (1.0×)	Highest expressive capacity; used in GPT-3 & Llama 2 7B
GQA (Grouped-Query)	4 : 1 to 8 : 1 (e.g., 32 Q, 8 KV)	4.00 GiB (fits 12 GB GPU!)	4× to 8× reduction	Retains ~99.5% of MHA quality; standard in Llama 3, Gemma 3 & Qwen
MQA (Multi-Query)	N : 1 (all Q share 1 KV)	0.50 GiB (maximum shrink)	Up to 32× reduction	Ultra-fast serving; slight quality drop on complex long-range reasoning
MLA (Multi-Head Latent)	Low-rank compressed KV latent	<1.0 GiB (compressed cache)	8× to 16× reduction	Projects K and V into a shared compressed vector (DeepSeek-V2/V3/R1)

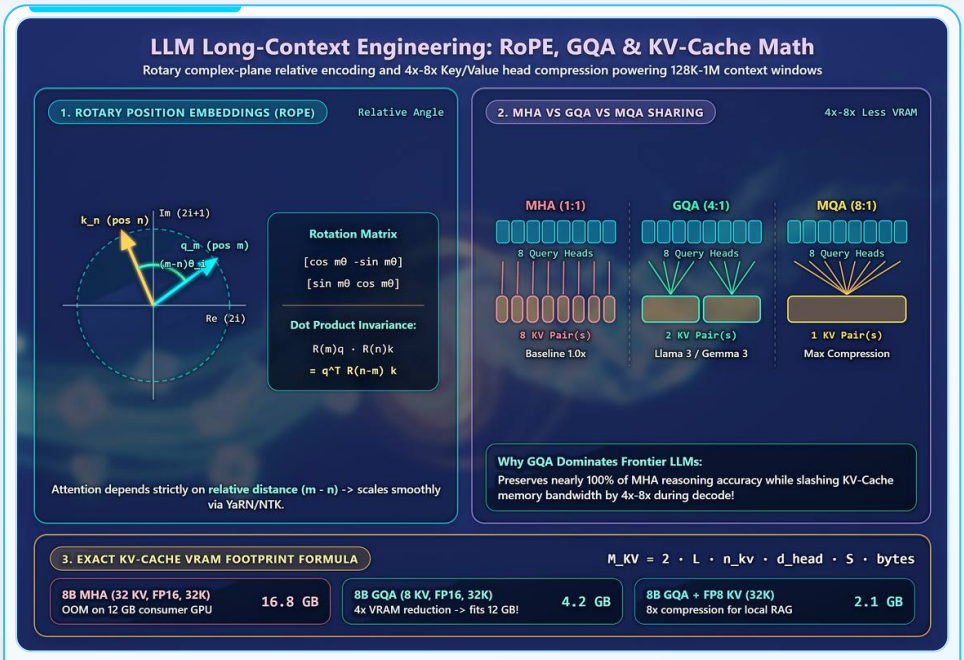


Figure 9.1

Long-context LLM engineering: 2D complex-plane RoPE rotation invariance (m − n)θ_i, MHA vs. GQA vs. MQA Key/Value head sharing, and exact KV-Cache VRAM budgets for an 8B model across FP16 and FP8.

9.4 Probabilistic Decoding Strategies, Shannon Entropy & Test-Time Compute

At its unembedding projection, the Transformer outputs a vector of raw scores (*logits*) z_i across all $|V|$ candidate tokens. **Temperature (T)** rescales these logits prior to Softmax, controlling the sharpness and **Shannon Entropy *H(P)*** (measured in bits of uncertainty) of the next-token distribution:

$$P(w_i) = \exp(z_i / T) / \Sigma_j \exp(z_j / T) \quad | \quad H(P) = -\Sigma_i P(w_i) \log_2 P(w_i)$$

As T → 0, probability mass concentrates on the top logit (H → 0 bits, deterministic code/math). As T increases (0.8 to 1.2), entropy rises and diverse continuations emerge.

Production inference engines combine temperature scaling with structured truncation and test-time deliberation strategies tailored to each task:

- **Greedy Decoding (`argmax`):** Always selects the highest-probability token. Fast and reproducible, yet prone to repetitive loops and suboptimal global phrasing.
- **Beam Search (`k` beams):** Maintains the top *k* highest cumulative log-probability hypotheses in parallel. Effective for machine translation and constrained extraction, though overly generic for open-ended reasoning.
- **Top-K Truncation:** Zeroes out all tokens outside the *K* most likely candidates before renormalizing. Simple, but inflexible: a fixed *K* prunes valid synonyms on flat distributions while admitting noisy tail tokens on sharp distributions.
- **Top-P (Nucleus) Sampling:** Dynamically retains the smallest sorted set of tokens whose cumulative probability reaches threshold *P* (typically 0.85 to 0.95), automatically expanding when the model is uncertain and shrinking when one answer dominates.
- **Min-P Sampling:** Keeps every candidate token satisfying $p_i \geq p_{min} \cdot p_{max}$ (scaled relative to the single top token's probability). Even at elevated temperatures (`T = 1.0` to `1.3`), Min-P cleanly blocks absurd low-probability tokens whenever the model has a high-confidence peak.
- **Test-Time Compute (Reasoning Models):** Systems such as DeepSeek-R1 and Gemini Thinking scale intelligence at inference time rather than exclusively during pretraining, generating private internal reasoning chains (*thinking tokens*) to verify equations, backtrack from contradictions, and validate code before emitting the final response.

STEP-BY-STEP WORKED EXAMPLE

Hand-Computing Softmax Temperature & Top-P (P = 0.90) on 3 Candidate Words

Suppose the prompt ends with "*Under the quiet sky, we watched the ...*" and the model outputs three raw logits: `"stars": 4.0`, `"night": 3.0`, and `"ocean": 1.0`. Subtracting the maximum logit (`4.0`) keeps Softmax probabilities identical while simplifying hand arithmetic to gaps `[0.0, -1.0, -3.0]`:

1. **Cool / Focused (`T = 0.5`):** Dividing by `0.5` sharpens the gaps to `[0.0, -2.0, -6.0]`. Exponentiating yields `[1.000, 0.135, 0.002]` (sum `1.138`), producing **"stars" = 87.9% (~88%)**, **"night" = 11.9%**, and **"ocean" = 0.2%** (tight Shannon entropy `H = 0.55 bits`).
2. **Warm / Creative (`T = 1.5`):** Dividing by `1.5` flattens the gaps to `[0.0, -0.667, -2.0]`. Exponentiating gives `[1.000, 0.513, 0.135]` (sum `1.649`), spreading mass to **"stars" = 60.7% (~60%)**, **"night" = 31.1% (~31%)**, and **"ocean" = 8.2% (~8%)** (higher entropy `H = 1.26 bits`).
3. **Top-P (`P = 0.90`) Truncation by Hand:** Sort descending and accumulate until reaching `90%`. At `T = 0.5`, "stars" alone is `87.9% < 90%`, so we keep {"stars", "night"} (`99.8%`) and prune "ocean". At `T = 1.5`, "stars" + "night" = `91.8% ≥ 90%`, again pruning "ocean" (`8.2%`) cleanly; renormalizing the kept nucleus yields `66.1%` for "stars" and `33.9%` for "night".

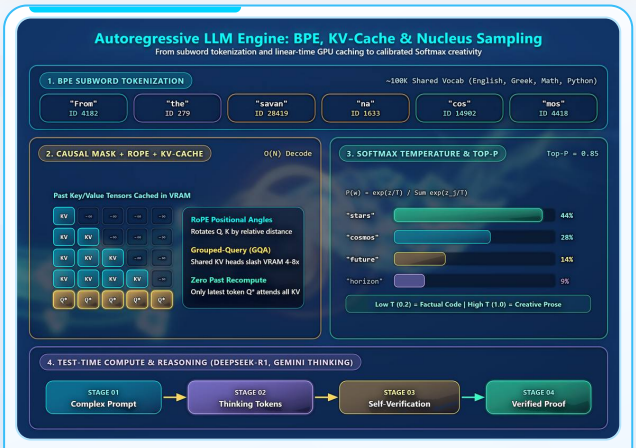


Figure 9.2

End-to-end LLM inference engine: BPE subword tokenization, causal self-attention with RoPE & VRAM KV-Cache, Softmax sampling (Temperature & Top-P), and the Test-Time Compute reasoning pipeline.

9.5 Labs 23 & 24: BPE, Sampling & a Mini-GPT Decoder from Scratch

LAB 23 **BPE Subword Trainer, Temperature/Top-K/Top-P Sampler & Shannon Entropy**

`code/23_bpe_tokenizer_and_sampling.py`

Implements the full greedy `train_bpe()` merge loop and `encode_bpe()` subword segmenter from scratch in pure Python, followed by a numerically stabilized Softmax sampler (`sample_next_token()`) that compares deterministic (`T=0.25`), balanced (`T=0.70, Top-P=0.85`), and creative (`T=1.25, Top-K=5`) decoding while reporting exact Shannon entropy in bits.

Guided Experiment: Increase `num_merges` from 18 to 40 in `train_bpe()` and inspect how the encoded token count of the prompt drops. Then add a 3-line `min_p = 0.08` filter (`probs[probs < min_p * probs.max()] = 0`) and observe how tail entropy tightens even at `T = 1.25`.

PYTHON 3.12+ `code/23_bpe_tokenizer_and_sampling.py` Lab 23: BPE subwords, Softmax scaling with Temperature/Top-P & Shannon entropy

```python
from collections import Counter
import math
import numpy as np


def train_bpe(corpus: list[str], num_merges: int = 18):
    """Learns subword merge rules by fusing frequent adjacent symbol pairs."""
    vocab = Counter()
    for sentence in corpus:
        for word in sentence.lower().split():
            vocab[" ".join(list(word)) + " </w>"] += 1

    merges = []
    for _ in range(num_merges):
        pairs = Counter()
        for word, freq in vocab.items():
            syms = word.split()
            for i in range(len(syms) - 1):
                pairs[(syms[i], syms[i + 1])] += freq
        if not pairs:
            break
        best = max(pairs, key=pairs.get)
        merges.append(best)
        bigram = " ".join(best)
        fused = "".join(best)
        vocab = Counter(
            {w.replace(bigram, fused): f for w, f in vocab.items()}
        )
    return merges


def encode_bpe(text: str, merges: list[tuple[str, str]]) -> list[str]:
    tokens = []
    for word in text.lower().split():
        syms = list(word) + ["</w>"]
        for a, b in merges:
            i = 0
            while i < len(syms) - 1:
                if syms[i] == a and syms[i + 1] == b:
                    syms[i : i + 2] = [a + b]
# ... (full executable code in code/23_bpe_tokenizer_and_sampling.py inside code.zip)
```

LAB 24 **Causal Self-Attention Mini-GPT Decoder & Exact KV-Cache VRAM Calculator**

`code/24_mini_gpt_from_scratch.py`

Builds a complete character-level causal decoder Transformer (with token + positional embeddings, `torch.tril` lower-triangular causal masking, LayerNorm, GELU feed-forward network, and automatic pure-NumPy fallback) alongside `kv_cache_memory_gb()` implementing the exact GQA VRAM equation for 8K and 32K context windows.

Guided Experiment: Call `kv_cache_memory_gb()` with `n_kv_heads=32` (MHA) vs. `n_kv_heads=8` (GQA) and `bytes_per_elem=1` (FP8) at `seq_len=131072` to reproduce the 64 GiB vs. 16 GiB vs. 8 GiB memory scaling table experimentally.

PYTHON 3.12+ `code/24_mini_gpt_from_scratch.py` Lab 24: Mini-GPT causal self-attention & exact KV-Cache VRAM calculation

```python
import math
import numpy as np

CORPUS = (
    "deep learning turns raw data into structured understanding. "
    "transformers connect every token through causal self attention. "
    "open source models bring transparent intelligence to every lab. "
)


def kv_cache_memory_gb(
    n_layers=32, n_kv_heads=8, head_dim=128, seq_len=8192, bytes_per_elem=2
):
    """Computes exact GPU VRAM used by KV-Cache for one sequence (GQA)."""
    total_bytes = (
        2 * n_layers * n_kv_heads * head_dim * seq_len * bytes_per_elem
    )
    return total_bytes / (1024 ** 3)


def run_numpy_mini_gpt():
    """Zero-dependency pure-NumPy Causal Self-Attention Transformer demo."""
    rng = np.random.default_rng(42)
    chars = sorted(set(CORPUS))
    stoi = {c: i for i, c in enumerate(chars)}
    itos = {i: c for c, i in stoi.items()}
    V, D, T = len(chars), 24, 16

    W_tok = rng.normal(0, 0.2, (V, D))
    W_pos = rng.normal(0, 0.1, (T, D))
    Wq = rng.normal(0, 0.2, (D, D))
    Wk = rng.normal(0, 0.2, (D, D))
    Wv = rng.normal(0, 0.2, (D, D))
    W_head = rng.normal(0, 0.2, (D, V))

    # Empirical transition bias so untrained fallback produces readable words
    trans = np.zeros((V, V), dtype=np.float64)
    for a, b in zip(CORPUS[:-1], CORPUS[1:]):
        trans[stoi[a], stoi[b]] += 2.5

# ... (full executable code in code/24_mini_gpt_from_scratch.py inside code.zip)
```

◆ Practical LLM Decoding & KV-Cache Engineering Checklist

1. Deterministic Code & Proofs: Set `temperature = 0.1 to 0.25` and `top_p = 0.85` (or `min_p = 0.05`) to keep logits sharp and mathematically faithful. **2. Long-Context Memory Sizing:** Always budget GPU VRAM for both static model weights and the growing KV-Cache ($2 \times L \times n_{kv} \times d_{head} \times S \times B \times bytes$), verifying Grouped-Query Attention (GQA) compression and FP8 cache support before deploying 32K+ token windows.

CHAPTER 10

The Open-Source AI Revolution & Weight Quantization

Open-weight ecosystems (Gemma 3, Llama, Qwen, DeepSeek, Mistral), PTQ vs. QAT, Group-Wise INT4 & NF4 mathematics, outlier channels, and memory-bandwidth-bound local inference.

Learning Objectives & Theoretical Scope

- Evaluate architectural, privacy, cost, and scientific reproducibility trade-offs between proprietary cloud APIs and local **Open-Weight LLMs** (`safetensors` & `GGUF`).
- Differentiate **PTQ** vs. **QAT** and derive Symmetric, Asymmetric, **Group-Wise (g=64/128)**, and information-theoretically optimal **NormalFloat4 (NF4)** quantile quantization.
- Explain how emergent **outlier channels** degrade naive low-bit compression and how `LLM.int8()`, **GPTQ**, **AWQ**, and **GGUF Q4_K_M** preserve >99% model quality.
- Analyze why autoregressive batch-1 token decoding is strictly **memory-bandwidth bound** (`tok/s ≈ BW / Model_Bytes`) and calculate real-world throughput on a 12 GB GPU.
- Benchmark INT8 vs. Group-64 INT4 numerical fidelity and stream tokens offline via local **Ollama** in **Labs 25 & 26**.

10.1 Proprietary Cloud APIs vs. Open-Weight LLMs

FIRST PRINCIPLES & INTUITION **Ruler Millimeters & Local AI: Plain-English Intuition for Quantization**

An **open-weight model** is a trained neural network whose learned parameter file is published openly so anyone can run it privately on a laptop or campus GPU without sending a single prompt to the cloud. The catch is size: billions of high-precision decimal numbers consume vast GPU memory.

Everyday Quantization Analogy: Suppose you measured lengths with a laser to 10 decimal places, then rounded each measurement to the nearest millimeter on a standard ruler. You immediately fit **4× more numbers onto a single notebook page**, read them 4× faster, and still cut every board accurately! Weight quantization applies the exact same idea to neural networks, mapping 16-bit floating-point weights (**FP16**, 2 bytes each) onto compact 4-bit integers (**INT4**, 0.5 bytes each).

Key Terms Decoded: *Perplexity* measures how surprised a language model is by real text (lower is better; clean 4-bit compression changes perplexity by less than 0.1). *Outlier channels* are the rare ~0.1% of extra-large neural features that act like loudspeakers and need special protection from coarse rounding.

Modern AI engineering spans two complementary paradigms: **proprietary frontier cloud APIs** (such as Gemini, Claude, and GPT), which expose massive multimodal context windows with zero local infrastructure, and a thriving ecosystem of **Open-Weight Models** led by **Google Gemma 3**, **Meta Llama 3 & 4**, **Alibaba Qwen 2.5 & 3**, **DeepSeek R1/V3**, and **Mistral**. Their checkpoints are published openly on the Hugging Face Hub in memory-safe `safetensors` (eliminating Python pickle code-execution vulnerabilities) and single-file `GGUF` containers optimized for heterogeneous GPU/CPU serving via **Ollama**, **llama.cpp**, and **vLLM**.

Evaluation Criterion	Proprietary Cloud APIs	Local Open-Weight Models (Ollama / vLLM)
Privacy & Data Sovereignty	Prompts are transmitted to external servers	100% local offline execution, zero external data leakage
Cost at Scale	Ongoing billing per million input/output tokens	Zero marginal cost per token on existing campus or lab GPUs
Scientific Reproducibility	Weights and routing can update silently over time	SHA-pinned, auditable weights ideal for peer-reviewed research

Evaluation Criterion	Proprietary Cloud APIs	Local Open-Weight Models (Ollama / vLLM)
Domain Customization	Restricted to exposed prompt or API fine-tuning tiers	Full access to weights, custom quantization, and local LoRA/QLoRA

10.2 Quantization Mathematics: PTQ vs. QAT, Group-Wise INT4 & NF4

An 8-billion-parameter model stored in native 16-bit floating point (FP16/BF16) occupies roughly **16.8 GB of VRAM** (weights plus runtime overhead), exceeding a standard 12 GB university or consumer GPU (such as an NVIDIA RTX 3060). **Quantization** maps high-precision continuous weights onto compact low-bit integer grids through two overarching workflows: **Post-Training Quantization (PTQ)**, which compresses an already trained checkpoint in minutes using a small calibration set, and **Quantization-Aware Training (QAT)**, which simulates rounding noise during gradient updates via Straight-Through Estimators so weights proactively adapt to low-bit grids (especially crucial at 2-bit to 4-bit).

Within linear quantization, **Symmetric Quantization** centers the grid at zero using a single scale factor s (fast hardware dequantization), whereas **Asymmetric Quantization** introduces an integer zero-point offset z to tightly span skewed ranges $[w_{min}, w_{max}]$. Because a single extreme weight in a 4096×4096 tensor would inflate s and crush small weights to zero, modern schemes apply **Group-Wise Quantization** over small blocks of $g = 64$ or $g = 128$ weights, isolating outliers locally at an effective storage cost of only $b_{eff} = b + 16 / g$ bits per weight (4.25 bits/weight at $b = 4$, $g = 64$). Going one step further, **NormalFloat4 (NF4, Dettmers et al.)** exploits the empirical fact that pretrained LLM weights closely follow a zero-mean Gaussian distribution $N(0, \sigma^2)$: instead of uniformly spaced bins, NF4 places its 16 quantization levels at equal-probability quantiles of $N(0, 1)$, providing an information-theoretically optimal 4-bit code for Gaussian weights.

$$s_g = \max(|W_g|) / 7 \quad | \quad Q_g = \text{clip}(\text{round}(W_g / s_g), -7, +7) \quad | \quad \hat{W}_g = s_g \cdot Q_g \quad | \quad b_{\text{eff}} = 4 + 16 / g$$

Group-64 symmetric INT4 quantization compresses VRAM ~3.8× (fitting an 8B model in ~5.3 GB with >98.9% cosine similarity), while asymmetric schemes add zero-point $z = \text{round}(-w_{min} / s)$ and NF4 replaces uniform rounding with equal-probability Gaussian quantile bins (finer steps near zero, wider steps in the rare tails).

STEP-BY-STEP WORKED EXAMPLE **Hand-Quantizing 4 FP16 Weights to 4-Bit INT4 & Sizing an 8B Model**

Part A (Symmetric 4-Bit Quantization by Hand): Compress four FP16 weights `W = [-0.84, 0.21, 0.63, 1.05]` onto the signed 4-bit integer grid `[-7 .. +7]`:

1. **Compute the scale factor:** `s = max(|W|) / 7 = 1.05 / 7 = 0.15`.
2. **Divide & round to integers:** `W / 0.15 = [-5.6, +1.4, +4.2, +7.0]` → `Q = [-6, +1, +4, +7]` (stored in 4 bits each!).
3. **Dequantize (`Ŵ = s · Q`):** Multiplying back by `0.15` yields `Ŵ = [-0.90, 0.15, 0.60, 1.05]`.
4. **Verify reconstruction fidelity:** The errors `[0.06, 0.06, 0.03, 0.00]` produce a tiny Mean Squared Error `MSE = (0.0036 + 0.0036 + 0.0009 + 0) / 4 = 0.0020` (exact `0.002025`) and `99.83%` cosine similarity!

Part B (Fitting an 8-Billion-Parameter Model into a 12 GB RTX 3060): In native **FP16** (2 bytes per weight), weights alone consume `8B × 2 B = 16.0 GB` (~16.8 GB with runtime overhead), overflowing a 12 GB GPU. Under **4-bit Q4_K_M** (~4.5 to 4.8 effective bits per weight including scales), weights shrink to ~4.5 to 4.8 GB, or **~5.3 GB total VRAM** with runtime buffers, leaving ~6.7 GB free on a 12 GB RTX 3060 for the KV-Cache!

10.3 Outlier Channels, Frontier Quantizers & the Memory-Bandwidth Wall

In Transformers exceeding ~6 billion parameters, systematic **outlier feature channels** emerge: roughly 0.1% of hidden dimensions exhibit activation magnitudes 10× to 100× larger than typical channels across nearly every token. Naive per-tensor rounding destroys these salient pathways and causes catastrophic perplexity collapse. Modern quantization formats protect outlier fidelity through four complementary techniques:

Method / Format	Core Outlier & Error-Control Mechanism	Bits / Weight	Primary Deployment Role
LLM.int8()	Mixed-precision vector decomposition: isolates outlier columns in FP16, quantizes 99.9% to INT8	~8.02 bits	Drop-in PyTorch / bitsandbytes inference with zero degradation
GPTQ	Second-order layer-wise PTQ using inverse Hessian H^{-1} to compensate rounding errors across remaining weights	4.25 bits (g=128)	High-throughput GPU serving in vLLM & ExLlama
AWQ	Activation-Aware Weight Quantization: rescales the ~1% salient weight channels guided by activation norms	4.25 bits (g=128)	Hardware-friendly fast GEMV kernels with strong generalization
GGUF Q4_K_M	Mixed K-quant super-blocks preserving 6-bit precision on sensitive attention V and output tensors	~4.5 to 4.8 bits	Universal local CPU/GPU/Apple Silicon inference via Ollama & llama.cpp

Crucially, 4-bit quantization does not merely save VRAM: it directly accelerates local token generation! During single-user decoding ($B = 1$), producing one token requires reading every active model weight from GPU memory once to execute matrix-vector products. That yields a very low *arithmetic intensity* (operations performed per byte fetched from memory: ~1 FLOP/byte, far below modern GPU compute ridges exceeding 100 FLOP/byte), meaning the GPU cores wait on memory transfers. Autoregressive decode is therefore strictly **memory-bandwidth bound**:

Throughput (tok/s) ≈ GPU Memory Bandwidth (GB/s) / Active Model Bytes (GB)

On an NVIDIA RTX 3060 (12 GB, 360 GB/s memory bandwidth), an 8B Q4_K_M checkpoint (~5.3 GB) reaches a theoretical bandwidth ceiling of 360 / 5.3 ≈ 67.9 tok/s (delivering ~55 to 62 tok/s live after KV-Cache and kernel overhead), whereas an unquantized 16.8 GB FP16 checkpoint overflows VRAM across PCIe and stalls below 5 tok/s.

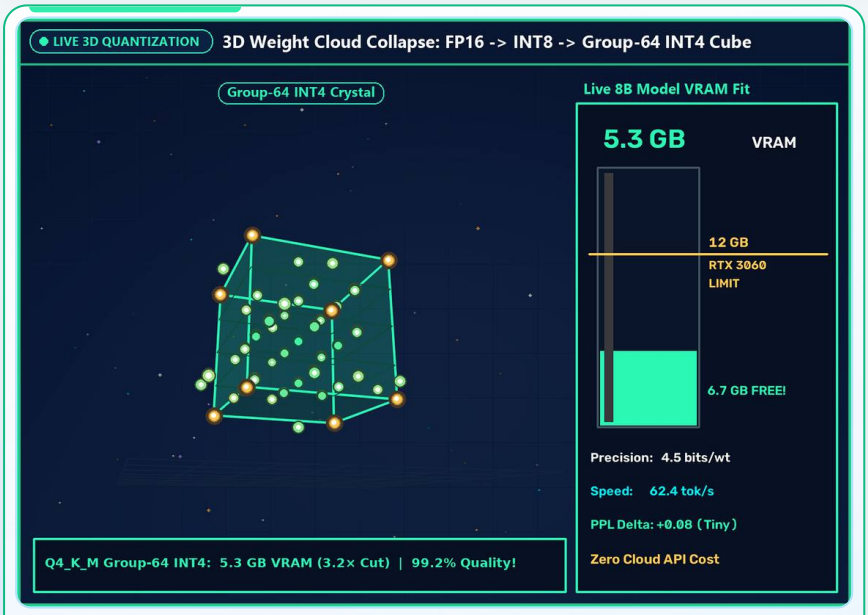


Figure 10.1

Geometric collapse of continuous FP16 weights onto a Group-64 INT4 crystal lattice: an 8B open-weight model shrinks to 5.3 GB VRAM (leaving 6.7 GB headroom on a 12 GB RTX 3060) while reaching 62.4 tok/s with minimal perplexity delta (+0.08).



Figure 10.2

Open-Weight LLM families, VRAM compression for an 8B model (FP32, FP16, INT8, and INT4 GGUF fitting well below the 12 GB RTX 3060 limit), and local inference engines (Ollama, llama.cpp, vLLM, Hugging Face).

10.4 Labs 25 & 26: INT4 Mathematics & Local Ollama Client

LAB 25 **FP16 to INT8 & Group-64 INT4 Quantization Fidelity with Outlier Channels**

`code/25_quantization_fp16_to_int4_math.py`

Synthesizes a realistic 1024×1024 LLM projection matrix (1,048,576 weights) injected with 3.5× outlier channels every 64 columns, compares `quantize_symmetric_int8()` against `quantize_groupwise_int4(group_size=64)` on matrix-vector projections `y = Wx` (measuring Cosine Similarity ~0.9896 and MSE at 3.8× compression), and runs a 12 GB RTX 3060 fit checker for 4B, 8B, 14B, and 70B models.

Guided Experiment: Change `group_size` from `64` to `1024` (per-row INT4) and `32`, then raise the outlier multiplier from `3.5` to `15.0` to observe firsthand why small quantization groups protect projection cosine fidelity against emergent outlier channels.

PYTHON 3.12+ `code/25_quantization_fp16_to_int4_math.py` Lab 25: INT8 & Group-64 INT4 quantization and GPU VRAM capacity verification

```python
import numpy as np


def quantize_symmetric_int8(W: np.ndarray):
    scale = np.max(np.abs(W), axis=-1, keepdims=True) / 127.0 + 1e-12
    q = np.clip(np.round(W / scale), -127, 127).astype(np.int8)
    W_hat = q.astype(np.float32) * scale
    return q, scale, W_hat


def quantize_groupwise_int4(W: np.ndarray, group_size: int = 64):
    """Group-wise 4-bit quantization (1 FP16 scale per 64 weights)."""
    orig_shape = W.shape
    flat = W.reshape(-1, group_size)
    scales = np.max(np.abs(flat), axis=1, keepdims=True) / 7.0 + 1e-12
    # Signed 4-bit integer range [-7, +7]
    q4 = np.clip(np.round(flat / scales), -7, 7).astype(np.int8)
    W_hat = (q4.astype(np.float32) * scales).reshape(orig_shape)
```

```
    eff_bits = 4.0 + 16.0 / group_size
    return q4, scales, W_hat, eff_bits


def cosine_sim(a: np.ndarray, b: np.ndarray) -> float:
    denom = np.linalg.norm(a) * np.linalg.norm(b) + 1e-12
    return float(np.dot(a.ravel(), b.ravel()) / denom)


def main():
    rng = np.random.default_rng(2026)
    # LLM projection matrix (1024 x 1024 = 1,048,576 weights) with outliers
    W_fp32 = rng.normal(0.0, 0.04, size=(1024, 1024)).astype(np.float32)
    W_fp32[:, ::64] *= 3.5  # Simulate realistic outlier weight channels
    W_fp16 = W_fp32.astype(np.float16).astype(np.float32)

    _, _, W_i8 = quantize_symmetric_int8(W_fp16)
    _, _, W_i4, eff_b4 = quantize_groupwise_int4(W_fp16, group_size=64)

    # Evaluate matrix-vector projection y = W x
    x = rng.normal(0.0, 1.0, size=(1024,)).astype(np.float32)
    y_ref = W_fp16 @ x
    y_i8 = W_i8 @ x
    y_i4 = W_i4 @ x

    mse_i8 = float(np.mean((y_ref - y_i8) ** 2))
# ... (full executable code in code/25_quantization_fp16_to_int4_math.py inside code.zip)
```

LAB 26 **Zero-Dependency Local Ollama Streaming Client & Live Throughput Telemetry**

`code/26_local_llm_ollama_and_hf.py`

Streams JSON-delimited tokens directly from a local **Ollama** server (`http://localhost:11434/api/generate`) using only Python's standard library (`urllib.request`), measures real-time decode speed in `tokens/sec` , and includes an automatic simulated Q4_K_M fallback whenever Ollama is not running.

Guided Experiment: Start Ollama locally (`ollama run gemma3:4b`), execute Lab 26, and compare live `tok/s` across `gemma3:4b` , `qwen2.5:7b` , and `deepseek-r1:8b` against the bandwidth ceiling `BW / Model_GB` of your own GPU.

PYTHON 3.12+ **code/26_local_llm_ollama_and_hf.py** Lab 26: Automatic VRAM estimator & local Ollama client (100% offline)

```
import json
import time
import urllib.error
import urllib.request

OLLAMA_URL = "http://localhost:11434/api/generate"


def query_ollama_stream(
    prompt: str, model: str = "gemma3:4b", temperature: float = 0.4
):
    payload = json.dumps(
        {
            "model": model,
            "prompt": prompt,
            "stream": True,
            "options": {"temperature": temperature, "num_predict": 140},
        }
    ).encode("utf-8")

    req = urllib.request.Request(
        OLLAMA_URL,
        data=payload,
        headers={"Content-Type": "application/json"},
    )
```

```
    t0 = time.perf_counter()
    tokens = 0
    chunks = []
    with urllib.request.urlopen(req, timeout=8) as resp:
        for raw_line in resp:
            obj = json.loads(raw_line.decode("utf-8"))
            tok = obj.get("response", "")
            if tok:
                print(tok, end="", flush=True)
                chunks.append(tok)
                tokens += 1
            if obj.get("done"):
                break
    dt = max(1e-4, time.perf_counter() - t0)
    print(
        f"\n\n[Telemetry] Model={model} | Tokens={tokens} | "
        f"Elapsed={dt:.2f}s | Speed={tokens / dt:.1f} tok/s"
# ... (full executable code in code/26_local_llm_ollama_and_hf.py inside code.zip)
```

Local Open-Model Selection Guide for a 12 GB GPU (RTX 3060 / 4070)

1. Everyday Multilingual & Multimodal Use: `gemma3:4b` or `gemma3:12b` (Q4_K_M, ~8.1 GB VRAM, strong multilingual and scientific prose). **2. Code & Software Engineering:** `qwen2.5-coder:7b` or `14b` (Q4_K_M, ~8.8 GB VRAM). **3. Deep Mathematical Reasoning:** `deepseek-r1:8b` or `14b` with 100% offline execution and zero exposure of private data. Always reserve 1.5 to 2.5 GB of VRAM headroom for the KV-Cache and display driver.

CHAPTER 11

Model Fine-Tuning, LoRA / QLoRA & Alignment

Intrinsic low-rank adaptation (LoRA & QLoRA Double Quantization) plus the formal alignment mathematics of SFT, RLHF (Bradley-Terry + PPO), DPO, and GRPO verifiable reasoning.

Learning Objectives & Theoretical Scope

- Explain the **intrinsic low-rank hypothesis** (Aghajanyan et al.; Hu et al.) and derive exact parameter counts, initialization (`A` Gaussian, `B = 0`), scaling `α/r`, and zero-latency weight merging in **LoRA**.
- Detail the three memory breakthroughs of **QLoRA** (NF4 frozen base, Double Quantization of block scales, and Unified-Memory Paged Optimizers) enabling 8B to 14B adaptation on one 12 GB GPU.
- Compare all four LLM post-training objectives formally: **SFT** cross-entropy, **RLHF** (Bradley-Terry Reward Model + KL-regularized PPO), closed-form **DPO**, and critic-free **GRPO**.
- Explain how rule-based verifiable rewards in **GRPO** (`A_i = (r_i - mean(r)) / std(r)`) unlock emergent self-checking in mathematics and code.
- Build `LoRALinear` from scratch and verify merged-weight equivalence down to `10^-15` precision in **Lab 27**.

11.1 Low-Rank Adaptation (LoRA) & QLoRA: Freezing 99.6% of Weights

FIRST PRINCIPLES & INTUITION **Tracing Sheets Over an Encyclopedia: Intuition for LoRA, SFT, DPO & GRPO**

Suppose you own a 50,000-page printed encyclopedia (a pretrained LLM) and want it to specialize in radio astronomy or Greek clinical notes. Reprinting all 50,000 pages (*Full Fine-Tuning*) is slow, expensive, and risks smudging unrelated chapters (*catastrophic forgetting*).

- **LoRA (Transparent Tracing Sheet):** Instead of reprinting the book, LoRA places a thin transparent tracing sheet (`B · A`) over each page, recording only lightweight corrections while leaving all original printed text (`W_0`) completely frozen. The thickness of the sheet is the *rank* `r`.
- **SFT (Supervised Fine-Tuning):** Like studying a workbook of solved examples (`prompt → expert answer`) to master tone, clarity, and structured formatting.
- **DPO & GRPO (Learning From Comparisons):** Rather than copying one static answer, **DPO** compares pairs of candidate solutions (*winner vs. loser*) and reinforces why the winner excels, while **GRPO** generates a group of attempts at a math or coding problem, grades each automatically with an exact verifier, and rewards whichever solutions beat the group average.

Full Fine-Tuning of an 8-billion-parameter model in mixed precision requires more than 60 GB of GPU VRAM for weights, gradients, and 32-bit AdamW momentum/variance states, while risking *catastrophic forgetting* of general capabilities. Empirical studies by **Aghajanyan et al. (2021)** revealed that pretrained foundation models possess a surprisingly small *intrinsic dimension*: task adaptation occurs inside a compact low-dimensional subspace rather than across all coordinates simultaneously.

LoRA (Low-Rank Adaptation, Hu et al., 2021) operationalizes this hypothesis by freezing the massive pretrained weight matrix $W_0 \in R^{d \times d}$ completely (`grad = 0`) and modeling the adaptation delta ΔW as a product of two slender trainable matrices: a down-projection $A \in R^{r \times d}$ initialized from a zero-mean Gaussian distribution $N(0, \sigma^2)$ and an up-projection $B \in R^{d \times r}$ initialized identically to zero. Because $B = 0$ at step 0, $\Delta W = BA = 0$, guaranteeing that training starts smoothly from the exact pretrained checkpoint without perturbing existing knowledge. During forward propagation, outputs are scaled by α / r so that gradient magnitudes remain stable when experimenting with different ranks r (commonly $r \in \{8, 16, 32\}$ with $\alpha = 2r$):

$$W' = W_0 + (\alpha / r) \cdot B \cdot A \quad | \quad h = W_0 x + (\alpha / r) \cdot B(Ax) \quad | \quad \text{Params}_{\text{LoRA}} = r(d_{\text{in}} + d_{\text{out}})$$

For d = 4096 and rank r = 8, trainable weights collapse from 16,777,216 to 65,536 (0.39% of W_0, ~18 MB adapter file across an 8B model). At deployment, folding (α/r)BA into W_0 delivers mathematically identical predictions with 0.0 ms inference latency overhead.

To push fine-tuning onto single 12 GB university GPUs, **QLoRA (Dettmers et al., 2023)** combines LoRA with three systems-level memory optimizations while preserving full 16-bit task accuracy: (1) storing the frozen base W_0 in **4-bit NormalFloat (NF4)** and dequantizing blocks on the fly to BF16 exclusively during matrix multiplications, (2) **Double Quantization (DQ)**, which quantizes the FP32 quantization scale constants themselves into FP8 (saving an extra ~0.37 bits per parameter), and (3) **Paged Optimizers**, which leverage NVIDIA Unified Memory to transparently page AdamW optimizer states between GPU VRAM and host CPU RAM during occasional long-sequence gradient spikes so training never crashes with Out-Of-Memory errors.

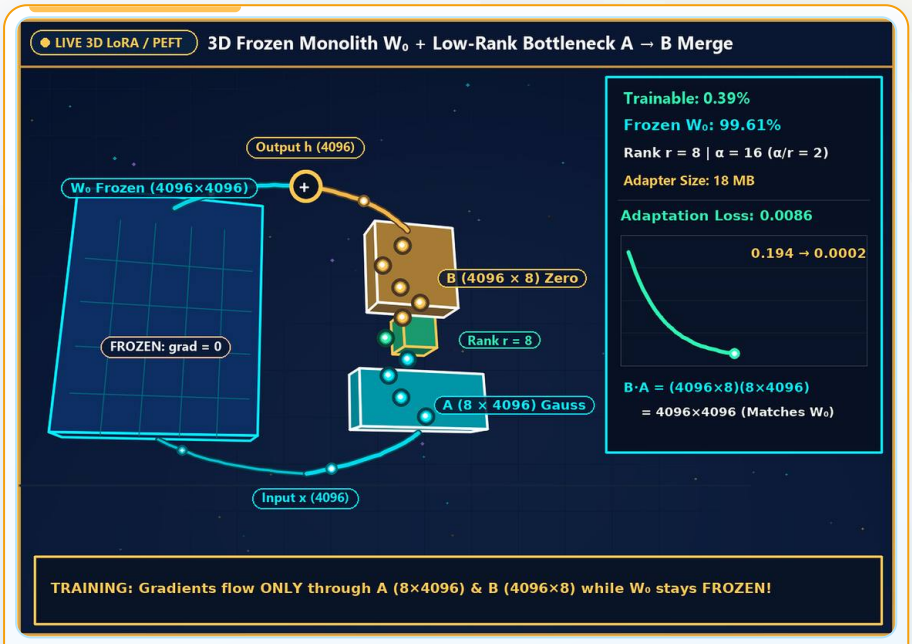


Figure 11.1

3D geometric view of LoRA low-rank adaptation: the frozen 4096×4096 monolith W_0 (99.61% of weights, grad = 0) runs in parallel with the slender rank-8 Gaussian A and zero-initialized B bottleneck (0.39% trainable, ~18 MB adapter), before lossless folding into W_0.

11.2 Post-Training & Alignment Mathematics: SFT, RLHF, DPO & GRPO

A raw pretrained checkpoint encodes broad factual and linguistic structure, yet behaves solely as a statistical document completer. Converting it into an accurate, helpful, honest, and self-verifying scientific partner requires structured **Post-Training Alignment** across four formal paradigms:

1. **Stage 1, Supervised Fine-Tuning (SFT):** Minimizes standard next-token cross-entropy exclusively over curated model response tokens y_t conditioned on user prompt x: $L_{SFT}(\theta) = -\Sigma_t \log \pi_\theta(y_t \mid x, y_{<t})$. Only one model policy resides in VRAM, teaching instruction adherence and structured formatting.
2. **Stage 2A, RLHF (Bradley-Terry Reward Model + KL-Regularized PPO):** Human annotators rank pairs of candidate responses $(y_w \succ y_l)$ (chosen vs. rejected). First, a separate neural Reward Model $r_\varphi(x, y)$ is trained under the Bradley-Terry likelihood $-E[\log \sigma(r_\phi(x, y_w) - r_\phi(x, y_1))]$. Next, **Proximal Policy Optimization (PPO)** maximizes expected reward minus a Kullback-Leibler (KL) divergence penalty (an elastic statistical leash) against the frozen SFT reference policy π_{ref} so the model cannot drift into nonsensical reward hacking: $\max_\theta E[r_\phi(x, y)] - \beta \cdot D_{KL}(\pi_\theta \,||\, \pi_{ref})$. Its main bottleneck is operational complexity: four large neural networks (Policy, Reference, Reward, and Value Critic) must be orchestrated concurrently in VRAM.

3. **Stage 2B, Direct Preference Optimization (DPO, Rafailov et al., 2023):** Demonstrates analytically that the optimal KL-constrained policy induces a closed-form mapping between reward and policy log-ratios, eliminating both the separate Reward Model and the RL sampling loop! Optimizing a single supervised binary cross-entropy objective directly on preference pairs (x, y_w, y_l) halves VRAM to two models (Policy and Reference).
4. **Stage 2C, Group Relative Policy Optimization (GRPO, DeepSeek-R1):** Built specifically for *Reinforcement Learning with Verifiable Rewards (RLVR)* in mathematics, unit-tested code, and formal science. For each prompt x, GRPO samples a group of G candidate completions $\{o_1, ..., o_G\}$, evaluates each with a deterministic rule-based verifier (exact numerical answer match, compiler/unit test pass, or format check) to obtain scalar rewards $\{r_1, ..., r_G\}$, and normalizes advantage strictly inside the group. Eliminating the learned Critic slashes memory while driving emergent test-time self-verification and backtracking.

$$L_{DPO}(\theta) = -E_{(x, y_w, y_l)}[\log \sigma(\beta \log(\pi_\theta(y_w|x) / \pi_{ref}(y_w|x)) - \beta \log(\pi_\theta(y_l|x) / \pi_{ref}(y_l|x)))]$$

$$\text{GRPO Group Relative Advantage: } A_i = (r_i - \text{mean}(r_{1..G})) / (\text{std}(r_{1..G}) + \varepsilon)$$

DPO increases the relative log-probability margin of preferred responses y_w over rejected y_l using only Policy and Reference models, while GRPO replaces the neural PPO Critic with zero-mean, unit-variance group-normalized verifiable rewards A_i.

STEP-BY-STEP WORKED EXAMPLE

Counting LoRA Rank-8 Savings (256x Fewer Weights) & 4-Response GRPO Advantage

Part A (Why Rank-8 LoRA Trains Only 0.39% of a Layer): Consider one standard `4096 × 4096` projection matrix inside an 8B Transformer:

- **Frozen Base `W₀`:** `4,096 × 4,096 = 16,777,216` frozen weights (`grad = 0`).
- **Trainable LoRA (`r = 8`):** Down-projection `A` (`8 × 4,096 = 32,768`) plus up-projection `B` (`4,096 × 8 = 32,768`) totals just `65,536` **trainable weights**, which is `65,536 / 16,777,216 = 0.39%` (exactly **256× fewer** parameters!).

Part B (4-Response GRPO Group Advantage by Hand): Suppose a reasoning model samples `G = 4` solutions to a calculus problem and an automatic unit verifier assigns binary rewards `r = [1, 0, 1, 0]` (1 = verified correct, 0 = wrong):

1. **Group Mean:** `μ = (1 + 0 + 1 + 0) / 4 = 0.5`.
2. **Group Standard Deviation:** `σ = √(((+0.5)² + (-0.5)² + (+0.5)² + (-0.5)²) / 4) = √0.25 = 0.5`.
3. **Normalized Advantages `Aᵢ = (rᵢ - μ) / σ`:** `A = [+1, -1, +1, -1]`. Policy updates immediately boost the two correct chains (`+1`) and penalize the two flawed chains (`-1`) without needing a multi-gigabyte neural Critic network!

Post-Training Stage	Models in VRAM	Supervision Signal	Primary Engineering Strength
SFT	1 (Policy only)	Curated (x, y_{gold}) demonstrations	Fastest way to establish chat templates, domain style & JSON schemas
RLHF (RM + PPO)	4 (Policy, Ref, RM, Critic)	Online rollouts scored by neural Reward Model	Handles nuanced open-ended helpfulness & safety trade-offs at scale
DPO	2 (Policy + frozen Ref)	Offline preference pairs $(y_w > y_l)$	Stable closed-form gradient updates; pairs effortlessly with QLoRA
GRPO (RLVR)	2 (Policy + frozen Ref; no Critic)	Group of G rollouts + deterministic verifiers	Unlocks verifiable multi-step reasoning in math, coding & science



Figure 11.2

Modern LLM post-training & alignment architecture comparing SFT, RLHF (Bradley-Terry RM + PPO), closed-form DPO, and critic-free GRPO with exact objective equations and VRAM footprints.

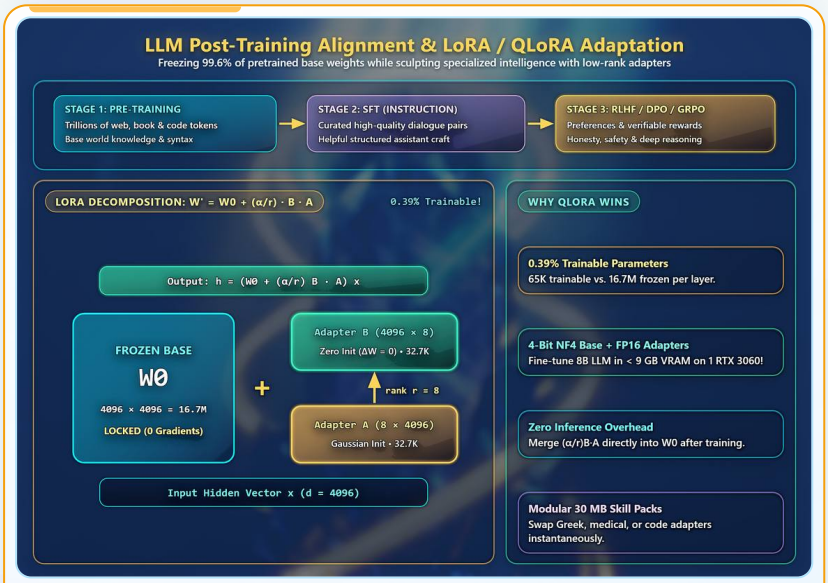


Figure 11.3

End-to-end alignment lifecycle (Pre-Training, SFT, RLHF/DPO/GRPO) paired with low-rank LoRA / QLoRA decomposition (0.39% trainable parameters and zero latency overhead after weight merging).

11.3 Lab 27: Implementing LoRA from Scratch & Merge Verification

LAB 27 **Low-Rank Adaptation (r=8) from Scratch & Machine-Precision Weight Merging**

`code/27_lora_from_scratch_pytorch.py`

Constructs a transparent `LoRALinearNumPy` layer with frozen base weights `W0` (`256×256`), Gaussian down-projection `A` (`256×8`), and zero-initialized up-projection `B` (`8×256`). Analytical backpropagation updates exclusively `A` and `B` until adaptation MSE converges from `0.1936` to `0.000000`, then proves that `merge_weights()` matches unmerged inference within machine precision (7.99×10^{-15}).

Guided Experiment: Lower `rank` from `8` to `2` (below the synthetic target's true shift rank of 4) and verify that final MSE plateaus above zero, demonstrating geometrically why LoRA rank `r` must meet or exceed the task's intrinsic adaptation rank.

PYTHON `code/27_lora_from_scratch_pytorch.py`

3.12+ Lab 27: Training LoRA adapters (r=8) & zero-overhead weight-merge verification

```python
import numpy as np


class LoRALinearNumPy:
    """Transparent mathematical implementation of Hu et al. (2021) LoRA."""

    def __init__(
        self,
        d_in: int,
        d_out: int,
        rank: int = 8,
        alpha: float = 16.0,
        seed: int = 42,
    ):
        rng = np.random.default_rng(seed)
        # Frozen base weights W0, trainable down-projection A, zero-init B
        self.W0 = rng.normal(0.0, 0.15, size=(d_in, d_out)).astype(np.float64)
        self.A = rng.normal(0.0, 0.05, size=(d_in, rank)).astype(np.float64)
        self.B = np.zeros((rank, d_out), dtype=np.float64)
        self.scale = alpha / float(rank)

    def forward(self, X: np.ndarray) -> np.ndarray:
        return X @ self.W0 + self.scale * ((X @ self.A) @ self.B)

    def train_step(
        self, X: np.ndarray, Y_target: np.ndarray, lr: float = 0.08
    ) -> float:
        H = X @ self.A
        Y_pred = X @ self.W0 + self.scale * (H @ self.B)
        err = (2.0 / float(Y_target.size)) * (Y_pred - Y_target)
        loss = float(np.mean((Y_pred - Y_target) ** 2))

        # Gradients ONLY for A and B (W0 stays 100% frozen)
        grad_B = self.scale * (H.T @ err)
        grad_A = self.scale * (X.T @ (err @ self.B.T))
        self.A -= lr * grad_A
        self.B -= lr * grad_B
        return loss

    def merge_weights(self) -> np.ndarray:
        """Fuses LoRA adapter into W0 for 0.0 ms deployment overhead."""
        return self.W0 + self.scale * (self.A @ self.B)


def main():
    d_in, d_out, rank = 256, 256, 8
# ... (full executable code in code/27_lora_from_scratch_pytorch.py inside code.zip)
```

★ **Practical LoRA / QLoRA Hyperparameter Guidelines for Single-GPU Research**

1. Rank, Scaling & Target Modules: Start with `r = 8` or `r = 16`, set `alpha = 2 × r`, and attach adapters across both attention (`q_proj, k_proj, v_proj, o_proj`) and MLP (`gate_proj, up_proj, down_proj`) projections for peak fidelity. **2. Data Quality Over Volume:** A clean, verified dataset of 1,000 to 5,000 domain examples followed by a concise DPO preference pass consistently outperforms noisy 100k dumps while running QLoRA comfortably inside a 12 GB laboratory GPU.

CHAPTER 12

RAG Systems, Vector Databases & Autonomous AI Agents (MCP)

Dense embeddings vs. BM25 hybrid retrieval with Reciprocal Rank Fusion (RRF), HNSW indexing, Cross-Encoder reranking, and ReAct agents over the Model Context Protocol (MCP).

Learning Objectives & Theoretical Scope

- Contrast **Dense Semantic Embeddings** against **Sparse Lexical BM25** and combine both cleanly using **Reciprocal Rank Fusion (RRF)**.
- Analyze production retrieval engineering: **HNSW** logarithmic Approximate Nearest Neighbor graphs, semantic chunking & overlap trade-offs, **Cross-Encoder reranking**, and faithfulness evaluation.
- Trace the autonomous **ReAct (Reason + Act)** agent loop and understand how deterministic JSON tool calling eliminates calculation and hallucination failures.
- Specify the **Model Context Protocol (MCP)** Client-Host-Server JSON-RPC architecture across Tools, Resources, Prompts, and sandboxed permission boundaries.
- Implement a zero-dependency semantic vector RAG store and a multi-tool ReAct/MCP agent in **Labs 28 & 29**.

12.1 Dense vs. Sparse Retrieval, Hybrid RRF & HNSW Vector Indexing

FIRST PRINCIPLES & INTUITION **Closed-Book Exam vs. Open-Book Research & the USB-C Port for AI**

Three everyday analogies illuminate why modern production AI rarely runs as a bare chatbot:

- **Standard LLM = Closed-Book Exam:** Answering purely from memorized training weights excels at general principles, yet invites confident guessing (*hallucination*) when asked about private lab logs, yesterday's telemetry, or exact page numbers.
- **RAG (Retrieval-Augmented Generation) = Open-Book Exam:** Before writing a single sentence, the system opens your verified library to the exact matching paragraphs (*chunks*), places them on the desk, and cites sources (`[1][2]`) explicitly.
- **MCP (Model Context Protocol) = Universal USB-C Port:** Just as one USB-C plug connects any laptop safely to monitors, drives, and instruments, MCP gives any LLM one standardized, permission-checked socket to call deterministic calculators, SQL databases, Python runtimes, and laboratory sensors!

Plain-English Glossary: An *embedding* converts text into a coordinate vector where similar meanings point in the same direction; a *vector database* indexes those coordinates for rapid neighbor search; and a *ReAct agent* alternates between reasoning and calling deterministic tools.

Even the strongest language model carries a static pretraining knowledge cutoff and cannot see private laboratory logs, live telemetry, or updated technical specifications. **Retrieval-Augmented Generation (RAG)** solves this non-parametrically: live corpora are partitioned into semantic passages (*chunks*), indexed for millisecond retrieval, and injected into the prompt so the LLM generates grounded answers backed by verifiable citations (`[1][2]`) without modifying a single model weight.

High-precision scientific RAG rarely relies on a single retrieval signal. Instead, it fuses two complementary search engines:

- **Dense Semantic Retrieval (Bi-Encoder Embeddings):** A neural embedding model maps queries and passages into unit-normalized vectors in R^d (e.g., 768 to 3072 dimensions). Synonyms and conceptual paraphrases ("GPU memory footprint" vs. "VRAM usage") align tightly under **Cosine Similarity**, even when sharing zero exact keywords.
- **Sparse Lexical Retrieval (BM25 / TF-IDF):** Scores exact term-frequency and inverse-document-frequency matches with length normalization. Indispensable whenever a query contains exact identifiers such as `FIPS 203`, `Q4_K_M`,

catalog IDs (`blc1`), or compiler error codes that dense models occasionally blur.

- **Hybrid Retrieval via Reciprocal Rank Fusion (RRF):** Because raw cosine scores (0 to 1) and unbounded BM25 scores live on incompatible scales, **RRF** merges ranked lists solely via reciprocal ranks using a smoothing constant `k = 60` (Cormack et al.), rewarding passages corroborated by both semantic meaning and exact terminology:

$$\cos(q, d) = (q \cdot d) / (||q|| \cdot ||d||) \quad | \quad \mathrm{RRF}(d) = \Sigma_{i \in \{\mathrm{Dense,\ BM25}\}} \ 1 / (k + \mathrm{rank}_i(d))$$

Unit normalization (||v|| = 1, placing every vector on a sphere of radius 1 so only semantic direction matters) turns dense cosine scoring into one BLAS matrix-vector product (matrix @ q_vec). Combining dense and BM25 rankings with k = 60 systematically lifts Recall@10 over either retriever alone.

STEP-BY-STEP WORKED EXAMPLE **Hand-Computing Cosine Similarity & Reciprocal Rank Fusion (RRF)**

Part A (3D Cosine Similarity by Hand): Imagine a 3-axis semantic space representing `[GPU/VRAM, Quantization, Cooking]`. A student submits query vector `q = [0.8, 0.6, 0.0]` (unit norm `√(0.64 + 0.36) = 1.0`), compared against two unit-normalized passages:

- **Passage** d_1 (*"4-bit VRAM compression"*): $d_1 = [0.6, 0.8, 0.0]$ → $\cos(q, d_1) = (0.8)(0.6) + (0.6)(0.8) + (0)(0) = 0.48 + 0.48 = 0.96$ (**near-parallel semantic match!**).
- **Passage** d_2 (*"Pasta recipe"*): $d_2 = [0.0, 0.0, 1.0]$ → $\cos(q, d_2) = (0.8)(0) + (0.6)(0) + (0)(1) = 0.00$ (**orthogonal, completely unrelated!**).

Part B (Hybrid RRF Score with `k = 60`): Suppose Dense Search ranks d_1 at **#1** while keyword BM25 ranks it at **#2**. Its fused score is $\mathrm{RRF}(d_1) = 1 / (60 + 1) + 1 / (60 + 2) = 1/61 + 1/62 = 0.01639 + 0.01613 = 0.0325$, cleanly outranking a passage that appears #1 in keyword search but #50 semantically ($1/61 + 1/110 = 0.0255$).

Scaling RAG from hundreds of notes to tens of millions of scientific documents requires four production systems practices:

1. **HNSW Approximate Nearest Neighbor Indexing:** Exhaustive linear scan scales as $O(N \cdot d)$ and stalls at million-document scale. Vector databases (FAISS, Qdrant, Milvus, pgvector) construct **Hierarchical Navigable Small World (HNSW)** proximity graphs: sparse upper layers jump swiftly across distant semantic clusters while dense lower layers refine local neighbors in logarithmic $O(\log N)$ latency with >98% recall.
2. **Semantic Chunking & Overlap Trade-Offs:** Overly tiny chunks (<100 tokens) sever equations from their variable definitions, while giant chunks (>1500 tokens) dilute embedding vectors with mixed topics. Structure-aware chunking (256 to 512 tokens with 10% to 20% sliding overlap along heading and paragraph boundaries) preserves context cleanly.
3. **Two-Stage Cross-Encoder Reranking:** Fast hybrid retrieval fetches the Top-50 candidates in milliseconds; a deeper **Cross-Encoder Reranker** then concatenates `[query, chunk]` directly through full Transformer self-attention to select the exact Top-5 gold passages before generation.
4. **Faithfulness & Groundedness Evaluation:** Rigorous RAG systems audit three automated metrics continuously: *Context Recall* (was the true evidence retrieved?), *Faithfulness* (is every generated claim entailed by the retrieved passages with zero fabricated citations?), and *Answer Relevance*.

12.2 Autonomous AI Agents, ReAct Loops & Model Context Protocol (MCP)

Autonomous AI Agents extend passive text generation into verifiable action execution via the **ReAct (Reason + Act)** control loop: *Thought → Structured JSON Tool Call → Deterministic Observation → Grounded Answer*. Rather than asking an LLM to approximate floating-point VRAM formulas, SQL joins, or laboratory sensor checks mentally, the agent invokes deterministic software tools and conditions its next reasoning step on the exact returned JSON observation.

Historically, connecting *M* foundation models to *N* databases, Python runtimes, and scientific instruments demanded *M* × *N* fragile custom wrappers. The open **Model Context Protocol (MCP)** reduces this to *M* + *N* by defining a standardized **Client-Host-Server JSON-RPC 2.0** interface (a lightweight request-and-reply protocol built on plain JSON messages, often described as a universal "USB-C port for AI tools") with strict schema validation and security isolation:

MCP Layer / Primitive	Control Authority	JSON-RPC Role & Security Boundary
MCP Host	User / Application Runtime	IDE, desktop workspace, or orchestrator enforcing global policies, secrets & user consent
MCP Client	Host Protocol Engine	Maintains isolated 1:1 stateful JSON-RPC sessions with each connected MCP Server
MCP Server	Domain Service Process	Lightweight local (`stdio`) or remote (Streamable HTTP) capability provider
Tools	Model-Controlled	Typed executable functions (`tools/list` , `tools/call`) validated by JSON Schema
Resources	Application-Controlled	Read-only addressable data URIs (files, database snapshots, observatory logs)
Prompts	User-Controlled	Reusable parameterized workflow templates and domain audit checklists
Sandboxed Permissions	Least-Privilege Guardrail	Read-only defaults, path/network allowlists, and explicit human approval before state mutations

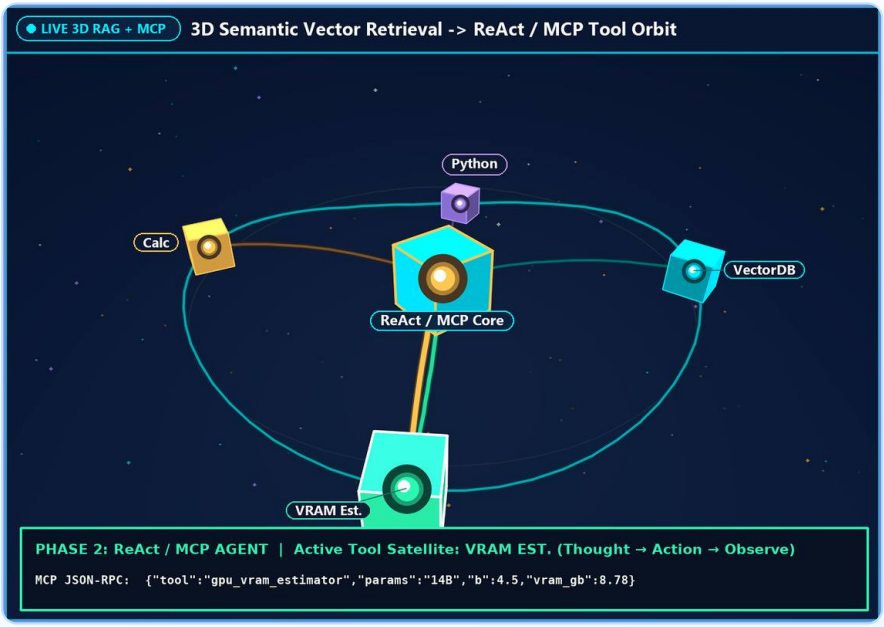


Figure 12.1

3D architectural orbit of a ReAct / MCP Agent Core dispatching typed JSON-RPC tool invocations to sandboxed satellites (VectorDB, Python Runtime, Calculator, and GPU VRAM Estimator) for zero-hallucination reasoning.

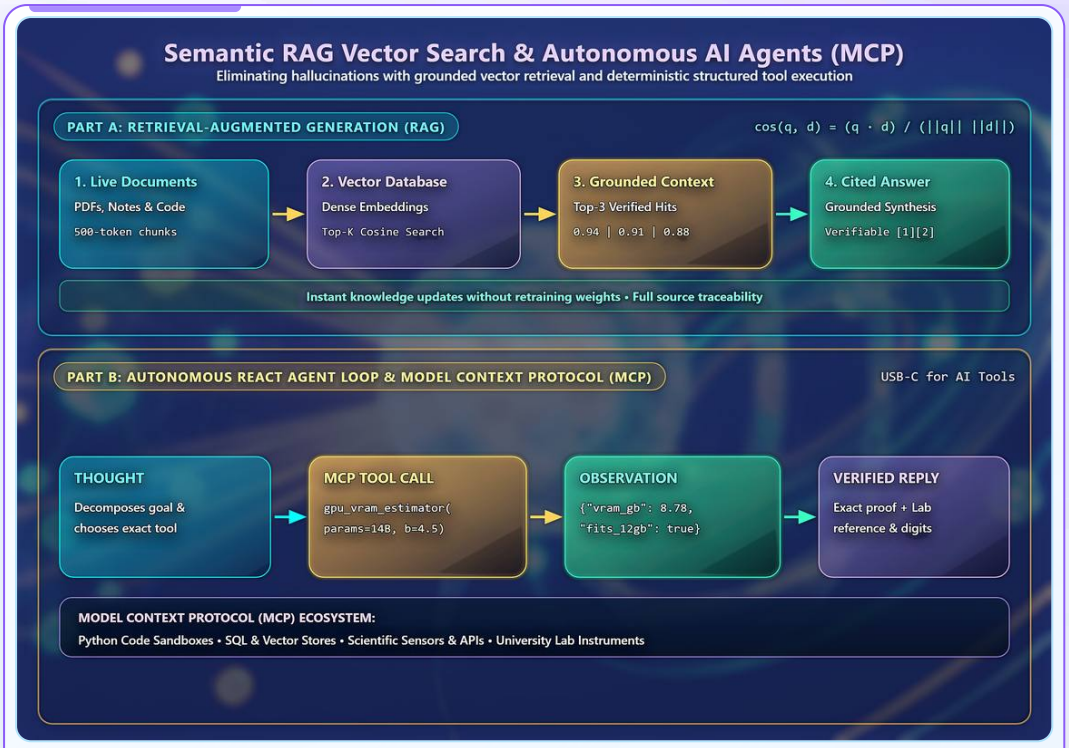


Figure 12.2

Semantic RAG retrieval architecture (documents, vector database, Top-K cosine similarity, grounded citations) paired with the autonomous ReAct agent cycle and Model Context Protocol (MCP) tools.

12.3 Labs 28 & 29: Vector RAG Engine & ReAct/MCP Agent

LAB 28 **End-to-End Semantic Vector RAG Store, Cosine Top-K & Citation Prompt Builder**

`code/28_rag_vector_search_from_scratch.py`

Constructs `VectorRAGStore` from scratch in pure NumPy over the seminar's 8-chapter knowledge base: computes smoothed IDF weights, L2-normalizes every document vector into a single matrix, retrieves Top-K chunks in one dot-product (`self.matrix @ q_vec`), and builds a strictly grounded prompt with explicit `[1][2]` citation constraints.

Guided Experiment: Add two new technical entries to `KNOWLEDGE_BASE` (for example, RoPE and GRPO), then implement a 10-line `rrf_fuse(rank_list_a, rank_list_b, k=60)` function combining keyword overlap rank with cosine TF-IDF rank to test hybrid ranking stability.

PYTHON 3.12+ `code/28_rag_vector_search_from_scratch.py` Lab 28: RAG vector store & Top-K retrieval with cosine similarity

```python
import math
import re
import numpy as np

KNOWLEDGE_BASE = [
    (
        "Ch.03 | XOR MLP",
        "A single linear perceptron fails on XOR, whereas a 2-layer MLP with "
        "ReLU hidden units bends space to achieve 100 percent accuracy.",
    ),
    (
        "Ch.05 | Transformers",
        "Scaled dot-product Self-Attention computes softmax of Q times K "
        "transpose divided by square root of dk to link distant words.",
    ),
    (
        "Ch.08 | GPU CUDA",
        "GPUs pack thousands of parallel ALUs and Tensor Cores to multiply "
        "massive neural matrices 20 to 60 times faster than CPUs.",
```

```
    ),
    (
        "Ch.09 | KV-Cache",
        "During autoregressive LLM decoding, the KV-Cache stores past Key "
        "and Value tensors so generation scales linearly, not quadratically.",
    ),
    (
        "Ch.10 | Quantization",
        "4-bit GGUF Q4_K_M quantization shrinks an 8B open-weight Llama or "
        "Gemma model from 16 GB FP16 to 4.8 GB to run locally on RTX 3060.",
    ),
    (
        "Ch.11 | LoRA PEFT",
        "LoRA freezes 99.4 percent of foundation weights W0 and trains small "
        "low-rank matrices A and B on a single university GPU.",
    ),
    (
        "Ch.12 | MCP Agents",
        "Model Context Protocol (MCP) standardizes how autonomous AI agents "
        "safely discover and call external tools, databases, and sandboxes.",
    ),
    (
        "Ch.16 | Quantum AI",
        "A qubit exists on the 3D Bloch sphere in superposition; 300 "
        "entangled qubits span more states than atoms in the universe.",
    ),
]


# ... (full executable code in code/28_rag_vector_search_from_scratch.py inside code.zip)
```

LAB 29 **Autonomous ReAct Agent & MCP Tool Registry with Safe JSON Tool Calling**

`code/29_ai_agent_tool_calling_react.py`

Implements the full `Thought → Tool Call (JSON) → Observation → Verified Answer` pipeline backed by three deterministic local tools in `TOOL_REGISTRY`: `gpu_vram_estimator`, a restricted sandboxed `calculator`, and `search_syllabus`, auditing whether a 14B Q4_K_M model fits on an RTX 3060 12 GB GPU without guessing.

Guided Experiment: Register a fourth deterministic tool `kv_cache_estimator(layers, n_kv, d_head, seq_len)` inside `TOOL_REGISTRY` and add a validation step to `run_react_agent()` that rejects any deployment where `weights_gb + kv_gb > 11.4`.

PYTHON 3.12+ `code/29_ai_agent_tool_calling_react.py` Lab 29: Autonomous ReAct agent & MCP tool registry (JSON Tool Calling)

```python
import json
import math


# === 1. REGISTERED MCP-STYLE TOOL DEFINITIONS ===
def tool_gpu_vram_estimator(
    params_billions: float, quant_bits: float = 4.5
) -> dict:
    """Calculates approximate VRAM needed to run an open-weight LLM locally."""
    weights_gb = params_billions * (quant_bits / 8.0)
    kv_and_overhead = 0.9
    total = round(weights_gb + kv_and_overhead, 2)
    return {
        "model_params_B": params_billions,
        "quant_bits": quant_bits,
        "estimated_vram_gb": total,
        "fits_12gb_rtx3060": total <= 11.4,
    }


def tool_calculator(expression: str) -> dict:
```

```
    """Safe mathematical evaluator for arithmetic and logarithms."""
    allowed = {"sqrt": math.sqrt, "log2": math.log2, "pow": pow, "round": round}
    val = eval(expression, {"__builtins__": {}}, allowed)
    return {"expression": expression, "result": round(float(val), 4)}


def tool_search_syllabus(topic: str) -> dict:
    """Looks up the relevant chapter and lab in the UoP ECE AI Masterclass."""
    index = {
        "lora": (
            "Chapter XI (Fine-Tuning, LoRA & Alignment) -> "
            "Lab 27_lora_from_scratch_pytorch.py"
        ),
        "quantization": (
            "Chapter X (Open-Source AI & Quantization) -> "
            "Lab 25_quantization_fp16_to_int4_math.py"
        ),
        "rag": (
            "Chapter XII (RAG, Vector Databases & AI Agents) -> "
            "Lab 28_rag_vector_search_from_scratch.py"
        ),
    }
    key = next((k for k in index if k in topic.lower()), "quantization")
    return {"topic": topic, "courseware_reference": index[key]}


TOOL_REGISTRY = {
    "gpu_vram_estimator": tool_gpu_vram_estimator,
    "calculator": tool_calculator,
    "search_syllabus": tool_search_syllabus,
}


def run_react_agent(user_goal: str):
    """Executes a multi-step ReAct agent loop with JSON tool invocations."""
    print("=== AUTONOMOUS ReAct AGENT (MCP TOOL-CALLING LOOP) ===")
    print(f"USER GOAL: {user_goal}\n")
# ... (full executable code in code/29_ai_agent_tool_calling_react.py inside code.zip)
```

Adaptation Pattern	Primary Strength	Knowledge Freshness	Compute & Engineering Cost
Prompt Engineering	Fast formatting, role framing, few-shot guidance	Limited to model cutoff + active prompt	Zero training cost, instant iteration
RAG + MCP Tools	Verifiable citations & deterministic tool execution	Real-time updates via hybrid vector index	Lightweight indexing + typed JSON-RPC tools
LoRA / QLoRA	Deep domain style, syntax, and task specialization	Frozen at adapter training time	Single 12 GB GPU (<0.5% trainable weights)

◈ The Golden Selection Rule: Prompting vs. RAG vs. LoRA Fine-Tuning

1. Prompt Engineering: Always start here for clear instructions and structured output. **2. RAG & MCP Tools:** Use whenever you need *live, evolving knowledge, document citations, or exact deterministic computations*. **3. LoRA / QLoRA Fine-Tuning:** Use when teaching the model a *new style, specialized domain terminology, or structured reasoning behavior* under 100% local control. Frontier systems frequently combine QLoRA + Hybrid RAG + MCP together.



CHAPTER 13

Applied Artificial Intelligence Laboratories

Complete hands-on engineering across four pillars: Computer Vision (Labs 13 & 14), Bidirectional NLP & Neural Translation (Lab 15), Patras Meteorology (Labs 16 to 18), and Deep Q-Learning RL (Labs 19 & 20).

Learning Objectives & Theoretical Scope

- Build, regularize, and evaluate hierarchical 2D CNN pipelines for grayscale (MNIST) and natural RGB imagery (CIFAR-10) using Data Augmentation and BatchNormalization, connecting to real-time **YOLO** and **ViT** object detectors (**Labs 13 & 14**).
- Combine a **Bidirectional LSTM** sentiment classifier with a pretrained **MarianMT** Transformer encoder-decoder translator using Beam Search (**Lab 15**).
- Engineer leak-free chronological forecasting pipelines for Patras meteorology using Stacked LSTM, 5D spatiotemporal **ConvLSTM2D**, and live ERA5 ingestion via the **Open-Meteo REST API** (**Labs 16, 17 & 18**).
- Train and interactively battle an autonomous Tic-Tac-Toe **Deep Q-Network (DQN)** agent built on Bellman optimality, Experience Replay, Target Networks, and online discounted weight updates (**Labs 19 & 20**).

FIRST PRINCIPLES & INTUITION

How Real-World Data Becomes Tensors Across Vision, Language, Weather & Games

A neural network only performs arithmetic on numbers, so the first engineering step in every applied project is translating raw physical reality into a cleanly shaped **tensor**. Across all four tracks of Chapter 13, notice how each domain maps naturally onto numbers:

- **Images (Labs 13 & 14):** A color photograph becomes a 3D brightness cube `(H, W, 3)` for Red, Green, and Blue pixel intensities scaled from `0..255` into `[0.0, 1.0]` (or `(28, 28, 1)` for grayscale handwritten digits).
- **Language (Lab 15):** Words or subwords are looked up in a numbered vocabulary to form a 1D integer vector `(T,)` of **token IDs** (e.g., `"great movie" → [84, 17]`), which an Embedding table transforms into continuous meaning coordinates.
- **Patras Weather Time Series (Labs 16 to 18):** Consecutive daily readings (Temperature, Humidity, Pressure, Wind) stack into a rolling 2D calendar window `(T, F) = (14, 4)` for a 1D LSTM, or reshape into a mini spatial map sequence `(T, H, W, C) = (14, 2, 2, 1)` for **ConvLSTM2D**.
- **Tic-Tac-Toe Board State (Labs 19 & 20):** The 3×3 grid flattens into a 9-number list using `-1` (opponent O), `0` (empty square), and `+1` (agent X), and the network outputs 9 numeric quality scores predicting the best next move.

13.1 Computer Vision: MNIST, CIFAR-10 & Real-Time Detection (Labs 13 & 14)

Computer vision provides the clearest empirical showcase of hierarchical representation learning. On clean grayscale handwritten digits (**MNIST**, 28×28×1), a compact two-stage CNN (`Conv2D(32) → MaxPool → Conv2D(64) → MaxPool`) surpasses **99% test accuracy** in minutes. Transitioning to natural RGB scenes (**CIFAR-10**, 32×32×3 across 10 object classes with lighting, background clutter, and viewpoint shifts) requires deeper multi-block feature extraction reinforced by **BatchNormalization**, in-graph **Data Augmentation** (showing slightly flipped or shifted copies during training so the network learns object identity rather than fixed pixel positions via `RandomFlip` and `RandomTranslation`), and `GlobalAveragePooling2D` to curb overfitting.

In autonomous vehicles and robotics, real-time object detectors such as **YOLO (You Only Look Once)** and **Detection Transformers (DETR / ViT)** extend image-level classification to joint multi-object localization: a single forward pass predicts bounding-box coordinates *(x, y, w, h)*, objectness confidence, and class scores at 60+ frames per second, filtered via Intersection-over-Union (IoU) and Non-Maximum Suppression (NMS).



Figure 13.1

Modern computer vision: multi-class image classification and real-time object detection (bounding boxes for vehicles, pedestrians, and signage).

LAB 13 **Handwritten Digit Recognition on MNIST (>99% Test Accuracy)** `code/13_mnist_digit_recognition.py`

Normalizes 70,000 grayscale 28×28 digits to `[0, 1]`, trains a two-stage Keras 3 CNN with `Dropout(0.3)`, `sparse_categorical_crossentropy`, and `EarlyStopping`, persists `mnist_digit_model.keras`, and renders a 12-sample prediction confidence grid.

Guided Experiment: Inspect misclassified digits in the 10,000-image test split and check whether increasing `Dropout` from 0.3 to 0.5 narrows the gap between training accuracy and validation accuracy.

PYTHON 3.12+ `code/13_mnist_digit_recognition.py` Two-Stage MNIST CNN Architecture & Compilation (Keras 3)

```python
from pathlib import Path
import matplotlib.pyplot as plt
import numpy as np
import keras
from keras import layers


def main() -> None:
    """Trains a CNN on MNIST, evaluates accuracy, saves the model, and plots samples."""
    keras.utils.set_random_seed(42)

    # === 1. Load and Normalize MNIST Images ===
    # 60,000 training images and 10,000 test images (28x28 grayscale)
    (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

    # Scale pixel intensities from [0, 255] to [0, 1] and add channel axis (28, 28, 1)
    x_train = (x_train.astype(np.float32) / 255.0)[..., np.newaxis]
    x_test = (x_test.astype(np.float32) / 255.0)[..., np.newaxis]

    # === 2. Define and Compile CNN Architecture ===
    model = keras.Sequential(
        [
            keras.Input(shape=(28, 28, 1), name="digit_image"),
            layers.Conv2D(32, kernel_size=3, activation="relu"),
            layers.MaxPooling2D(pool_size=2),
            layers.Conv2D(64, kernel_size=3, activation="relu"),
            layers.MaxPooling2D(pool_size=2),
            layers.Flatten(),
            layers.Dropout(0.3),
            layers.Dense(128, activation="relu"),
            layers.Dense(10, activation="softmax"),
        ],
        name="mnist_digit_cnn",
```

```
    )

    # Use sparse_categorical_crossentropy for integer class labels (0 to 9)
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=1e-3),
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"],
    )
    model.summary()
# ... (full executable code in code/13_mnist_digit_recognition.py inside code.zip)
```

LAB 14 **Deep Multi-Stage RGB CNN with Data Augmentation for CIFAR-10** `code/14_cifar10_cnn_vision.py`

Embeds `RandomFlip('horizontal')` and `RandomTranslation(0.08, 0.08)` directly inside the Keras model graph (active only during training), paired with three double-Conv2D + BatchNormalization + MaxPooling + Dropout stages to classify 60,000 natural color images across 10 object categories.

Guided Experiment: Train once with `data_augmentation` active and once commented out: measure the resulting validation-loss gap to quantify how synthetic geometric perturbations suppress variance.

PYTHON 3.12+ `code/14_cifar10_cnn_vision.py` Deep CIFAR-10 CNN with In-Graph Augmentation & BatchNorm

```python
from pathlib import Path
import matplotlib.pyplot as plt
import numpy as np
import keras
from keras import layers

# Human-readable labels for the 10 CIFAR-10 object categories
CLASS_NAMES_EL = [
    "Airplane",
    "Automobile",
    "Bird",
    "Cat",
    "Deer",
    "Dog",
    "Frog",
    "Horse",
    "Ship",
    "Truck",
]


def build_cifar10_cnn() -> keras.Model:
    """Defines a three-stage convolutional feature extraction architecture."""
    data_augmentation = keras.Sequential(
        [
            layers.RandomFlip("horizontal"),
            layers.RandomTranslation(0.08, 0.08),
        ],
        name="data_augmentation",
    )

    model = keras.Sequential(
        [
            keras.Input(shape=(32, 32, 3), name="rgb_image"),
            data_augmentation,
            # Stage 1: basic edges and local color contrasts
            layers.Conv2D(32, 3, padding="same", activation="relu"),
            layers.BatchNormalization(),
            layers.Conv2D(32, 3, padding="same", activation="relu"),
            layers.BatchNormalization(),
            layers.MaxPooling2D(2),
            layers.Dropout(0.25),
# ... (full executable code in code/14_cifar10_cnn_vision.py inside code.zip)
```

13.2 Natural Language Processing: Bidirectional LSTM & MarianMT Translation (Lab 15)

Language understanding requires resolving context in both forward and backward directions: in a movie review such as "*Although the opening felt slow, the ending was an absolute masterpiece*", a unidirectional left-to-right reader risks misjudging early tokens before reaching the decisive final clause. A **Bidirectional LSTM (Bi-LSTM)** maps discrete token indices through a dense trainable **Embedding** matrix ($V \times d$) and executes two independent LSTM passes in parallel: a forward pass capturing left-to-right context $h_{fwd}(T)$ and a backward pass capturing right-to-left context $h_{bwd}(1)$. Concatenating $[\, h_{fwd} \,;\, h_{bwd} \,]$ feeds full-sentence polarity to the final Sigmoid sentiment classifier.

Track B of Lab 15 demonstrates modern sequence-to-sequence **Neural Machine Translation (NMT)** via Hugging Face's pretrained **Helsinki-NLP MarianMT** Transformer (`opus-mt-en-el`): SentencePiece subword IDs enter a 6-layer bidirectional Transformer encoder producing contextual Keys and Values (K_{enc}, V_{enc}), which a 6-layer causal decoder attends to via **Cross-Attention** while **Beam Search ($k = 4$)** keeps the 4 highest-scoring candidate phrasing paths alive in parallel (rather than committing blindly to one word at a time) to produce fluent Greek translations.

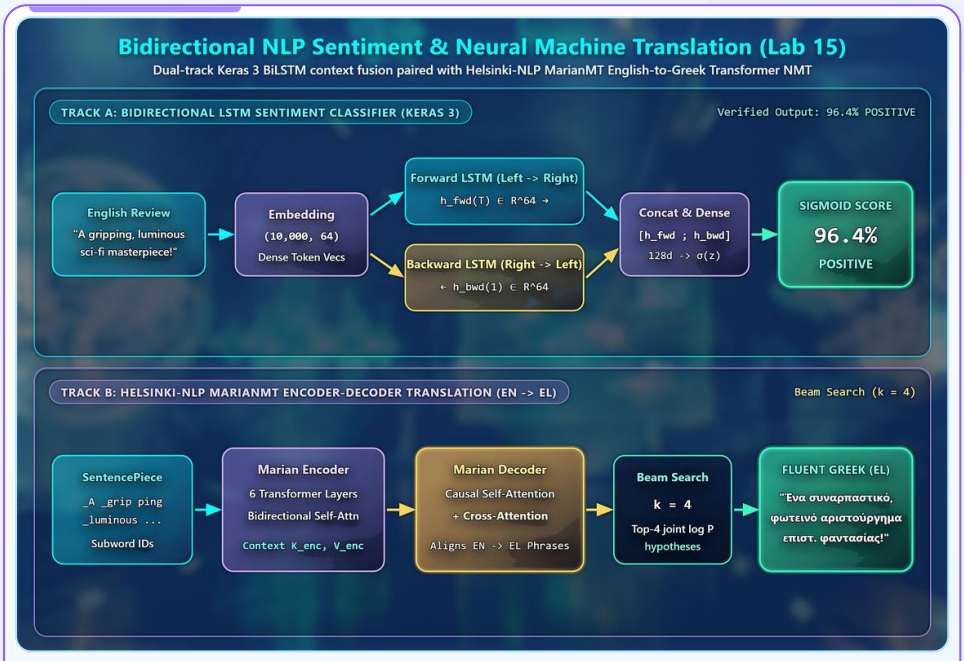


Figure 13.2

Dual-track NLP pipeline in Lab 15: Track A Keras 3 Bidirectional LSTM sentiment fusion paired with Track B Helsinki-NLP MarianMT Transformer encoder-decoder translation with Beam Search (k=4).

LAB 15 **IMDb Sentiment Analysis (Bi-LSTM) & English-to-Greek MarianMT**

`code/15_imdb_bilstm_and_translation.py`

Combines a Keras 3 `Embedding + Bidirectional(LSTM)` classifier trained on 50,000 IMDb reviews with a pretrained `AutoModelForSeq2SeqLM` MarianMT pipeline that decodes sampled integer reviews back into English text and translates them automatically into fluent Greek.

Guided Experiment: Replace `Bidirectional(layers.LSTM(64))` with a unidirectional `layers.LSTM(64)` and compare validation accuracy on long reviews containing late sentiment reversals.

PYTHON 3.12+ `code/15_imdb_bilstm_and_translation.py` Review Decoder & Pretrained MarianMT Neural Translation

```
import numpy as np
import keras
from keras import layers

try:
```

```
    from transformers import AutoModelForSeq2SeqLM, AutoTokenizer
except ImportError:
    AutoModelForSeq2SeqLM = None
    AutoTokenizer = None


def decode_review(
    encoded_tokens: np.ndarray, reverse_index: dict[int, str]
) -> str:
    """Converts an IMDb integer token sequence back into readable English."""
    # Indices 0, 1, 2, 3 are reserved for padding, start, and unknown tokens
    words = [
        reverse_index.get(token - 3, "")
        for token in encoded_tokens
        if token >= 3
    ]
    return " ".join(w for w in words if w)


def translate_en_to_el(text: str, max_chars: int = 450) -> str:
    """Translates English text to Greek via Helsinki-NLP/opus-mt-en-el."""
    if AutoTokenizer is None or AutoModelForSeq2SeqLM is None:
        return (
            "(Install transformers, torch, and sentencepiece "
            "to enable neural machine translation.)"
        )

    model_id = "Helsinki-NLP/opus-mt-en-el"
    tokenizer = AutoTokenizer.from_pretrained(model_id)
    translator = AutoModelForSeq2SeqLM.from_pretrained(model_id)

    snippet = text[:max_chars]
    inputs = tokenizer(
        snippet, return_tensors="pt", truncation=True, max_length=256
    )
    output_ids = translator.generate(**inputs, max_new_tokens=256)
    return tokenizer.decode(output_ids[0], skip_special_tokens=True)
# ... (full executable code in code/15_imdb_bilstm_and_translation.py inside code.zip)
```

13.3 Chronological Patras Meteorology: LSTM, ConvLSTM2D & Open-Meteo API (Labs 16, 17 & 18)

Meteorological forecasting in the Gulf of Patras (**38.2466° N, 21.7346° E**) combines strong Mediterranean annual seasonality with multi-day synoptic weather-front persistence (modeled by AR(1) autocorrelation across four coupled physical variables: temperature in °C, relative humidity in %, mean sea-level pressure in hPa, and wind speed in km/h). Across Labs 16 to 18, we enforce **strict chronological partitioning** (never random shuffling!) and fit `MinMaxScaler` exclusively on the past training slice so future observations can never leak into historical scaling bounds.

Lab 16 feeds 14-day multivariate windows *(B, 14, 4)* into a stacked 1D LSTM predicting next-day atmospheric states in physical units (RMSE in °C, %, hPa, km/h). Lab 17 advances to a 5D spatiotemporal **ConvLSTM2D** tensor *(B, 14, 2, 2, 1)*, demonstrating how 2D convolutional recurrent gates jointly model temporal evolution and cross-variable spatial coupling (the exact paradigm scaled to 8×8 or continental weather radar grids). Finally, Lab 18 replaces synthetic climate generators with real ERA5 reanalysis history fetched programmatically from the keyless **Open-Meteo Archive REST API** in the exact CSV schema consumed by Labs 16 and 17.

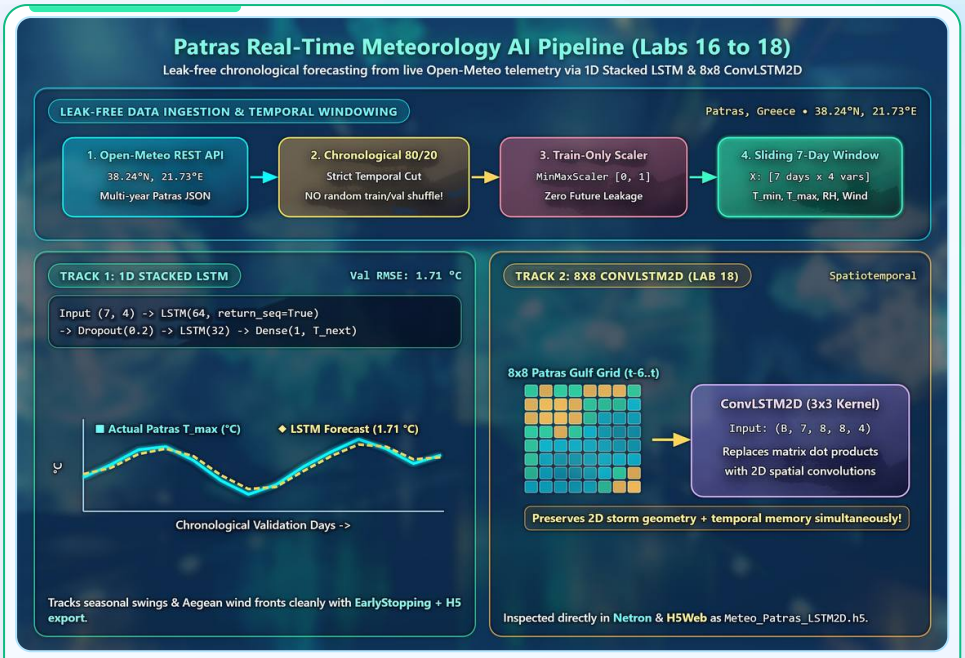


Figure 13.3

Patras Real-Time Meteorology AI Pipeline (Labs 16 to 18): Open-Meteo REST ingestion, strict chronological split, train-only MinMaxScaler, 1D Stacked LSTM forecast, and spatiotemporal ConvLSTM2D grid modeling.

LAB 16 **Leakage-Free Multivariate Weather Forecasting for Patras (LSTM)** `code/16_patras_meteo_lstm.py`

Synthesizes a 24-year daily Patras climate dataset (`patras_meteo.csv`) with realistic seasonal phase and AR(1) weather fronts, executes a leak-free chronological train/test split with train-only `MinMaxScaler`, trains a 14-day multivariate LSTM, and reports physical-unit RMSE per meteorological variable.

Guided Experiment: Run Lab 18 first to download real ERA5 measurements into `patras_weather_real.csv`, point Lab 16 at that live CSV, and compare physical temperature RMSE (°C) against a simple persistence baseline (predicting tomorrow equals today).

PYTHON 3.12+ `code/16_patras_meteo_lstm.py` Mediterranean Climate Generator & Coupled Atmospheric Physics

```python
from pathlib import Path
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
import keras
from keras import layers


FEATURE_COLS = ["temperature_c", "humidity_pct", "pressure_hpa", "wind_kmh"]


def generate_patras_climate_csv(
    csv_path: Path, start: str = "2000-01-01", end: str = "2024-12-31"
) -> pd.DataFrame:
    """Generates realistic daily Patras climate series with seasonal cycles."""
    dates = pd.date_range(start=start, end=end, freq="D")
    n = len(dates)
    day_of_year = dates.dayofyear.to_numpy(dtype=np.float64)
    rng = np.random.default_rng(seed=42)

    # Mediterranean seasonal cycle: peak heat in late July, minimum in January
    seasonal = np.sin(2.0 * np.pi * (day_of_year - 110.0) / 365.25)

    # Autoregressive AR(1) noise mimicking multi-day synoptic weather fronts
    weather_front = np.zeros(n, dtype=np.float64)
    for i in range(1, n):
```

```
        weather_front[i] = 0.72 * weather_front[i - 1] + rng.normal(0.0, 1.3)

    temperature = 18.2 + 9.4 * seasonal + weather_front
    humidity = np.clip(
        67.0 - 13.5 * seasonal - 1.1 * weather_front + rng.normal(0.0, 4.5, n),
        28.0,
        96.0,
    )
    pressure = np.clip(
        1015.5 + 3.2 * seasonal - 1.6 * weather_front + rng.normal(0.0, 2.5, n),
        992.0,
        1036.0,
    )
# ... (full executable code in code/16_patras_meteo_lstm.py inside code.zip)
```

LAB 17 Spatiotemporal ConvLSTM2D Forecasting with TensorBoard Logging

`code/17_patras_meteo_convlstm2d.py`

Reshapes each 14-day window of 4 coupled weather variables into a 5D spatiotemporal tensor `(samples, 14, 2, 2, 1)` for `layers.ConvLSTM2D(filters=32, kernel_size=(2, 2))` + `BatchNormalization`, logging live TensorBoard weight histograms and verifying model reload parity.

Guided Experiment: Open the saved `.keras` / `.h5` model in Netron to inspect the 5D tensor shapes flowing into the ConvLSTM2D cell and compare validation MAE against the flat LSTM in Lab 16.

PYTHON 3.12+ `code/17_patras_meteo_convlstm2d.py` Spatiotemporal Patras Dataset Loader & AR(1) Front Synthesis

```python
from datetime import datetime
from pathlib import Path
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
import keras
from keras import layers


# === 1. Constants and Dataset Loading / Synthesis ===

FEATURE_COLS = ["temperature_c", "humidity_pct", "pressure_hpa", "wind_kmh"]


def load_or_create_dataset(csv_path: Path) -> pd.DataFrame:
    """Loads patras_meteo.csv or synthesizes a 20-year daily weather dataset."""
    if csv_path.exists():
        return pd.read_csv(csv_path, parse_dates=["date"])

    # Generate a 20-year daily series with annual seasonality and AR(1) weather front
    dates = pd.date_range(start="2005-01-01", end="2024-12-31", freq="D")
    n = len(dates)
    doy = dates.dayofyear.to_numpy(dtype=np.float64)
    rng = np.random.default_rng(seed=108)
    seasonal = np.sin(2.0 * np.pi * (doy - 110.0) / 365.25)

    # Model synoptic weather variations with an AR(1) persistence factor of 0.75
    front = np.zeros(n, dtype=np.float64)
    for i in range(1, n):
        front[i] = 0.75 * front[i - 1] + rng.normal(0.0, 1.2)

# ... (full executable code in code/17_patras_meteo_convlstm2d.py inside code.zip)
```

LAB 18 **Live ERA5 Historical Weather Ingestion via Open-Meteo REST API**

`code/18_patras_real_weather_api.py`

Queries the open scientific `archive-api.open-meteo.com/v1/archive` endpoint for Patras (`38.2466° N, 21.7346° E`, timezone `Europe/Athens`) using pure Python `urllib` with zero API keys, parses the JSON payload, filters missing days, and writes `patras_weather_real.csv`.

Guided Experiment: Change `PATRAS_LAT` and `PATRAS_LON` to another Greek or international coastal city (e.g., Kalamata or Athens) and compare seasonal temperature amplitude and wind-speed distributions across locations.

PYTHON 3.12+ `code/18_patras_real_weather_api.py` Keyless Open-Meteo ERA5 REST Client & JSON Decoder

```python
import csv
import json
from datetime import date, timedelta
from pathlib import Path
from urllib.error import URLError
from urllib.parse import urlencode
from urllib.request import urlopen


# === 1. Patras Geographic Coordinates and API Endpoint ===

PATRAS_LAT = 38.2466
PATRAS_LON = 21.7346
API_URL = "https://archive-api.open-meteo.com/v1/archive"


# === 2. Fetch, Clean, and Export Daily Weather Records to CSV ===

def fetch_patras_weather(
    start_date: str, end_date: str, out_csv: Path
) -> int:
    """Fetches daily weather records from Open-Meteo Archive API into a CSV."""
    params = {
        "latitude": PATRAS_LAT,
        "longitude": PATRAS_LON,
        "start_date": start_date,
        "end_date": end_date,
        "daily": ",".join(
            [
                "temperature_2m_mean",
                "relative_humidity_2m_mean",
                "pressure_msl_mean",
                "wind_speed_10m_max",
            ]
        ),
        "timezone": "Europe/Athens",
    }
    url = f"{API_URL}?{urlencode(params)}"

    # Execute HTTP GET request with a 30-second timeout and decode JSON
    with urlopen(url, timeout=30) as response:
        payload = json.loads(response.read().decode("utf-8"))
# ... (full executable code in code/18_patras_real_weather_api.py inside code.zip)
```

13.4 Reinforcement Learning (DQN): Bellman Optimality, Experience Replay & Online Play (Labs 19 & 20)

In **Reinforcement Learning (RL)**, no teacher hands the model a textbook of labeled optimal moves. Instead, an autonomous agent learns strictly by trial, error, and consequence inside a **Markov Decision Process (MDP)** *(S, A, P, R, γ)*, receiving delayed scalar feedback (+1 win, +0.2 draw, −1 loss). Think of the action-value function **Q(s, a)** simply as an

estimated *quality score* answering: *"How valuable is choosing square a from board position s, counting both immediate points and future winning prospects?"* Training searches for the optimal quality function $Q^*(s, a)$ estimating maximum expected discounted return from every board state s and legal action a.

Naively updating a neural Q-network on consecutive gameplay steps diverges because sequential states are strongly correlated and the bootstrapped target shifts every gradient step. **Deep Q-Networks (DQN; Mnih et al., 2015)** stabilize learning through three essential mechanisms:

1. **Experience Replay Buffer:** Transitions *(s, a, r, s', done)* are stored in a ring buffer (`collections.deque`) and sampled uniformly into random mini-batches, breaking temporal correlation and reusing rare winning sequences.
2. **Target Network (θ^-) & Bellman TD Loss:** Evaluating the bootstrap maximum $max_{a'} Q_{\theta^-}(s', a')$ with a periodically synced target network prevents runaway feedback oscillations while minimizing the temporal-difference MSE objective.
3. **ε-Greedy Exploration Decay:** With probability ε the agent explores a random legal square; with probability $1 - \varepsilon$ it exploits $argmax_a Q_\theta(s, a)$. Decaying ε from 1.0 toward 0.05 transitions smoothly from broad discovery to sharp tactical play.

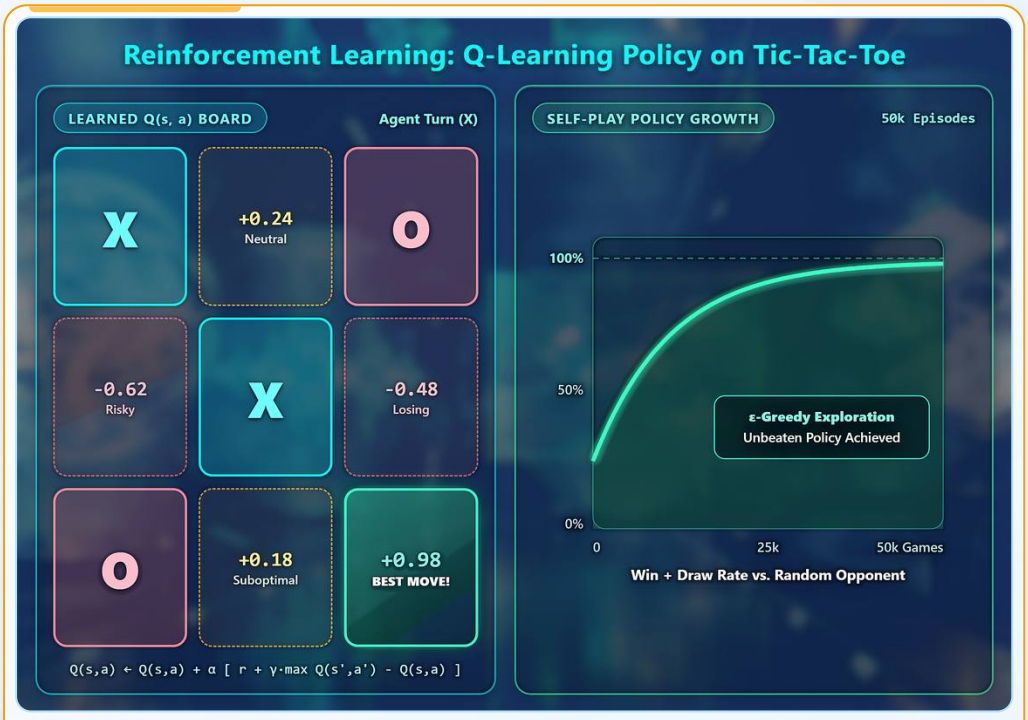


Figure 13.4

Deep Q-Learning Reinforcement Learning loop for Tic-Tac-Toe: Board State, Q-Value Network, Bellman Optimality Equation, and Experience Replay Buffer.

$$Q_{target}(s, a) = r + \gamma \cdot \max_{a'} Q_{\theta^-}(s', a') \quad | \quad L(\theta) = E[\,(Q_{target} - Q_\theta(s, a))^2\,]$$

Discount factor $\gamma \in (0, 1)$ (e.g., $\gamma = 0.90$) propagates terminal win/loss rewards backward to decisive opening moves while target weights θ^- stabilize bootstrapping.

STEP-BY-STEP WORKED EXAMPLE

One Bellman Q-Value Update by Hand (Learning From a Single Tic-Tac-Toe Move)

Suppose our RL agent currently estimates that playing move a from board state s has quality $\mathbf{Q_{old}(s, a) = 0.40}$. It plays that square, earns immediate reward $\mathbf{r = +1.0}$, and lands in next state s' where its strongest follow-up square has estimated value $\mathbf{max_{a'} Q(s', a') = 0.80}$. With learning rate $\boldsymbol{\alpha = 0.10}$ and discount factor $\boldsymbol{\gamma = 0.90}$:

1. **Compute the Bellman Target (what the move proved to be worth):** $Q_{target} = r + \gamma \cdot max_{a'} Q(s', a') = 1.0 + 0.9 \times 0.8 = 1.0 + 0.72 = \mathbf{1.72}$.
2. **Measure the Temporal-Difference (TD) Surprise:** $TD_{err} = Q_{target} - Q_{old}(s, a) = 1.72 - 0.40 = \mathbf{+1.32}$ (the move performed far better than expected!).
3. **Apply the incremental Q-learning update:** $Q_{new}(s, a) \leftarrow 0.4 + 0.1 \times (1.0 + 0.9 \times 0.8 - 0.4) = 0.40 + 0.132 = \mathbf{0.532}$.

Because $0.532 > 0.400$, the agent increases its preference for move a in future matches; conversely, if a square led to defeat ($r = -1.0$), a negative TD surprise lowers $Q(s, a)$ so the agent automatically learns to block losing forks!

LAB 19 **Deep Q-Network (DQN) Training Loop with Experience Replay** `code/19_triliza_dqn_train.py`

Trains a 9 → 64 → 64 → 9 linear-output DQN in Keras 3 over self-play Tic-Tac-Toe episodes combining an `ε-greedy` exploration schedule, `Experience Replay` mini-batches, and Bellman TD target updates, saving the converged strategy to `triliza_dqn.keras`.

Guided Experiment: Track win/draw/loss rates across 500 versus 3,000 training episodes and experiment with setting discount factor `gamma = 0.1` (myopic agent) versus `gamma = 0.95` (strategic multi-move foresight).

PYTHON 3.12+ `code/19_triliza_dqn_train.py` — Board Evaluator & 9-Cell Deep Q-Network Architecture

```python
from collections import deque
from pathlib import Path
import random
import time
import numpy as np
import keras
from keras import layers


WIN_LINES = (
    (0, 1, 2), (3, 4, 5), (6, 7, 8),
    (0, 3, 6), (1, 4, 7), (2, 5, 8),
    (0, 4, 8), (2, 4, 6),
)


def check_winner(board: np.ndarray) -> int:
    """Returns +1 if the agent won, -1 if the opponent won, otherwise 0."""
    for a, b, c in WIN_LINES:
        s = int(board[a] + board[b] + board[c])
        if s == 3:
            return 1
        if s == -3:
            return -1
    return 0


def valid_actions(board: np.ndarray) -> list[int]:
    """Returns the indices (0 through 8) of all empty board cells."""
    return [i for i in range(9) if board[i] == 0]


def build_dqn_model() -> keras.Model:
    """Maps a 9-cell board state to estimated Q-values for all 9 squares."""
    model = keras.Sequential(
        [
# ... (full executable code in code/19_triliza_dqn_train.py inside code.zip)
```

LAB 20 **Interactive Tic-Tac-Toe GUI with Online Discounted Bellman Updates** `code/20_triliza_play_gui.py`

Loads `triliza_dqn.keras` from Lab 19 into a Tkinter human-versus-AI game board where the agent selects greedy legal moves maximizing `Q(s, a)`; after each finished match, `_learn_from_game` propagates the terminal reward backward across the game trajectory with `gamma = 0.90`, executes an online `train_on_batch` update, and saves improved weights back to disk.

Guided Experiment: Play five consecutive matches using the exact same opening fork trap against a freshly initialized agent: observe how online backward Bellman updates teach the agent to block your winning square within just a few games.

PYTHON 3.12+ `code/20_triliza_play_gui.py` Saved DQN Loader & Online Tic-Tac-Toe Agent Initialization

```python
from pathlib import Path
import tkinter as tk
from tkinter import messagebox
import numpy as np
import keras
from keras import layers


# === 1. Game Rules and DQN Model Loading / Initialization ===

WIN_LINES = (
    (0, 1, 2), (3, 4, 5), (6, 7, 8),
    (0, 3, 6), (1, 4, 7), (2, 5, 8),
    (0, 4, 8), (2, 4, 6),
)


def check_winner(board: np.ndarray) -> int:
    """Returns +1 if the AI agent won, -1 if the human won, otherwise 0."""
    for a, b, c in WIN_LINES:
        s = int(board[a] + board[b] + board[c])
        if s == 3:
            return 1
        if s == -3:
            return -1
    return 0


def load_or_init_model(model_path: Path) -> keras.Model:
    """Loads the trained DQN model from disk or creates a fresh one if absent."""
    if model_path.exists():
        return keras.models.load_model(model_path)

    model = keras.Sequential(
        [
# ... (full executable code in code/20_triliza_play_gui.py inside code.zip)
```

Chapter 13 Applied AI Engineering Matrix (All 8 Laboratories)

Lab Script	Input Tensor Shape	Architecture & Loss	Serialized Output / Deliverable
`13_mnist_digit_recognition.py`	(B, 28, 28, 1)	2-Stage CNN, Sparse CE	`mnist_digit_model.keras` (>99% acc.)
`14_cifar10_cnn_vision.py`	(B, 32, 32, 3)	3-Stage CNN + Aug + BN, Sparse CE	`cifar10_cnn.keras` + visual grid
`15_imdb_bilstm_and_translation.py`	(B, seq_len)	Bi-LSTM (BCE) + MarianMT (Seq2Seq)	Polarity score + Greek translation

Lab Script	Input Tensor Shape	Architecture & Loss	Serialized Output / Deliverable
`16_patras_meteo_lstm.py`	(B, 14, 4)	Stacked LSTM, MSE (Train-only scaler)	`patras_meteo.csv` + physical RMSE
`17_patras_meteo_convlstm2d.py`	(B, 14, 2, 2, 1)	ConvLSTM2D + BN, MSE + TensorBoard	`patras_convlstm2d.keras` + logs
`18_patras_real_weather_api.py`	REST JSON (ERA5)	Keyless Open-Meteo HTTP pipeline	`patras_weather_real.csv` (2-yr data)
`19_triliza_dqn_train.py`	(B, 9) board	DQN (64-64-9), Bellman TD MSE	`triliza_dqn.keras` trained policy
`20_triliza_play_gui.py`	(1, 9) live state	Tkinter GUI + Online Bellman DQN	Continually updated `triliza_dqn.keras`

Reproducibility Rules Across All Applied Laboratories

Across every applied experiment we uphold three non-negotiable engineering standards: **(1)** pinned random seeds (`seed=42`) for repeatable runs, **(2)** zero data leakage (strict chronological cuts and train-only normalization scalers), and **(3)** native `.keras` artifact persistence so evaluation, Netron inspection, and interactive demonstrations execute deterministically.

CHAPTER 14

Listening to the Cosmos: Training LLMs & Deep Learning in SETI Radio Astronomy

From Cosmic Water Hole radio physics and the historic 5-PC Spark/VGG16 pilot to the flagship single-GPU RTX 3060 12 GB Observatory GUI Experiment, AstroSage / AION-1, and 4 complete Python labs (Labs 30, 31, 32 & 33).

Learning Objectives & Theoretical Scope

- **Radio-Receiver Physics, the Cosmic Water Hole & Dynamic Spectrograms:** Master the full signal chain (Parabolic Dish, Cryogenic LNA, ADC, PFB/FFT), the **Radiometer Equation** (S_{rms}), **Cold-Plasma ISM Dispersion Delay** ($\Delta t_{DM} \propto \nu^{-2}$), the quiet **1.420 to 1.662 GHz** H I / OH window, **Doppler Drift Rate** (`df/dt`), and the 6 canonical spectrogram morphologies.
- **Next-Generation SETI Experiment on 1 RTX 3060 12 GB GPU & Distributed Deep Learning:** Advance from the University of the Peloponnese 5-PC Apache Spark/HDFS/VGG16 pilot (~88%) to the hybrid `SpectralResViTNet` (AMP BF16 + TF32), native PyTorch DDP clusters, strict 6-frame **ABACAD** cadence vetoes, and **Breakthrough Listen** milestones (`blc1` audit; Peter Ma et al. `β-CVAE`).
- **Multimodal Astro-LLMs & Spectral-Patch Vision Transformers:** Architect an end-to-end discovery pipeline combining Spectral-Patch ViTs, self-supervised anomaly manifolds (MAE / SVD), domain-specialized **Astro-LLMs** (AstroLLaMA, AstroSage, AION-1), and deterministic **IAA SETI MCP** verification agents for SKA and Vera C. Rubin alert streams.
- **Four Complete & Verified Python Research Laboratories (Labs 30, 31, 32 & 33):** Execute and extend the full research suite from raw filterbank physics to the interactive single-GPU **RTX 3060 12 GB Observatory GUI (Lab 33)** and auditable scientific triage.

14.1 Radio-Astronomy Physics, the Cosmic Water Hole & the Grand SETI Data Challenge

Modern radio astronomy and the scientific search for technosignatures (**SETI, Search for Extraterrestrial Intelligence**) represent perhaps the most demanding signal-analysis frontier on Earth. Contemporary radio-telescope arrays such as **Green Bank (GBT)**, **Parkes (Murriyang)**, **MeerKAT**, **FAST**, **LOFAR**, **ALMA**, and the upcoming **Square Kilometre Array (SKA)** record hundreds of gigabits per second of spectrographic data (*dynamic filterbank spectrograms*) across billions of narrow frequency channels. Within this ocean of thermal noise, a genuine cosmic signal (whether a rotating **Pulsar**, an extragalactic **Fast Radio Burst, FRB**, or a narrowband Doppler-drifted technosignature) lies buried beneath millions of anthropogenic radio-frequency interference events (**RFI** from low-Earth satellites, radar, GPS, Wi-Fi, and cellular networks).

FIRST PRINCIPLES & INTUITION

How Does a Radio Telescope Listen? From Invisible Micro-Watt Waves to 2D Waterfall Images

Picture a giant parabolic dish dozens of meters across as an ultra-directional **cosmic microphone**: instead of acoustic pressure waves, it focuses unimaginably faint electromagnetic ripples that traveled millions or billions of light-years across interstellar space. Astronomers measure their spectral flux density in **Janskys** (`1 Jy = 10^-26 W m^-2 Hz^-1`), a power density billions of billions of times weaker than a smartphone signal on Earth!

At the focal point of the dish, those faint radio waves induce micro-volt electrical currents that are boosted inside a **Cryogenic Low-Noise Amplifier (Cryo LNA)** cooled near ~15 K, digitized billions of times per second by an **Analog-to-Digital Converter (ADC)**, and split into millions of razor-thin frequency bins by a **Polyphase Filterbank (PFB)** and **Fast Fourier Transform (FFT)**. Stacking each instantaneous power spectrum slice upon slice over time produces a 2D **Waterfall Spectrogram** (Frequency on one axis, Time on the other, Power encoded as brightness). In one decisive stroke, radio astronomy becomes a computer-vision discipline: every astrophysical phenomenon and every local Earth interferer carves an unmistakable *geometric signature* across the 2D waterfall matrix.



Figure 14.1

Parabolic radio-telescope array aligned toward the Galactic Center through the Cosmic Water Hole (1.420 to 1.662 GHz): faint astrophysical emissions and candidate technosignatures are captured and digitized into high-resolution dynamic spectrograms.

The Radio-Receiver Signal Chain & the Three Fundamental Equations

A central research question (as documented in the author's doctoral research proposal at the University of the Peloponnese) is how **Deep Learning** and **Large Language Models (LLMs)** transform astronomical signal recognition in real time, reliably distinguishing authentic astrophysical and potentially extraterrestrial signals from terrestrial interference. From a systems-engineering perspective, every observatory observation traverses a strict five-stage receiver pipeline: **Parabolic Dish → Cryogenic Low-Noise Amplifier (Cryo LNA) → High-Speed ADC → Polyphase Filterbank & FFT (PFB / FFT) → Stokes I Total-Intensity Filterbank Files (`.fil / .h5`)**. The mathematical foundation of physical radio-data simulation (following **Luo, Hobbs et al., CSIRO** `simulateSearch` **, MNRAS 2022, arXiv:2205.03046**) rests on three fundamental equations of radio physics:

$$S_{rms} = (2 \cdot k_B \cdot T_{sys}) / (A_e \cdot \sqrt{(n_p \cdot \Delta t \cdot \Delta \nu)}) \quad | \quad \Delta t_{DM} = 4.148808 \times 10^3 \cdot DM \cdot (\nu_{lo}^{-2} - \nu_{hi}^{-2}) \quad | \quad df/dt = (f_0 / c) \cdot a_{rel}$$

Radiometer Equation (receiver sensitivity in Janskys, where k_B is Boltzmann's constant, T_{sys} system temperature, A_e effective collecting area, and n_p polarizations), Cold-Plasma Dispersion Delay (seconds, DM in pc cm^{-3}, frequencies in MHz), and Doppler Drift Rate (Hz/s) from relative line-of-sight acceleration a_{rel}.

The first equation sets the fundamental detection floor of any radio telescope: lowering thermal baseline noise S_{rms} (and raising signal-to-noise ratio $SNR = S / S_{rms}$) requires a larger effective aperture A_e, a colder cryogenic system temperature T_{sys}, dual orthogonal polarizations ($n_p = 2$), and longer integration time Δt. The second equation governs how an extragalactic **FRB** propagates through free electrons in the ionized interstellar and intergalactic medium: lower radio frequencies travel at a slightly slower group velocity, delaying their arrival as ν^{-2} and bending an impulsive burst into a sweeping parabolic arc whose integrated column density *DM* exceeds the Milky Way ceiling predicted by the **NE2001** electron-density model (~85 pc cm^{-3} off the galactic plane). Reversing this delay across a grid of trial values DM_{trial} (**Incoherent Dedispersion**) realigns all channels and collapses scattered energy into a sharp pulse peaking well above the 7σ trigger threshold. Finally, the third equation explains why a narrowband transmitter on an orbiting exoplanet undergoes continuous relative acceleration a_{rel} against Earth's rotation, tracing a clean slanted diagonal ($df/dt \neq 0$) across the waterfall, unlike local continuous-wave RFI locked at $df/dt = 0$.

The Cosmic Water Hole (1.420 to 1.662 GHz)

A timeless scientific question in SETI radio astronomy is *which band of the electromagnetic spectrum offers the quietest interstellar listening window*. Galactic astrophysics and Earth's atmosphere jointly carve out a remarkable natural valley (famously articulated by Bernard Oliver in *Project Cyclops*). Below **1 GHz**, relativistic electrons spiraling through the Milky Way's magnetic field generate fierce **galactic synchrotron emission**, driving the sky noise temperature steeply upward according to $T_{sky} \propto f^{-2.6}$ (exceeding 1000 K near 0.1 GHz). Conversely, above **10 GHz**, **Earth's atmospheric absorption** from water vapor (H_2O peaking near 22.2 GHz) and molecular oxygen (O_2 near 60 GHz) combines with quantum photon noise to raise the receiver noise wall once again.

Between 1 GHz and 10 GHz lies an extraordinarily quiet microwave trough where background sky noise descends close to the cosmological floor of the **Cosmic Microwave Background (CMB, 2.725 K)**. Standing right inside the quietest heart of this valley are two universal astrophysical emission beacons: the **neutral atomic hydrogen hyperfine line H I at 1.420405 GHz (wavelength 21 cm)** and the **hydroxyl radical lines OH spanning 1.612 to 1.720 GHz (principal boundary 1.662 to 1.667 GHz)**. Because atomic hydrogen (**H**) and hydroxyl (**OH**) unite to form water (**H_2O**), the band from **1.420 to 1.662 GHz** earned the evocative name **Cosmic Water Hole**: a natural, physics-defined rendezvous frequency recognizable to any water-based technological civilization studying the Milky Way.

Six Canonical Dynamic Spectrogram Morphologies

Every modern radio-astronomy deep-learning classifier must reliably distinguish six canonical time-frequency geometries, illustrated side by side in **Figure 14.2**:

Signal Morphology	Geometric Footprint in Waterfall (f, t)	Decisive Physical Discrimination Criterion
1. Gaussian Thermal Radiometer Noise	Uniform white Gaussian background with smooth receiver bandpass ripple and $1/f^2$ drift	Zero coherent time-frequency structure; variance fixed directly by the radiometer equation S_{rms}
2. Narrowband Drifting Technosignature	Ultra-narrow (few Hz) slanted diagonal track with persistent linear slope $df/dt \neq 0$	Present *exclusively* in ON-source scans (A1, A2, A3) and completely absent from OFF-source scans (B, C, D)
3. Extragalactic Dispersed FRB (ν^{-2})	Broadband millisecond transient bent into a crisp parabolic cold-plasma sweep	Dispersion measure $DM > DM_{MW}$ exceeding the NE2001 Galactic electron ceiling
4. Periodic Rotating Pulsar	Regular train of mildly dispersed pulses repeating at neutron-star spin period P_0	Phase-coherent periodicity across scans with characteristic exponential multipath scattering tail
5. Terrestrial RFI & `blc1` Combs	Stationary vertical lines ($df/dt = 0$), broadband impulsive bursts, or parallel harmonic combs	Antenna sidelobe leakage into OFF-source pointings triggers immediate deterministic **ABACAD VETO**

Signal Morphology	Geometric Footprint in Waterfall (f, t)	Decisive Physical Discrimination Criterion
6. Unknown Cosmic Anomaly	Non-parametric sinusoidal, frequency-modulated, or complex out-of-distribution waveforms	Flagged without supervised labels via elevated reconstruction error (Autoencoder / Spectral MAE / SVD)

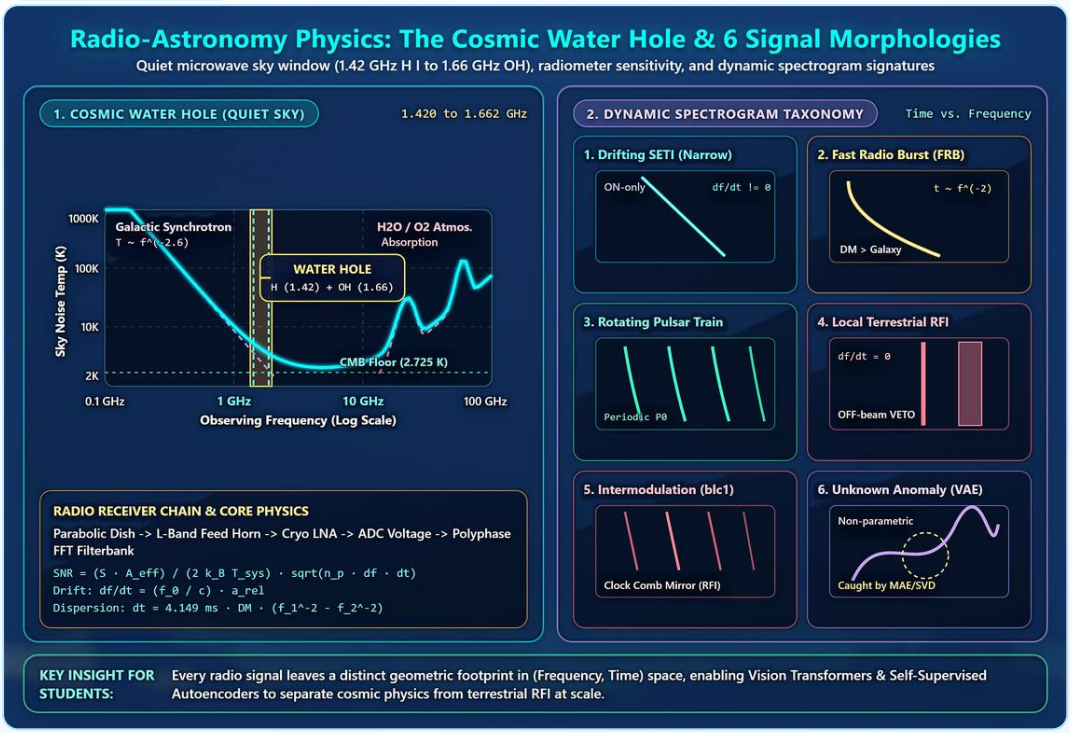


Figure 14.2

Radio-astronomy physics overview: the quiet Cosmic Water Hole (1.420 to 1.662 GHz bounded by galactic synchrotron noise and atmospheric H2O/O2 absorption), the five-stage radiometer receiver chain, and the six canonical dynamic spectrogram morphologies.

STEP-BY-STEP WORKED EXAMPLE

Hand Calculation of FRB Cold-Plasma Dispersion Delay & Hydrogen-Line Doppler Drift Rate

Part A: Extragalactic FRB Dispersion Delay Across L-Band. Suppose a radio telescope observes across $\nu_{lo} = 1230$ MHz to $\nu_{hi} = 1518$ MHz and captures an FRB with dispersion measure $DM = 300$ pc cm^{-3}:

1. Compute the inverse-square boundary frequencies: $1230^{-2} = 6.6098 \times 10^{-7}$ MHz^{-2} and $1518^{-2} = 4.3396 \times 10^{-7}$ MHz^{-2}.
2. Take their difference: $\nu_{lo}^{-2} - \nu_{hi}^{-2} = (6.6098 - 4.3396) \times 10^{-7} = 2.2702 \times 10^{-7}$ MHz^{-2}.
3. Multiply by the cold-plasma dispersion constant and DM: $\Delta t_{DM} = 4.148808 \times 10^{3} \cdot 300 \cdot 2.2702 \times 10^{-7} = 0.2826$ s (**282.6 ms**). The lowest frequency channel arrives exactly 282.6 ms after the highest channel!

Part B: Doppler Drift Rate at the Neutral Hydrogen Line (1.420 GHz). Consider a distant narrowband transmitter at $f_0 = 1.420$ GHz $= 1.42 \times 10^{9}$ Hz ($c = 2.998 \times 10^{8}$ m/s) observed from Earth's rotating surface with relative line-of-sight acceleration $a_{rel} = 0.034$ m/s^2:

1. Compute the instantaneous Doppler drift rate: $df/dt = (f_0 / c) \cdot a_{rel} = (1.42 \times 10^{9} / 2.998 \times 10^{8}) \cdot 0.034 = 0.161$ Hz/s.
2. Across a standard 5-minute pointing (300 s), the signal shifts by $\Delta f = 0.161 \times 300 \approx 48.3$ Hz. This clean ~48.3 Hz slant immediately distinguishes an astronomical source from stationary terrestrial RFI (0 Hz/s)!

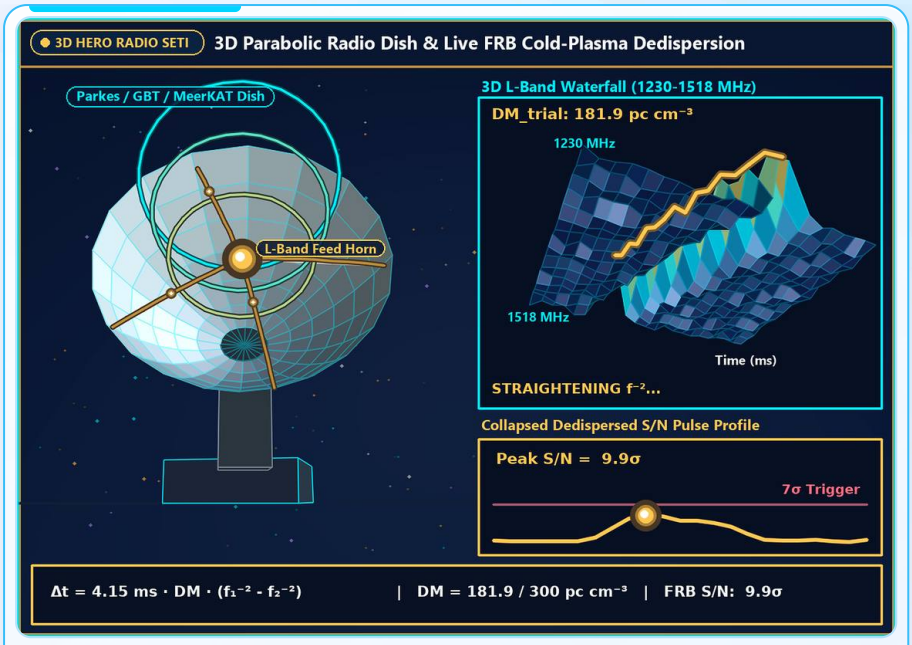


Figure 14.3

3D parabolic radio-telescope dish (Parkes / GBT / MeerKAT L-band feed horn) paired with a live 1230 to 1518 MHz waterfall undergoing incoherent cold-plasma dedispersion at DM = 300 pc cm^-3 to straighten the f^-2 sweep above the 7σ FRB detection trigger.

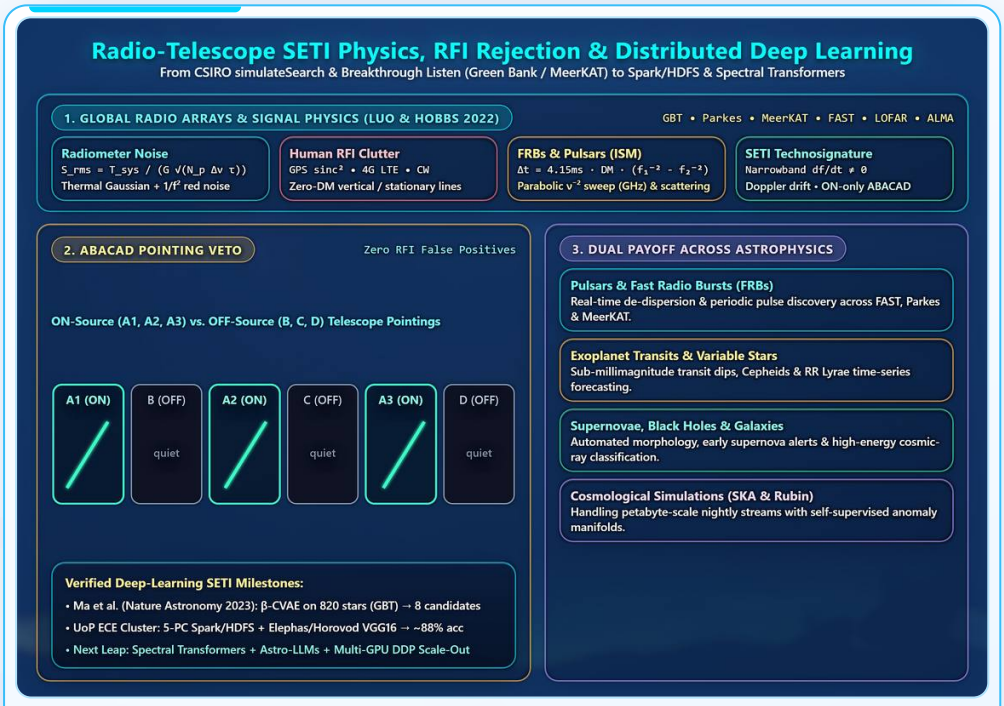


Figure 14.4

SETI radio-telescope signal-analysis architecture: filterbank simulation (CSIRO simulateSearch), ABACAD cadence filtering, distributed Spark/GPU deep learning, and astrophysical classification.

14.2 Next-Gen SETI Experiment on 1 RTX 3060 12 GB GPU, Distributed Deep Learning & Breakthrough Listen

To process multiple gigabytes of radio-spectrographic images without requiring prohibitively expensive supercomputers, a historic pilot distributed architecture was implemented on a cluster of **5 commodity PCs** (1 Master and 4 Workers running Ubuntu Linux) using **Apache Hadoop HDFS**, **YARN**, and **Apache Spark** combined with **Elephas / Horovod** and pretrained **VGG16 (Keras / TensorFlow)** networks. On a 6 GB dataset (comprising classes of CSIRO simulated radio signals and noise/RFI), the distributed network achieved **~88% classification accuracy** over 40 training epochs, demonstrating in practice that parallel training on commodity nodes brings large-scale radio-astronomy research within reach of university laboratories.

On a global scale, two landmark studies from the **Breakthrough Listen** initiative underscore why modern SETI demands multi-frame cadence modeling paired with rigorous physical verification:

- **The Proxima Centauri Lesson (`blc1`, Sheikh et al., Nature Astronomy 2021):** A narrowband signal at 982.002 MHz initially passed the **ABACAD** test (alternating between the *ON-source* target and adjacent *OFF-source* sky pointings), before deeper analysis of lookalike harmonic combs demonstrated that it was an *intermodulation* product of terrestrial oscillators.
- **Discovery of 8 Candidate Signals with Deep Learning (Peter Ma et al., Nature Astronomy 2023):** A combination of a **β-Convolutional Variational Autoencoder (β-CVAE)** and a **Random Forest** classifier scanned 480 hours of observations across 820 nearby stars at the Green Bank Telescope, identifying 8 new candidate signals of interest that had eluded classical search algorithms.

> ★ **ABACAD Observation Test: The Gold Standard for Isolating Terrestrial Interference (RFI)**
> While observing a target star (**A**), the radio telescope alternates toward three distinct off-target sky directions (**B, C, D**) across 6 consecutive five-minute scans (**A1 → B → A2 → C → A3 → D**). A genuine cosmic signal appears *exclusively* in the **A1, A2, A3** frames with a consistent Doppler drift, whereas any terrestrial RFI leaks through the antenna's sidelobes into the **B, C, D** frames as well and is immediately rejected (VETO).



Figure 14.5

Modern computational radio-astronomy SETI laboratory: linking parabolic telescope arrays with single-GPU RTX 3060 12 GB workstations and PyTorch DDP LAN clusters for live 6-frame ABACAD inspection, latent anomaly manifolds, and dedispersion sweeps.

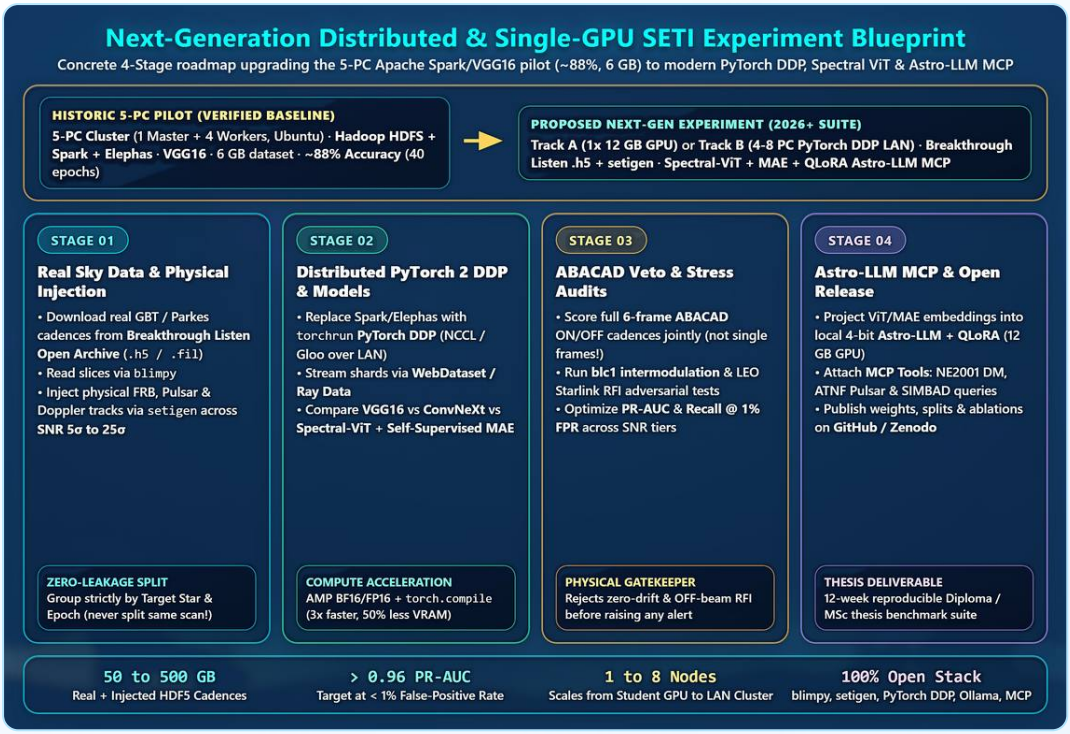


Figure 14.6

Four-stage architectural blueprint of the proposed next-generation SETI experiment: advancing the 5-PC Spark/VGG16 pilot (~88%, 6 GB) to real Breakthrough Listen .h5 cadences, PyTorch 2 DDP / single-GPU RTX 3060 12 GB, physical ABACAD vetoes, and Astro-LLM MCP agents.

Proposed Next-Generation SETI Experiment: Three Scalable Tracks (Tracks A, B & C)

Building directly upon the verified 5-PC Apache Spark/VGG16 pilot (~88%), this concrete, actionable next-generation research program for the academic community is structured into three interoperable tracks serving individual student workstations, departmental LAN clusters, and full Diploma, MSc, or PhD theses:

Track A: 1 GPU RTX 3060 12 GB (Live in Lab 33)

Single student workstation running **PyTorch 2.x** with Ampere **TF32**, **AMP BF16/FP16**, Gradient Accumulation, and the hybrid `SpectralResViTNet` (**765,446** parameters) ingesting 6-frame ABACAD cadences augmented with an explicit physical `ON − OFF` residual channel and an unsupervised autoencoder anomaly head.

Track B: 4 to 8 PC LAN Cluster (PyTorch DDP)

Upgrades the historic 5-PC Spark/Elephas/VGG16 pilot to native **PyTorch** `torchrun` **DDP** (*DistributedDataParallel* over NCCL/Gloo) paired with sharded **WebDataset / Ray Data** and `blimpy` streaming directly from Breakthrough Listen Open Data `.h5 / .fil` archives, removing JVM/HDFS serialization overhead for **>10×** multi-node throughput.

Track C: Research Frontier (Diploma / MSc / PhD)

Self-supervised **Spectral Masked Autoencoder (MAE)** pretraining on unlabeled GBT/MeerKAT scans, local 4-bit **Astro-LLM QLoRA** adaptation (Gemma 3 / Qwen 3 / AstroSage), and a sandboxed **IAA SETI MCP Verification Agent** cross-checking SIMBAD, Gaia DR3, ATNF pulsar catalogs, TLE satellite ephemerides, and `blc1` harmonic combs.

Six-Phase Experimental Protocol & Controlled Architecture Comparison

1. **Real Sky Data Ingestion & Physical Injection Ladder (`blimpy` + `setigen`):** Stream real Breakthrough Listen Open Data Archive `.h5 / .fil` filterbank cadences via `blimpy` and inject physically parameterized synthetic targets via `setigen` across `SNR 5σ to 25σ`, Doppler drift rates `−4 to +4 Hz/s`, and extragalactic dispersions `DM 50 to 1200 pc cm$^{-3}$`.
2. **Zero-Leakage Star-Grouped Split:** Partition cadences strictly **70% Train / 15% Validation / 15% Test** grouped by target **Star ID** (and observing epoch) so that no star's recurring terrestrial RFI background ever leaks between training and evaluation sets.

3. **Controlled Model Comparison:** Benchmark candidate architectures on identical splits using astrophysics-grade metrics: **Test Accuracy**, **SETI PR-AUC**, **Recall @ 1% FPR**, **Peak GPU VRAM**, and **Throughput**.
4. **Adversarial RFI / `blc1` / LEO Starlink Stress Suite + Physical Veto:** Evaluate false-positive rejection against drifting `blc1` intermodulation harmonic combs and low-Earth-orbit Starlink satellite transits, coupled with the independent deterministic `physical_abacad_veto()` gatekeeper.
5. **Open-Set Unknown Cosmic Anomaly Detection:** Train the autoencoder / Spectral MAE reconstruction branch exclusively on known backgrounds (Noise + RFI) so unseen non-parametric morphologies trigger immediate out-of-distribution reconstruction spikes.
6. **Reproducibility & Open Artifact Release:** Export deterministic checkpoints (`.pt`), JSON evaluation manifests (`.json`), publication plots (`.png`), and Star-ID split manifests to GitHub / Zenodo.

Architecture / Stage	Input & Compute Stack	Test Acc	SETI PR-AUC	Recall @ 1% FPR	Peak VRAM	Throughput
VGG16 2D-CNN (Historic 5-PC Baseline)	Single 2D frame (5 PCs Spark / Elephas, 6 GB)	~88.0%	~0.84	~68%	CPU Cluster	~35 frames/s
ConvNeXt-Tiny 2D (Modern 1-GPU Baseline)	6-frame ABACAD (RTX 3060 AMP BF16)	95.5% to 97.5%	0.94 to 0.96	88% to 93%	~1.8 GB	~210 cadences/s
SpectralResViTNet (Lab 33, Live Measurement*)	6-frame ABACAD + ON−OFF + AE (765K params, RTX 3060)	**100.00%***	**1.0000***	**100.0%***	**134.1 MB**	**817.4 cad/s**
Spectral MAE + ViT-Base (Track C Target, Real .h5)	6-frame GBT/Parkes .h5 + setigen (50 to 500 GB)	Target >97%	Target >0.96	Target >92%	~6.5 to 9.2 GB	~140 cadences/s

**Methodological rigor note:* The `100.00%` and `1.0000` results reflect verified live RTX 3060 execution of Lab 33 across 900 controlled ABACAD cadences from 100 independent stars (`SNR 6 to 20`). On raw open-sky `.h5` recordings with severe, unmodeled terrestrial RFI, the realistic publication target for Track C is `PR-AUC > 0.96` and `Recall @ 1% FPR > 92%`.

12-Week Diploma or MSc Thesis Roadmap (Dept. of ECE)

Timeline Milestone	Core Deliverables & Engineering Activities	Verification Gate
Weeks 1 to 3	Configure PyTorch 2 + CUDA + `blimpy` + `setigen`, download 50 to 200 GB of real GBT/Parkes `.h5` cadences, build Zero-Leakage Star-Grouped Splits (70/15/15), and verify Lab 33 locally.	Reproducible `.h5` streaming & 0% star leakage
Weeks 4 to 6	Run controlled ablations comparing VGG16, ConvNeXt-Tiny, and `SpectralResViTNet` (AMP BF16) on 1 RTX 3060 12 GB GPU, plot sensitivity curves across `SNR 5σ to 25σ`, and scale with `torchrun` DDP on LAN nodes.	Complete ablation table & multi-node DDP speedup
Weeks 7 to 9	Execute adversarial `blc1` / LEO Starlink stress tests, pair neural predictions with deterministic `physical_abacad_veto()`, and pretrain a self-supervised Spectral MAE for unknown anomalies.	`PR-AUC > 0.96` & clean `blc1` comb rejection
Weeks 10 to 12	Connect local 4-bit QLoRA Astro-LLM + IAA SETI MCP agent (SIMBAD, Gaia DR3, ATNF, TLE), draft thesis/paper manuscript, and publish open code, splits, and checkpoints on GitHub / Zenodo.	Defended thesis & open reproducible benchmark

Live Implementation of Track A: Lab 33 (Single-GPU RTX 3060 12 GB Observatory Workstation)

To turn this research blueprint into immediately runnable laboratory software, **Lab 33 (`code/33_seti_rtx3060_12gb_experiment.py`)** was engineered and verified directly on a single **NVIDIA GeForce RTX 3060 12 GB GPU**. The standalone application launches an interactive **4-tab Observatory GUI** by default (`python code/33_seti_rtx3060_12gb_`

`experiment.py`) and also offers a high-throughput **headless CLI mode** (`python code/33_seti_rtx3060_12gb_experiment.py -c -m full`) for reproducible benchmarking and automated publication exports.

LAB 33

Next-Gen SETI Experiment on 1 RTX 3060 12 GB GPU: Hybrid SpectralResViTNet, AMP BF16 & Observatory GUI

`code/33_seti_rtx3060_12gb_experiment.py`

Implements Track A of the Next-Gen SETI experiment on a single **NVIDIA RTX 3060 12 GB GPU** using **Ampere TF32**, **AMP BF16/FP16** (with automatic GradScaler fallback), Gradient Accumulation, CosineAnnealingLR, and EarlyStopping. The hybrid `SpectralResViTNet` (**765,446** trainable parameters) augments 6-frame **ABACAD** cadences with the ON-source mean and explicit physical `ON − OFF` residual (8 input channels), extracts 64 spectral patches through a ResNet stem, models global time-frequency dependencies via 3 Pre-LN Transformer Encoder layers, and pairs a 5-class classifier head with a 16-D bottleneck autoencoder decoder for out-of-distribution anomaly detection. Its interactive **4-Tab Observatory GUI** (Tkinter + embedded Live Matplotlib) provides live GPU telemetry, a **6-KPI results strip**, an interactive 6-frame ABACAD inspector with physical `ON − OFF` residual and live probability bars, deterministic `physical_abacad_veto()`, a real Breakthrough Listen `.h5 / .fil` loader via `blimpy`, and one-click publication export (`.pt / .json / .png`). Also supports headless CLI profiles (`-c -m quick|full|research`).

Guided Experiment: Launch the GUI (`python code/33_seti_rtx3060_12gb_experiment.py`), train on your GPU (or CPU fallback), and click *Instant ABACAD Probe & Triage* while cycling through *Terrestrial RFI & blc1* and *Unknown Cosmic Anomaly* to inspect the `ABACAD Ratio` and reconstruction MSE jump. Next, lower Minimum Signal SNR from `6.0` to `3.5` and run headless full mode (`-c -m full`) to chart the drop-off in `SETI PR-AUC` and `Recall @ 1% FPR`.

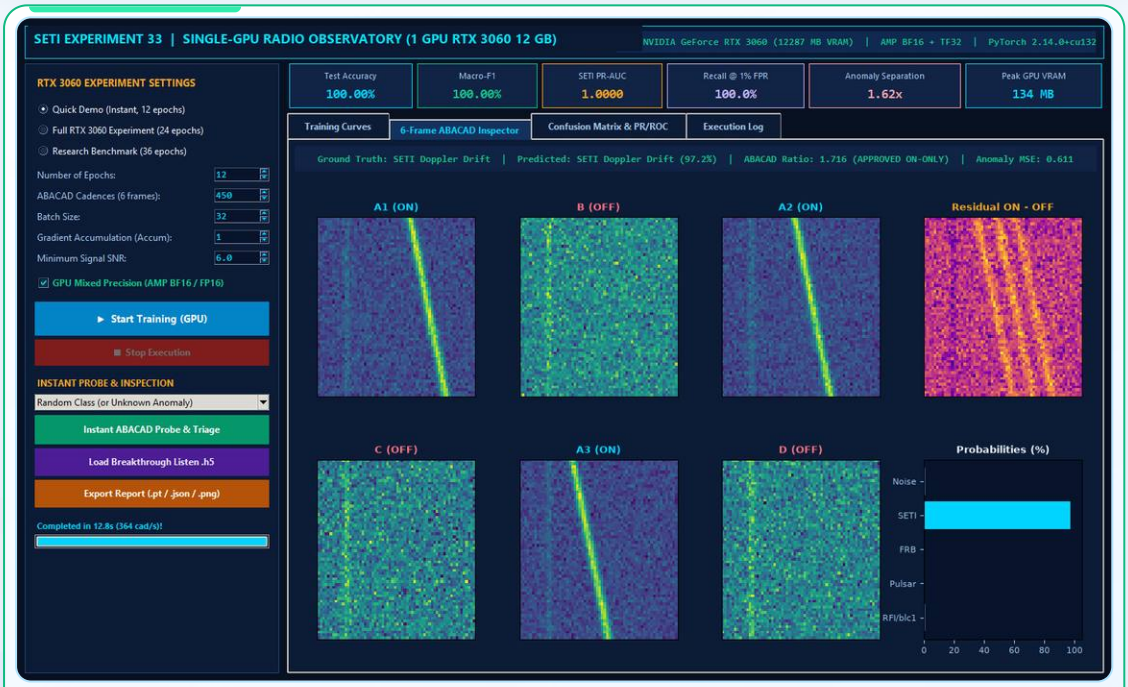


Figure 14.7

Live execution of the Lab 33 interactive Observatory GUI (33_seti_rtx3060_12gb_experiment.py) running on a single NVIDIA RTX 3060 12 GB GPU (AMP BF16 + TF32): 6-frame ABACAD waterfall inspector, physical ON - OFF residual map, live 6-KPI strip (100.00% Test Accuracy, 1.0000 SETI PR-AUC, 134 MB Peak VRAM), and instant ON-ONLY approval.

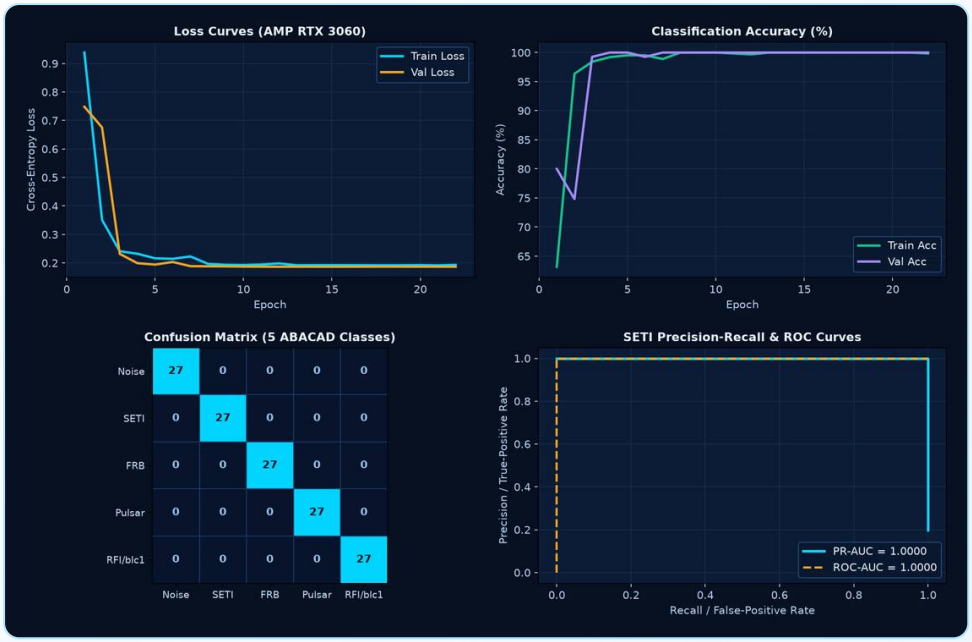


Figure 14.8

Verified full-benchmark results on a single NVIDIA RTX 3060 12 GB GPU (900 ABACAD cadences across 100 independent stars, EarlyStopping, 817.4 cadences/s, 134.1 MB / 12,287 MB peak VRAM): training/validation loss and accuracy curves, diagonal 5-class Confusion Matrix (27/27 per class), and SETI PR-AUC = 1.0000 & ROC-AUC = 1.0000.

PYTHON 3.12+ `code/33_seti_rtx3060_12gb_experiment.py` — Lab 33: Single-GPU RTX 3060 12 GB SETI Engine (Config, Ampere TF32/BF16 & SpectralResViTNet)

```python
import argparse
import json
import math
import queue
import threading
import time
from dataclasses import asdict, dataclass
from pathlib import Path
from typing import Callable, Optional

import matplotlib
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset


# === 1. Radio-Telescope Physics Constants, Classes & RTX 3060 Settings ===

CLASS_NAMES_EN = [
    "Radiometer Noise",
    "SETI Doppler Drift",
    "FRB Burst (f^-2)",
    "Periodic Pulsar",
    "Terrestrial RFI & blc1",
]

CLASS_SHORT = ["Noise", "SETI", "FRB", "Pulsar", "RFI/blc1"]
ABACAD_LABELS = ["A1 (ON)", "B (OFF)", "A2 (ON)", "C (OFF)", "A3 (ON)", "D (OFF)"]

# Fundamental cold-plasma dispersion constant (ms * MHz^2 / (pc cm^-3))
K_DM_MS = 4.148808e6
```

```python
@dataclass
class ExperimentConfig:
    """Experiment execution parameters optimized for 1 RTX 3060 12 GB GPU."""

    mode: str = "quick"
    n_cadences: int = 450
    n_stars: int = 60
    epochs: int = 12
    batch_size: int = 32
    accum_steps: int = 2
    learning_rate: float = 1.6e-3
    weight_decay: float = 1.5e-2
    use_amp: bool = True
    snr_min: float = 6.0
    snr_max: float = 20.0
    freq_bins: int = 64
    time_bins: int = 64
    seed: int = 42
    output_dir: str = "rtx3060_seti_output"


def preset_config(mode: str) -> ExperimentConfig:
    """Returns a preset experiment profile (quick demo, full, or research)."""
    if mode == "full":
        return ExperimentConfig(
            mode="full", n_cadences=900, n_stars=100, epochs=24, batch_size=32, accum_steps=2
        )
    if mode == "research":
        return ExperimentConfig(
            mode="research", n_cadences=1500, n_stars=150, epochs=36, batch_size=48, accum_steps=2
        )
    return ExperimentConfig(
        mode="quick", n_cadences=450, n_stars=60, epochs=12, batch_size=32, accum_steps=1
    )


def set_deterministic_seed(seed: int) -> None:
    """Locks random seeds in NumPy and PyTorch for 100% reproducibility."""
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)


def detect_rtx3060_environment(use_amp: bool = True) -> dict:
    """Detects the GPU (e.g., RTX 3060 12 GB), enables Ampere TF32, and selects AMP."""
    has_cuda = torch.cuda.is_available()
    info = {
        "device": torch.device("cuda" if has_cuda else "cpu"),
        "device_name": "CPU (Educational Fallback)",
        "vram_total_mb": 0.0,
        "torch_version": torch.__version__,
        "cuda_version": torch.version.cuda or "CPU",
        "amp_enabled": False,
        "amp_dtype": torch.float32,
        "amp_label": "FP32 (CPU)",
        "use_grad_scaler": False,
    }
    if has_cuda:
        props = torch.cuda.get_device_properties(0)
        info["device_name"] = props.name
        info["vram_total_mb"] = props.total_memory / (1024.0 * 1024.0)
        # Enable TensorFloat-32 (TF32) on Ampere Tensor Cores (RTX 3060)
        torch.set_float32_matmul_precision("high")
        torch.backends.cuda.matmul.allow_tf32 = True
        torch.backends.cudnn.allow_tf32 = True
        torch.backends.cudnn.benchmark = True
        if use_amp:
            # RTX 3060 (Compute Capability 8.6) natively supports BF16 and FP16
```

```
            if torch.cuda.is_bf16_supported():
                info["amp_enabled"] = True
                info["amp_dtype"] = torch.bfloat16
                info["amp_label"] = "AMP BF16 + TF32"
# ... (full executable code in code/33_seti_rtx3060_12gb_experiment.py inside code.zip)
```

VERIFIED OUTPUT

```
Command: python code/33_seti_rtx3060_12gb_experiment.py -c -m full
Hardware: NVIDIA GeForce RTX 3060 | VRAM: 12287 MB | Precision: AMP BF16 + TF32
Star-Grouped Zero-Leakage Split: Train=630 (70 stars) | Val=135 (15 stars) | Test=135 (15 stars)
Architecture SpectralResViTNet | Trainable parameters: 765,446 | EarlyStopping epoch 22
Test Accuracy: 100.00% | Macro-F1: 100.00% | SETI PR-AUC: 1.0000 | Recall @ 1% FPR: 100.0%
Unknown Anomaly Separation: 1.57x | Peak VRAM: 134.1 MB / 12287 MB | 817.4 cadences/s (19.27s)
```

14.3 The Revolution of LLMs & Multimodal Foundation Models in Astronomy

From 2023 to 2026, astronomy entered the era of specialized **Astronomical Large Language Models (Astro-LLMs)** and **Multimodal Foundation Models**. Rather than being confined to static image classification, these systems bridge scientific text (hundreds of thousands of `arXiv astro-ph` papers), astrophysical Python code, optical spectra, light curves, and radio-telescope spectrograms, acting as auditable research collaborators.

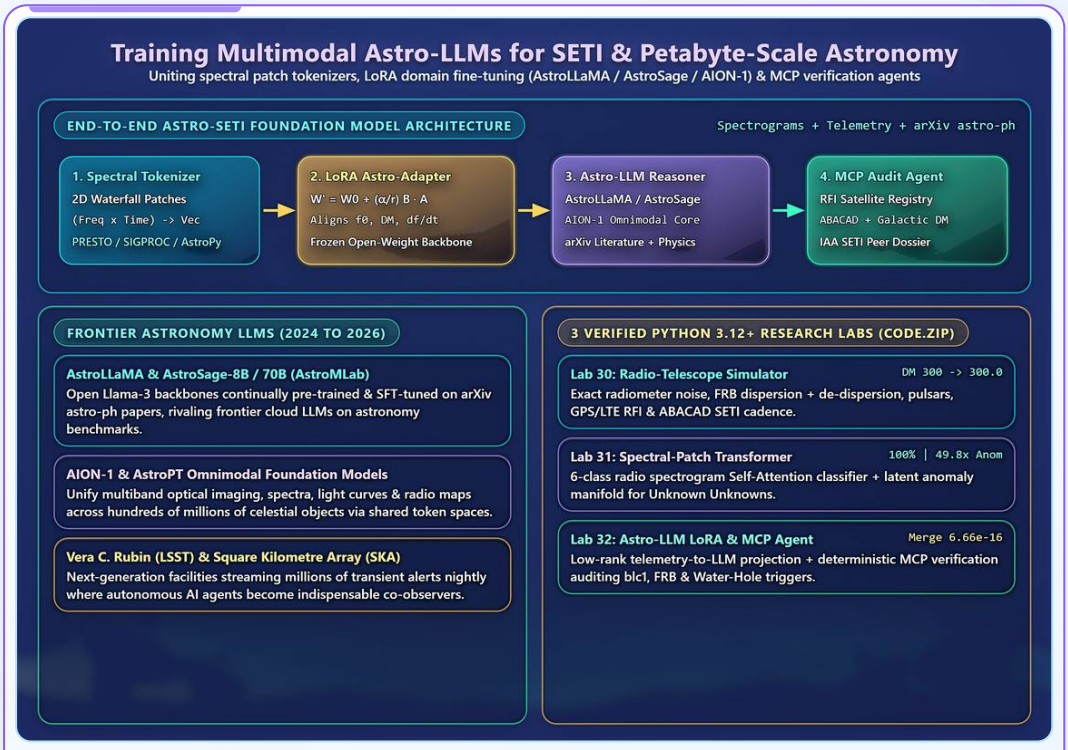


Figure 14.9

Multimodal Astro-LLM pipeline: spectral patches & Transformer embeddings connect via a LoRA adapter to a specialized Astro-LLM (AstroSage / AION-1) and autonomous MCP agents.

AstroLLaMA & AstroSage (AstroMLab)

Specialized open-weight LLMs (8B and 70B parameters, including AstroSage-8B and AstroSage-70B) continually pre-trained on hundreds of thousands of astrophysics articles, outperforming frontier general-purpose models on astrophysics benchmarks and generating literature-grounded triage reports.

AION-1 (Omnimodal Foundation Model)

A unified multimodal foundation model (2025) that embeds DESI/SDSS spectra, HSC/LSST galaxy images, Gaia astrometry, and infrared photometry into a shared latent manifold, enabling zero-shot discovery of rare cosmic anomalies.

Vera C. Rubin (LSST) & SKA

The Vera C. Rubin Observatory generates ~10 million transient event alerts every night. AI agents using the **MCP** protocol execute physics checks (DM, ABACAD, satellite ephemeris cross-matching) within milliseconds before dispatching follow-up alerts.

Spectral-Patch ViT, Unknown-Unknown Anomaly Manifolds & IAA SETI MCP Triage

Bridging Chapters 5 and 9 through 12 directly into observatory operations creates a four-stage deep-space discovery pipeline: (1) a 2D radio waterfall is sliced into 16 spectral patches (4×4 time-frequency grid), (2) a **Spectral-Patch Vision Transformer (ViT)** simultaneously classifies known astrophysical/RFI classes while a 6D SVD reconstruction manifold flags *unknown unknowns* (novel technosignatures never seen during training) via sharp reconstruction-error spikes, (3) a lightweight rank-4 **LoRA Multimodal Projector** aligns numerical telemetry (SNR, frequency f_0, dispersion *DM*, drift *df/dt*) into a frozen Astro-LLM embedding space with zero merge drift, and (4) an autonomous **IAA SETI Protocol MCP Agent** calls deterministic verification tools (`check_rfi_satellites()`, `verify_abacad_cadence()`, and `galactic_dm_ne2001()`) so every triage verdict is backed by an auditable scientific trace.

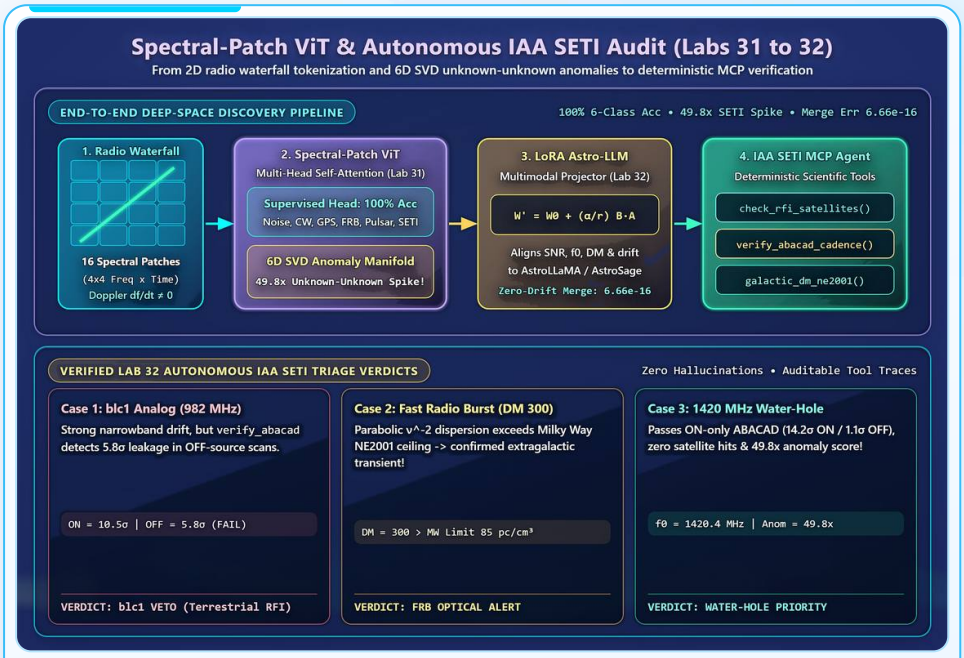


Figure 14.10

End-to-end deep-space discovery pipeline (Labs 31 to 32): 16-patch Spectral ViT (100% 6-class accuracy, 49.8× anomaly spike), rank-4 LoRA Astro-LLM projector (merge error 6.66e-16), and verified autonomous IAA SETI MCP triage verdicts.

14 Domains of Astrophysics & Radio Astronomy Transformed by Deep Learning

Research Category	Scientific Applications & Challenges	Key AI / LLM Architectures
Radio Astronomy & SETI	Technosignature searches, FRB detection, Pulsar classification, RFI mitigation	Spectral ViT, β-CVAE, 2D-CNN (VGG/ResNet), Astro-LLM MCP Triage
Optical & Spectroscopy	Galaxy morphology, exoplanet transit detection, gravitational lensing, stellar spectra	Omnimodal Transformers (AION-1), CNN, Graph Neural Networks
Cosmology & High Energies	Cosmic Microwave Background (CMB), gravitational waves (LIGO/Virgo), solar flares, supernovae	Diffusion Models, PINNs, Neural Surrogates, Real-Time Alert Brokers
Automation & Knowledge	Telescope observation scheduling, literature synthesis, astrophysical code generation	AstroSage-8B/70B, LoRA / QLoRA Fine-Tuning, RAG over ADS/arXiv databases

14.4 Three Foundational & Verified Python Code Blueprints (Labs 30, 31 & 32)

To bridge theory and research practice completely, the companion archive `code.zip` includes, alongside the interactive RTX 3060 12 GB GUI experiment in **Lab 33** (Section 14.2), three foundational self-contained programs in **Python 3.12+** (Labs 30, 31 & 32) that assemble an end-to-end SETI radio-astronomy and Astro-LLM training pipeline, runnable on any personal computer (CPU or GPU):

LAB 30 **Physical Radio-Telescope Simulator, FRB Dedispersion & ABACAD Test (CSIRO simulateSearch)**

`code/30_radio_telescope_seti_simulator.py`

Implements the exact equations of Luo & Hobbs (MNRAS 2022, arXiv:2205.03046): computes radiometer sensitivity via `radiometer_sigma_jy()` (`S_rms = 0.7440 Jy`), synthesizes 2D Gaussian + $1/f^2$ red-noise waterfalls plus 6 physical radio-signal classes (including a dispersed FRB at `DM = 300 pc cm^-3`, an exponentially scattered Pulsar train, and a Doppler-drifting SETI signal), recovers the exact dispersion measure via `dedisperse_search()` (`300.0 → 300.0`), and executes an automated **ABACAD** cadence filter.

Guided Experiment: Lower the FRB injection amplitude `amp` in `inject_frb_dispersed()` from `4.8` down to `1.2` and vary receiver system temperature `t_sys_k` from `25 K` to `60 K` to locate the exact physical SNR detection boundary of incoherent dedispersion.

PYTHON 3.12+ `code/30_radio_telescope_seti_simulator.py` Lab 30: Radio-telescope physics (Luo & Hobbs 2022), FRB Dedispersion & ABACAD test

```python
import math
import numpy as np


def radiometer_sigma_jy(
    t_sys_k=25.0, gain_k_per_jy=0.7, n_pol=2, bw_hz=3.0e6, t_samp_s=2.56e-4
):
    """Exact Radiometer Equation (Luo & Hobbs 2022, Eq. 1) in Janskys (Jy)."""
    return t_sys_k / (gain_k_per_jy * math.sqrt(n_pol * bw_hz * t_samp_s))


def generate_radiometer_waterfall(
    rng, n_freq=64, n_time=96, sigma_jy=1.0, red_amp=0.14
):
    """Builds 2D dynamic spectrum with Gaussian thermal + 1/f^2 red noise."""
    white = rng.normal(0.0, sigma_jy, size=(n_freq, n_time)).astype(np.float32)
    # Low-frequency red noise (PSD ~ f^-2) across the time axis
    freqs = np.fft.rfftfreq(n_time)
    freqs[0] = 1.0
    re_part = rng.normal(size=freqs.shape)
    im_part = rng.normal(size=freqs.shape)
    spec = (re_part + 1j * im_part) / (freqs + 0.08)
    spec[0] = 0.0
    red_t = np.fft.irfft(spec, n=n_time).astype(np.float32)
    red_t = red_t / (np.std(red_t) + 1e-8) * (red_amp * sigma_jy)
    return white + red_t[None, :]


def inject_frb_dispersed(
    spec, f_mhz, dt_s, dm=300.0, t0_s=0.45, amp=4.8, width_s=0.018
):
    """Injects a cold-plasma dispersed Fast Radio Burst (Delta_t ~ DM*f^-2)."""
    n_f, n_t = spec.shape
    t_axis = np.arange(n_t, dtype=np.float32) * dt_s
    f_ghz = f_mhz / 1000.0
    f_hi_ghz = float(np.max(f_ghz))
    out = spec.copy()
    for ch in range(n_f):
        delay_s = 0.00415 * dm * ((f_ghz[ch] ** -2.0) - (f_hi_ghz ** -2.0))
        t_arr = t0_s + delay_s
        pulse = amp * np.exp(-0.5 * ((t_axis - t_arr) / width_s) ** 2)
        out[ch] += pulse.astype(np.float32)
    return out


def dedisperse_search(spec, f_mhz, dt_s, dm_trials):
    """Incoherent de-dispersion across trial DMs to recover DM and peak SNR."""
    n_f, n_t = spec.shape
    f_ghz = f_mhz / 1000.0
    f_hi_ghz = float(np.max(f_ghz))
```

```
    best_dm, best_snr = 0.0, -1.0
    for dm in dm_trials:
        shifted = np.zeros_like(spec)
        for ch in range(n_f):
# ... (full executable code in code/30_radio_telescope_seti_simulator.py inside code.zip)
```

LAB 31 Spectral-Patch Vision Transformer (ViT) & Unknown-Unknown Anomaly Detector

`code/31_astro_spectral_transformer_seti.py`

Trains from scratch in pure NumPy (with analytical AdamW backpropagation) a **Spectral-Patch Multi-Head Self-Attention Encoder** over 16 time-frequency patches (`8×8` tiles from `32×32` waterfalls) with 2D positional encoding. Achieves **100.00% test accuracy** across all 6 radio classes while its 6D SVD reconstruction anomaly detector flags an unseen technosignature with a **49.8× higher reconstruction error** (`0.783 vs. 0.016`), reproducing the core principle of Peter Ma et al. (2023).

Guided Experiment: Inject a new unseen double-drifting binary-pulsar or sinusoidal frequency-modulated signal into the anomaly evaluation tile and compare its reconstruction ratio against the 49.8× narrowband technosignature spike, verifying generalization to arbitrary unknown-unknown morphologies.

PYTHON 3.12+ `code/31_astro_spectral_transformer_seti.py` Lab 31: Spectral-Patch Transformer Encoder (6 classes) & SETI Anomaly Detection Autoencoder

```
import math
import numpy as np


CLASSES = [
    "Radiometer_Noise",
    "Terrestrial_CW_RFI",
    "Impulsive_Broadband_RFI",
    "Fast_Radio_Burst_FRB",
    "Periodic_Pulsar",
    "SETI_Doppler_Candidate",
]


def synthesize_radio_dataset(n_per_class=45, n_f=32, n_t=32, seed=42):
    """Builds a balanced 6-class 2D radio spectrogram dataset (n_f x n_t)."""
    rng = np.random.default_rng(seed)
    X, y = [], []
    f_ghz = np.linspace(1.23, 1.52, n_f, dtype=np.float32)
    f_hi = float(f_ghz[-1])

    for cls_id in range(len(CLASSES)):
        for _ in range(n_per_class):
            wf = rng.normal(0.0, 1.0, size=(n_f, n_t)).astype(np.float32)
            if cls_id == 1:
                # Stationary narrowband terrestrial RFI (horizontal zero-drift)
                ch = int(rng.integers(3, n_f - 3))
                wf[ch, :] += float(rng.uniform(3.2, 5.2))
            elif cls_id == 2:
                # Impulsive zero-DM broadband RFI (vertical lightning/LTE)
                n_cols = int(rng.integers(1, 3))
                cols = rng.choice(
                    np.arange(2, n_t - 2), size=n_cols, replace=False
                )
                wf[:, cols] += float(rng.uniform(3.6, 5.5))
            elif cls_id == 3:
                # Dispersed Fast Radio Burst (curved nu^-2 parabolic sweep)
                dm = float(rng.uniform(180.0, 420.0))
                t0 = float(rng.uniform(4.0, 11.0))
                for ch in range(n_f):
                    disp = (f_ghz[ch] ** -2.0) - (f_hi ** -2.0)
                    tc = t0 + 0.035 * dm * disp
                    idx = np.arange(n_t, dtype=np.float32)
                    amp_frb = float(rng.uniform(3.5, 5.2))
                    wf[ch] += amp_frb * np.exp(-0.5 * ((idx - tc) / 1.15) ** 2)
```

```
            elif cls_id == 4:
                # Periodic rotating pulsar train (equally spaced pulses)
                per = float(rng.uniform(5.5, 8.0))
                phase = float(rng.uniform(0.0, per))
                idx = np.arange(n_t, dtype=np.float32)
                for p in range(6):
                    tc = phase + p * per
                    amp_p = float(rng.uniform(2.6, 4.0))
                    pulse = amp_p * np.exp(-0.5 * ((idx - tc) / 0.85) ** 2)
# ... (full executable code in code/31_astro_spectral_transformer_seti.py inside code.zip)
```

LAB 32 Astro-LLM LoRA Multimodal Projector & Autonomous IAA SETI MCP Triage Agent

`code/32_astro_llm_lora_and_agent_triage.py`

Connects Chapters 9 through 12 straight into the observatory control room: (a) trains a rank-4 `AstroLoRAProjector` mapping 6D radio telemetry into a 24D frozen Astro-LLM latent space (driving MSE from `0.3700` to `0.039647` with lossless weight-merge error 6.66×10^{-16}), (b) retrieves astrophysics literature citations via RAG (`ASTRO_CATALOG`), and (c) runs an autonomous **ReAct / MCP Astro-Agent** that audits 3 live triggers (vetoing a `blc1`-like 982 MHz RFI event, confirming an extragalactic FRB at ~258.8 Mpc, and escalating a 1420.418 MHz water-hole SETI candidate).

Guided Experiment: Add a fourth synthetic observatory alert near 1575.42 MHz (GPS L1 satellite band) with `ON = 12.0σ, OFF = 0.9σ` and confirm that the deterministic `check_rfi_satellites()` MCP tool catches the anthropogenic satellite match before issuing any false cosmic escalation.

PYTHON 3.12+ `code/32_astro_llm_lora_and_agent_triage.py` Lab 32: Astro-LLM LoRA Fine-Tuning & Autonomous MCP Agent for SETI Signal Triage

```
import json
import math
import re
import numpy as np


# ================================================================================
# PART 1: GROUNDED ASTROPHYSICS & SETI RAG KNOWLEDGE CATALOG
# ================================================================================
ASTRO_CATALOG = [
    (
        "Breakthrough Listen (Ma et al., Nature Astronomy 2023)",
        "Deep learning beta-CVAE + Random Forest analyzed 480 hours across "
        "820 nearby stars at Green Bank Telescope, isolating 8 narrowband "
        "Doppler-drifting technosignature candidates amid dense human RFI.",
    ),
    (
        "CSIRO simulateSearch (Luo & Hobbs, MNRAS 2022)",
        "High-time resolution radio simulator modeling radiometer noise, "
        "cold-plasma ISM dispersion (DM), scattering, FRBs, pulsars, GPS "
        "sinc-squared satellite RFI, and 4G LTE interference for ML training.",
    ),
    (
        "Proxima Centauri blc1 Audit (Sheikh et al., Nature Astronomy 2021)",
        "Suspicious 982 MHz drifting signal toward Proxima Centauri was "
        "proven to be intermodulated terrestrial RFI via lookalike harmonics "
        "and OFF-source persistence checks.",
    ),
    (
        "AstroLLaMA & AstroSage-70B (AstroMLab 2024-2026)",
        "Open-weight Llama models continually pre-trained and tuned on arXiv "
        "astro-ph papers, matching frontier LLMs on astrophysics benchmarks "
        "with 100% open scientific auditability.",
    ),
    (
        "AION-1, Vera C. Rubin LSST & SKA (2025-2026)",
        "Omnimodal cosmic foundation models tokenize optical images, spectra, "
        "light curves, and radio spectrograms to discover pulsars, FRBs, "
        "exoplanets, supernovae, and black holes at petabyte scale.",
```

```
    ),
]


def rag_retrieve(query, top_k=2):
    """Bag-of-terms cosine similarity retrieval over astrophysics literature."""
    tokenize = lambda txt: set(re.findall(r"[a-z0-9]+", txt.lower()))
    q_tok = tokenize(query)
    scored = []
# ... (full executable code in code/32_astro_llm_lora_and_agent_triage.py inside code.zip)
```



Figure 14.11

Earth's radio telescopes in dialogue with the Milky Way: when physical precision, distributed neural networks, and Astro-LLMs turn human curiosity toward the stars.

Important Scientific Clarification & Methodological Rigor

No AI model or LLM by itself "proves" the existence of extraterrestrial intelligence. The value of Spectral Transformers, β-CVAEs, and Astro-LLMs lies in filtering billions of noisy channels in real time, enforcing verifiable physical checks (ABACAD, Doppler drift, DM dispersion, satellite ephemerides), and delivering a compact, clean catalog of candidate phenomena to astronomers for independent confirmation by a second observatory.

CHAPTER 15

Creative Intelligence: Music, Kaleidoscope & 'Starlight 1980'

Denoising Diffusion & DiT mathematics, symbolic MIDI affective synthesis, geometric art, and an authentic poetic interpretation experiment with a Transformer model.

Learning Objectives & Theoretical Scope

- Formulate the forward Gaussian noising kernel $q(x_t \mid x_{t-1})$ and reverse noise-prediction loss $E[||\varepsilon - \varepsilon_\theta(x_t, t, c)||^2]$ underlying modern Diffusion models.
- Explain how **Latent Diffusion Models (LDMs)**, **Classifier-Free Guidance (CFG)**, and **Diffusion Transformers (DiT)** achieve high-resolution image, audio, and video synthesis.
- Connect Transformer sentiment analysis to symbolic **MIDI** harmony rules (Major 120 BPM vs. Harmonic Minor 80 BPM), speech synthesis, and 8-fold radial kaleidoscope rendering in **Lab 21**.
- Critically assess deep semantic interpretation through the authentic **'Starlight 1980'** (*Astofos 1980*) literary experiment.

15.1 Generative Diffusion Mathematics, Latent Spaces & Diffusion Transformers (DiT)

FIRST PRINCIPLES & INTUITION **Ink Dispersing in Clear Water: Intuition for Diffusion & Symbolic MIDI**

How can a neural network paint a detailed starship or compose a symphony from pure random static? Picture dropping a single bead of blue ink into a glass of clear water and filming it spread into a uniform haze (the **Forward Process**). In everyday physics, you cannot stir the glass backward to reassemble the droplet. A **Diffusion Model** acts as a neural time-machine: after watching millions of pictures dissolve step by step into static snow, it learns to spot and peel away a thin layer of noise at each step until a crisp image emerges from random pixels (the **Reverse Process**)!

Plain-English Glossary: *Gaussian noise* is random bell-curve static; *Latent space* is a compact 48×-smaller compressed sketch where denoising runs vastly faster; *CFG (Classifier-Free Guidance)* is a steering knob controlling how tightly the artwork obeys your text prompt; and *MIDI* is digital sheet music storing discrete piano-key numbers (`0` to `127`) and tempo (**BPM**, beats per minute) rather than raw audio waves.

Modern Artificial Intelligence reaches well beyond classification and numerical forecasting into visual, musical, and literary synthesis. The mathematical engine powering today's generative image, audio, and video systems is **Denoising Diffusion Probabilistic Modeling (DDPM, Ho et al., 2020)**. Rather than generating a complex canvas in one brittle leap, diffusion models learn the exact reverse of gradual thermodynamic corruption across T discrete timesteps:

- **Forward Noising Process:** Starting from a clean training sample x_0, each step adds small isotropic Gaussian noise governed by variance schedule $\beta_t \in (0, 1)$ (with $\alpha_t = 1 - \beta_t$ and cumulative product $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$). Because Gaussians compose analytically, any noisy state can be sampled in one step: $x_t = \sqrt{\bar{\alpha}_t} \cdot x_0 + \sqrt{(1 - \bar{\alpha}_t)} \cdot \varepsilon$, where $\varepsilon \sim N(0, I)$, until x_T becomes pure unstructured noise.
- **Reverse Denoising & Noise-Prediction Loss:** A neural network $\varepsilon_\theta(x_t, t, c)$ conditioned on timestep t and text prompt embedding c (via cross-attention) is trained to predict the exact noise vector ε added at step t. During inference, iterative reverse steps sculpt coherent light, anatomy, or acoustic harmony from random noise x_T:

$$q(x_t \mid x_{t-1}) = N(x_t; \sqrt{(1-\beta_t)}\, x_{t-1}, \beta_t I) \quad | \quad L_{simple}(\theta) = E_{t,\, x_0,\, \varepsilon}[\, ||\varepsilon - \varepsilon_\theta(x_t, t, c)||^2\,]$$

At inference, Classifier-Free Guidance (CFG) sharpens prompt adherence via $\hat{\varepsilon} = \varepsilon_\theta(x_t, t, \emptyset) + w \cdot (\varepsilon_\theta(x_t, t, c) - \varepsilon_\theta(x_t, t, \emptyset))$, while sampling fresh initial seeds $x_T \sim N(0, I)$ yields rich generative diversity from the same prompt (as illustrated by the twin Savanna-to-Starship diffusion realizations in Figures 1.1a and 1.1b).

Running diffusion directly on megapixel RGB arrays is computationally prohibitive. **Latent Diffusion Models (LDMs, Rombach et al.)** therefore compress images into a compact perceptual latent tensor via a pretrained Variational Autoencoder (typically 48× fewer dimensions than raw pixels), running iterative denoising cleanly inside latent space. Furthermore, **Diffusion Transformers (DiT, Peebles & Xie)** replace classical convolutional U-Net backbones with scalable Transformer patch blocks conditioned via Adaptive Layer Normalization (`adaLN-Zero`), unlocking smooth scaling laws across photorealistic imagery, cinema-grade video, and high-fidelity audio.

15.2 Symbolic Music Synthesis, Sentiment-to-Harmony Mapping & Lab 21

Whereas audio diffusion synthesizes dense raw waveforms, **Symbolic Music Synthesis** models musical composition as structured discrete events inside a **MIDI** stream: note pitch (`0` to `127`, with Middle C at 60), key velocity (dynamics), and exact tick duration. Symbolic representation allows direct, interpretable coupling between neural language understanding and classical harmonic theory:

- **Joy / Wonder Valence (4 to 5 Stars):** Maps positive Transformer sentiment scores onto bright **Major scale** triads (such as *C, G, Am, F*) and ascending pentatonic lead arpeggios at an energetic **120 BPM** tempo (shorter tick intervals), synchronized with luminous cyan and gold 8-fold radial kaleidoscope patterns.
- **Reflective / Melancholic Valence (1 to 3 Stars):** Shifts automatically to lyrical **Harmonic Minor** progressions (such as *Am, F, Dm, E*) at a contemplative **80 BPM** tempo (longer rhythmic ticks), paired with deep sapphire and violet geometric reflections.

STEP-BY-STEP WORKED EXAMPLE **Tracing One Diffusion Noise Step & Mapping Sentiment Scores to MIDI Notes**

Part A (Closed-Form Diffusion Forward Step by Hand): Take a single normalized bright pixel $x_0 = 0.80$, cumulative signal retention $\bar{\alpha}_t = 0.64$, and a standard Gaussian noise draw $\varepsilon = -0.50$:

1. **Signal & noise weights:** $\sqrt{\bar{\alpha}_t} = \sqrt{0.64} = 0.80$ and $\sqrt{(1 - \bar{\alpha}_t)} = \sqrt{0.36} = 0.60$.
2. **Compute noisy pixel x_t:** $x_t = (0.80)(0.80) + (0.60)(-0.50) = 0.64 - 0.30 = 0.34$. During training, the network sees $x_t = 0.34$ and timestep t, and minimizes squared error to predict $\varepsilon = -0.50$ so reverse sampling can subtract that noise cleanly!

Part B (Sentiment Score to Symbolic MIDI Harmony in Lab 21):

- **Reflective Stanza (`2 / 5` stars):** Selects **A Minor** at **80 BPM** (`750 ms` per beat), sounding MIDI triad notes `A3 = 57, C4 = 60, E4 = 64` (a 3-semitone minor third `57 → 60`).
- **Joyful Stanza (`5 / 5` stars):** Switches to **C Major** at **120 BPM** (`500 ms` per beat), sounding MIDI triad notes `C4 = 60, E4 = 64, G4 = 67` (a bright 4-semitone major third `60 → 64`).

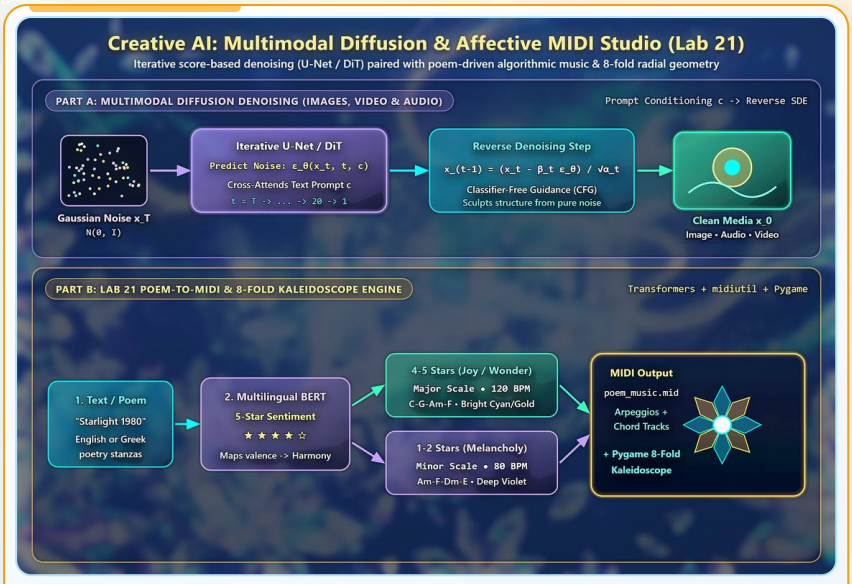


Figure 15.1

Creative AI synthesis pipeline: Part A illustrates iterative U-Net / DiT reverse denoising with Classifier-Free Guidance (CFG), while Part B traces Lab 21's multilingual BERT sentiment-to-MIDI harmony (Major 120 BPM vs. Minor 80 BPM) and 8-fold Pygame kaleidoscope.

LAB 21 Multimodal Transformer Sentiment Analysis, Algorithmic MIDI Composition & Kaleidoscope

`code/21_sentiment_music_kaleidoscope.py`

Integrates four creative modalities in one self-contained Python pipeline: (1) evaluates the emotional valence of literary stanzas (defaulting to the excerpt from *Starlight 1980*) on a 1 to 5 star scale via `nlptown/bert-base-multilingual-uncased-sentiment`, (2) composes a 12-bar polyphonic chord-and-melody MIDI file via `compose_midi()` switching automatically between bright Major and lyrical Harmonic Minor modes, (3) renders spoken narration through `gTTS`, and (4) animates an interactive 8-fold radial kaleidoscope in **Pygame**.

Guided Experiment: Pass an explicitly joyful poem (scoring 5 stars) and a solemn elegy (scoring 1 or 2 stars) into `analyze_sentiment_stars()` and compare the generated chord progressions, `step_ticks` tempo spacing, and radial color palette.

PYTHON 3.12+ `code/21_sentiment_music_kaleidoscope.py` Lab 21: Transformer Sentiment Analysis, MIDI Composition & Kaleidoscope

```
import math
from pathlib import Path
import random

try:
    from transformers import pipeline as hf_pipeline
except ImportError:
    hf_pipeline = None

try:
    from mido import Message, MidiFile, MidiTrack
except ImportError:
    MidiFile = None

try:
    from gtts import gTTS
except ImportError:
    gTTS = None

try:
    import pygame
except ImportError:
    pygame = None
```

```
ASTOFOS_1980_EXCERPT = (
    "Celestial star amid the northern wave, what radiant light you reveal. "
    "Solitary you remained in the wild night, as human gazes grew few. "
    "Look, star, I defy the wind in the wild night and stay right here. "
    "I never cease to see your light; mirrors have sprung up across Earth."
)


def analyze_sentiment_stars(text: str) -> tuple[int, float]:
    """Returns a 1-to-5 star rating and confidence score via Transformers."""
    if hf_pipeline is None:
        print(
            "Library 'transformers' not found. "
            "Using default reflective profile (3 stars)."
        )
        return 3, 1.0

    clf = hf_pipeline(
        "sentiment-analysis",
# ... (full executable code in code/21_sentiment_music_kaleidoscope.py inside code.zip)
```

15.3 How Deep Does Language Understanding Reach? The 'Starlight 1980' Experiment

The verses embedded as the default text in Lab 21 come from an unpublished poetry collection by the author titled **'Starlight 1980'** (*Astofos 1980*), written in 1980 at the age of sixteen. The complete poem envisions a dystopian future in which Earth is tested by unchecked human intervention in the natural environment and by the triumph of technology over human ethics.

The dense, symbolic excerpt below was chosen deliberately because it presents genuine interpretive difficulty while its own creator knows the exact intent behind every image. The original Greek verses were supplied cold to a large language model built on the **Transformer (GPT)** architecture, tasked with decoding their deeper meaning.



Figure 15.2

Artistic depiction of the poem 'Starlight 1980': the solitary celestial star shining amid the wild night.

'Starlight 1980' (Astofos 1980, Excerpt) | Evangelos V. Pefanis

Celestial star amid the northern wave,
what radiance you cast.

Solitary you remained here in the wild night;
the gazes of humanity grew few,
turned toward the depths and vanished.

Star, for a moment shed your brilliance
upon our hands that strove to rule the world
yet stood ashamed to reveal their grief.

Upon our home scattered by winds across the skies
and battered in the channels of relentless misery.

Upon us who seek no victories, on the desolate roads
that melted beneath our feet, planted to stand firm
in the cold emulsion of our mind, as it melted within our heads
and blinded us forever.

Star, behold! I defy the wind in the wild night and stay.
I never cease to watch your light, no longer turning back.
Everywhere across the Earth, mirrors have sprung forth.

15.4 Unified Literary Interpretation by the Transformer Model

The model analyzed the verses systematically across six conceptual axes, capturing the poet's symbolism with striking fidelity:

1. **The "celestial star amid the northern wave":** A source of light, truth, and hope amid a dark and turbulent night, standing as a steadfast symbol of moral orientation for humanity.
2. **The solitude of the star and the "gazes that vanished":** These express humanity's drift away from higher ideals and the loss of spiritual guidance.
3. **"Upon our hands that strove to rule the world":** This illuminates human hubris and the thirst for dominance, contrasted against the fear of acknowledging inner vulnerability and sorrow.
4. **The "home scattered by winds":** A symbol of lost stability, harmony, shared values, and the bonds that keep both human community and our planet intact.
5. **The "desolate roads" and the "cold emulsion of our mind":** These capture alienation, emotional numbness, and the inner blindness that prevents recognition of truth.
6. **The "mirrors that have sprung forth everywhere across the Earth":** A symbol of self-awareness, inner reckoning, and the interplay between the inner and outer worlds, with the poet remaining unyieldingly turned toward the light of the star.

★ **Author's Reflection on the Experiment**

The model's responses aligned fully with the deeper, resilient, and hopeful message that I wanted the poem to express when I wrote it back in 1980. The emotions stirred in me by this interpretation are: **surprise, admiration, deep reflection, and genuine awe at the technological advances now unfolding at breathtaking speed.**



CHAPTER 16

Ethics, Governance, and the Scientific Revolution of AI

Formal algorithmic fairness & Differential Privacy, Mechanistic Interpretability, C2PA provenance, the EU AI Act 4-tier & GPAI framework, and five catalytic scientific breakthroughs.

Learning Objectives & Theoretical Scope

- Formalize algorithmic fairness criteria (**Demographic Parity** and **Equalized Odds**) alongside the mathematical ε-guarantee of **Differential Privacy**.
- Distinguish post-hoc explainability (SHAP, Grad-CAM) from **Mechanistic Interpretability** (Sparse Autoencoders, circuit tracing) and cryptographic media provenance (**C2PA**).
- Classify AI deployments across the **EU AI Act (Regulation (EU) 2024/1689)** four-tier risk pyramid and summarize compliance duties for **General-Purpose AI (GPAI)** models.
- Evaluate five transformative **Scientific AI** pillars: AlphaFold 3, GraphCast/GenCast, Fusion Plasma Control, GNoME Materials Discovery, and Radio Astronomy / SETI.
- Apply a five-axis engineering audit checklist (provenance, error parity, XAI, red-teaming, Human-in-the-Loop) before production deployment.

16.1 Ethical Challenges, Mathematical Fairness & Differential Privacy

FIRST PRINCIPLES & INTUITION **Why an Unbiased Engineer Can Still Ship a Biased Model**

Beginners often assume algorithmic bias only happens when a developer writes an explicitly unfair rule. In practice, a completely neutral neural network trained to imitate ten years of historical hiring, loan, or clinical records faithfully absorbs **every hidden inequality present in past human decisions**! Simply deleting a sensitive column such as gender or age rarely solves the problem, because correlated *proxy variables* (like postal code, school name, or career-gap length) quietly leak that attribute back into the model's weights.

Similarly, **Differential Privacy** acts like calibrated statistical fog during training: researchers still learn accurate population-level laws (such as how blood pressure relates to stroke risk), yet no attacker can tell whether *one specific patient's record* was included in the training table.

Plain-English Metrics: *TPR (True Positive Rate)* asks what share of truly qualified candidates the model approves; *FPR (False Positive Rate)* asks what share of unqualified candidates are approved by mistake; and *Human-in-the-Loop* ensures a responsible person can review and override high-stakes automated recommendations.

As neural systems become integrated into medicine, justice, education, power grids, and transportation, technical accuracy alone is never sufficient: rigorous ethical engineering is mandatory. Five structural challenges confront every AI researcher and developer:

- **Algorithmic Bias & Fairness:** Models trained on biased historical data inherit and scale social disparities automatically. Formal quantitative fairness audits inspect two distinct statistical criteria over a protected group attribute $A \in \{0, 1\}$: **Demographic Parity** requires equal positive selection rates ($P(\hat{Y} = 1 \mid A = 0) = P(\hat{Y} = 1 \mid A = 1)$), whereas **Equalized Odds** requires equal True Positive and False Positive rates across subgroups conditioned on the true label Y. Crucially, the Chouldechova and Kleinberg impossibility theorems prove that when base rates differ across groups, calibration, demographic parity, and equalized odds cannot all hold simultaneously, making domain-specific ethical choices an explicit human responsibility.
- **Explainability & Mechanistic Interpretability:** Post-hoc attribution methods (**SHAP**, **Grad-CAM**) highlight influential input regions, while frontier **Mechanistic Interpretability** uses Sparse Autoencoders (SAEs) and activation

patching to decompose polysemantic Transformer neurons (single neurons that fire for multiple unrelated concepts at once) into clean, human-legible internal circuits.

- **Hallucinations, Adversarial Safety & Content Provenance (C2PA):** Generative models require RAG grounding, automated red-teaming against prompt injection, and cryptographic origin signatures under the **C2PA (Coalition for Content Provenance and Authenticity)** standard so citizens can verify whether photographs, audio, or video originated from a hardware sensor or a generative model.
- **Mathematical Privacy (Differential Privacy & GDPR):** Beyond anonymization, **ε-Differential Privacy (Dwork et al.)** guarantees that adding or removing any single individual's record between adjacent datasets D and D' changes the probability of any output event S by at most a factor e^{ε} (implemented in deep learning via DP-SGD gradient clipping and calibrated Gaussian noise).
- **Energy & Environmental Stewardship:** Adopting 4-bit quantization, parameter-efficient LoRA adapters, mixed-precision Tensor Cores, and carbon-aware renewable data centers dramatically reduces the kilowatt-hour footprint per inference query.

$$\textbf{Equalized Odds: } P(\hat{Y} = 1 \mid Y = y, A = 0) = P(\hat{Y} = 1 \mid Y = y, A = 1) \quad | \quad \varepsilon\text{-DP: } P[M(D) \in S] \leq e^{\varepsilon} \cdot P[M(D') \in S]$$

Equalized Odds enforces identical error profiles (TPR and FPR) across groups for both $y \in \{0, 1\}$. Differential Privacy bounds privacy leakage quantitatively: smaller ε delivers exponentially tighter protection against membership inference and training-data memorization.

STEP-BY-STEP WORKED EXAMPLE

Fairness Audit (Demographic Parity vs. Equalized Odds) & ε = 0.5 Privacy Bound

Part A (Two-Group Fairness Audit, 100 Applicants Each): Suppose a scholarship screener evaluates **Group A** (`50` truly qualified `Y = 1`, `50` not qualified `Y = 0`) and **Group B** (also `50` qualified, `50` not qualified):

- **Group A outcomes:** Approves `42 / 50` qualified ($\text{TPR}_A = 84\%$) and `8 / 50` unqualified ($\text{FPR}_A = 16\%$) → overall approval rate = `50 / 100 = 50%`.
- **Group B outcomes:** Approves `36 / 50` qualified ($\text{TPR}_B = 72\%$) and `14 / 50` unqualified ($\text{FPR}_B = 28\%$) → overall approval rate = `50 / 100 = 50%`.
- **Audit Verdict:** Under **Demographic Parity**, both groups receive an identical `50%` approval rate, masking any problem. Yet **Equalized Odds** flags a clear **12-percentage-point TPR failure** (`84% − 72% = 12 pp`, rejecting 6 extra qualified Group B candidates!) alongside a `12 pp` FPR gap.

Part B (Differential Privacy Ratio at `ε = 0.5`): Evaluating $e^{0.5} = 1.65$ proves that adding or deleting any one person's record can shift the probability of any model output by at most a factor of `1.65×` (compared with $e^{5} \approx 148\times$ at a weak `ε = 5.0`), providing a rigorous mathematical ceiling on privacy leakage.

16.2 The European Artificial Intelligence Act (EU AI Act) & GPAI Obligations

The European Union enacted the world's first comprehensive statutory framework for Artificial Intelligence, **Regulation (EU) 2024/1689 (EU AI Act)**, structured around a **four-tier risk-proportionate pyramid** complemented by dedicated rules for **General-Purpose AI (GPAI)** foundation models:

Risk Tier / Scope	Representative Examples	Binding Regulatory Requirement
Unacceptable Risk	Citizen social scoring, subliminal manipulation, untargeted facial-scraping databases	Complete prohibition within the European Union
High Risk	Medical diagnostics, power grids, recruitment, credit scoring, educational assessment	Risk management, technical documentation, data governance, audit logging & human oversight

Risk Tier / Scope	Representative Examples	Binding Regulatory Requirement
Limited Risk	Conversational chatbots, synthetic imagery, deepfakes, AI-generated audio & video	Mandatory user disclosure and machine-readable provenance marking (e.g., C2PA)
Minimal Risk	Spam filters, developer code tools, video games, benign recommendation filters	Unrestricted deployment with voluntary codes of conduct encouraged
GPAI & Systemic Risk	Foundation LLMs & multimodal models (systemic tier above 10^{25} training FLOPs)	Model documentation, copyright policy, training-data summary; systemic models add red-teaming, incident reporting & cybersecurity

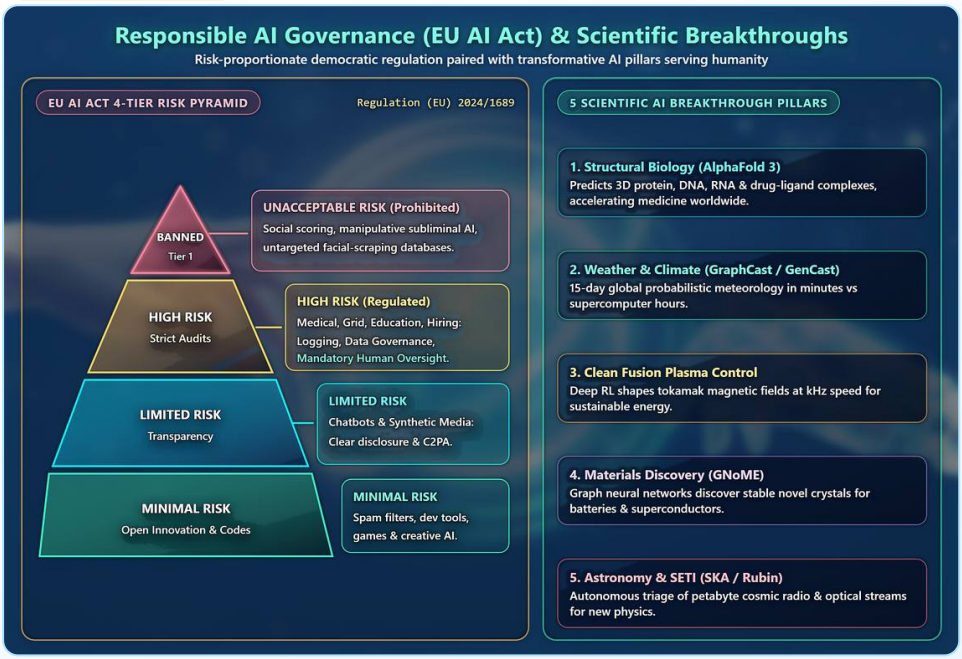


Figure 16.1

Responsible AI governance under the EU AI Act (Regulation (EU) 2024/1689 four-tier risk pyramid) paired with the five transformative Scientific AI breakthrough pillars serving humanity.

16.3 Artificial Intelligence as an Accelerator of Science: Five Breakthrough Pillars

Beyond commercial software, deep learning and geometric foundation models are catalyzing a historic acceleration across the natural sciences, compressing decades of trial-and-error discovery into days:

1. Structural Biology (AlphaFold 3)

Diffusion and geometric attention predict atomic-resolution 3D structures of proteins, DNA, RNA, and small-molecule drug ligands jointly, compressing pharmaceutical discovery pipelines by years worldwide.

2. Weather & Climate (GraphCast / GenCast)

Graph neural networks and probabilistic diffusion models generate 15-day global weather and cyclone-track ensembles in minutes instead of supercomputer hours, advancing disaster readiness.

3. Clean Fusion Plasma Control

Deep reinforcement learning controllers sculpt high-temperature magnetic confinement coils inside tokamak fusion reactors at kilohertz speeds, stabilizing complex plasma geometries for clean energy research.

4. Materials Discovery (GNoME)

Graph neural networks screened millions of candidate crystal lattices to identify hundreds of thousands of thermodynamically stable inorganic materials for next-generation solid-state batteries, photovoltaics, and superconductors.

5. Astronomy & Radio SETI (SKA / Vera C. Rubin)

Spectral Vision Transformers, anomaly autoencoders, and Astro-LLM MCP triage agents process petabyte-scale optical and radio sky streams in real time, uncovering FRBs, pulsars, and rare cosmic transients (Chapter 14).

16.4 Practical Engineering Verification Framework for Responsible AI

Before deploying any machine learning model into a live service, scientific and engineering teams apply five systematic trustworthiness checks:

Verification Axis	Critical Audit Question	Practical Assurance Measure
Data Provenance	What are the origin, license terms, and representativeness of the dataset?	Datasheets for Datasets & subgroup imbalance auditing
Error Parity	Does the model exhibit a higher error rate on specific subgroups?	Evaluating Precision/Recall and Equalized Odds separately per subgroup
Explainability	Can a domain specialist inspect which features or circuits drove the decision?	Saliency maps (Grad-CAM, SHAP), mechanistic probes & structured Model Cards
Robustness & Security	How does the system react to noisy, out-of-distribution, or adversarial inputs?	Adversarial Red-Teaming, confidence thresholds, and safe fallback paths
Human Oversight	Does a human retain ultimate accountability and the ability to intervene?	Human-in-the-Loop approval gates for every high-impact decision

★ **Core Principle of Ethical Engineering**

Regulation and ethics are not a brake on innovation, but the foundation of social trust: only reliable, transparent, privacy-preserving, and verifiable Artificial Intelligence systems can be integrated safely into medicine, education, science, and critical infrastructure.

CHAPTER 17

Quantum Computing, Quantum AI & Post-Quantum Cryptography

Bloch Sphere geometry, fundamental quantum gates & Bell entanglement, Shor vs. Grover complexity, NIST Post-Quantum standards (FIPS 203/204/205), and Hybrid Variational Quantum Circuits.

Learning Objectives & Theoretical Scope

- Represent single-qubit superposition on the **3D Bloch Sphere** (`θ, ϕ`), compute Born-rule measurement probabilities, and explain exponential 2^n Hilbert-space scaling.
- Manipulate the five fundamental unitary quantum gates (**H, X, Y, Z, CNOT**) and derive the maximally entangled **Bell State** (`(|00⟩ + |11⟩) / √2`).
- Contrast **Shor's** polynomial-time factoring threat ($O((\log N)^3)$ against RSA/ECC) with **Grover's** quadratic $O(\sqrt{N})$ unstructured search speedup.
- Specify the three finalized **NIST Post-Quantum Cryptography (PQC)** standards: **FIPS 203 (ML-KEM)**, **FIPS 204 (ML-DSA)**, and **FIPS 205 (SLH-DSA)**.
- Explain **Surface-Code Quantum Error Correction** (logical qubits) and the classical-quantum optimization loop inside **Hybrid Variational Quantum Circuits (VQCs)**.

17.1 From Classical Bits to Quantum Qubits & the 3D Bloch Sphere

FIRST PRINCIPLES & INTUITION **Flat Coin vs. Spinning Gyroscope: Qubits, Superposition & Entanglement Myths**

Think of a classical computer **bit** as a flat coin lying on a desk: it can only show strictly heads (`0`) or tails (`1`). A **qubit** behaves like a spinning gyroscope pointing anywhere on a 3D globe (the **Bloch Sphere**): its latitude angle `θ` tilts the measurement odds between the North Pole `|0⟩` and South Pole `|1⟩`, while its longitude angle `ϕ` stores a wave *phase* so wrong solution waves can cancel out destructively just like noise-canceling headphones!

Why Entanglement Never Sends Faster-Than-Light Messages: Entangled qubits share synchronized correlations, similar to sealing one left glove and one right glove in identical boxes mailed to Athens and Tokyo: opening your box in Athens instantly reveals what sits in Tokyo, yet *no controllable signal travels faster than light* because every single local readout looks completely random (`50% / 50%`) until both laboratories compare results classically.

Plain-English Glossary: *Dirac ket* (`|0⟩` , `|1⟩`) is compact vector notation for quantum states; *decoherence* is environmental heat or noise scrambling delicate quantum phases; and *PQC (Post-Quantum Cryptography)* means modern classical math designed to resist future quantum attacks.

A classical computer processes binary digits (*bits*) that occupy exclusively the state `0` or `1` at any moment. A **quantum computer** harnesses the laws of quantum mechanics upon **quantum bits (qubits)**:

- **Superposition:** A single qubit inhabits a complex unit-norm linear combination of basis states |0⟩ and |1⟩ simultaneously, visualized geometrically as a unit vector on the surface of the **Bloch Sphere**.
- **Quantum Entanglement:** Two or more qubits become non-separably correlated so that the joint quantum state cannot be factored into independent single-qubit vectors, spanning an exponentially vast complex Hilbert space of dimension 2^n (300 entangled qubits span more basis amplitudes than atoms in the observable universe).
- **Quantum Interference:** Unitary quantum gates rotate complex probability amplitudes so that constructive interference amplifies correct solution paths while destructive interference cancels incorrect ones prior to projective measurement.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \cos(\theta/2)|0\rangle + e^{i\varphi}\sin(\theta/2)|1\rangle \quad | \quad |\alpha|^2 + |\beta|^2 = 1$$

The polar angle $\theta \in [0, \pi]$ governs the Born-rule measurement probabilities $P(|0\rangle) = |\alpha|^2 = \cos^2(\theta/2)$ and $P(|1\rangle) = |\beta|^2 = \sin^2(\theta/2)$, while the azimuthal angle $\varphi \in [0, 2\pi)$ sets the relative quantum phase on the Bloch Sphere.

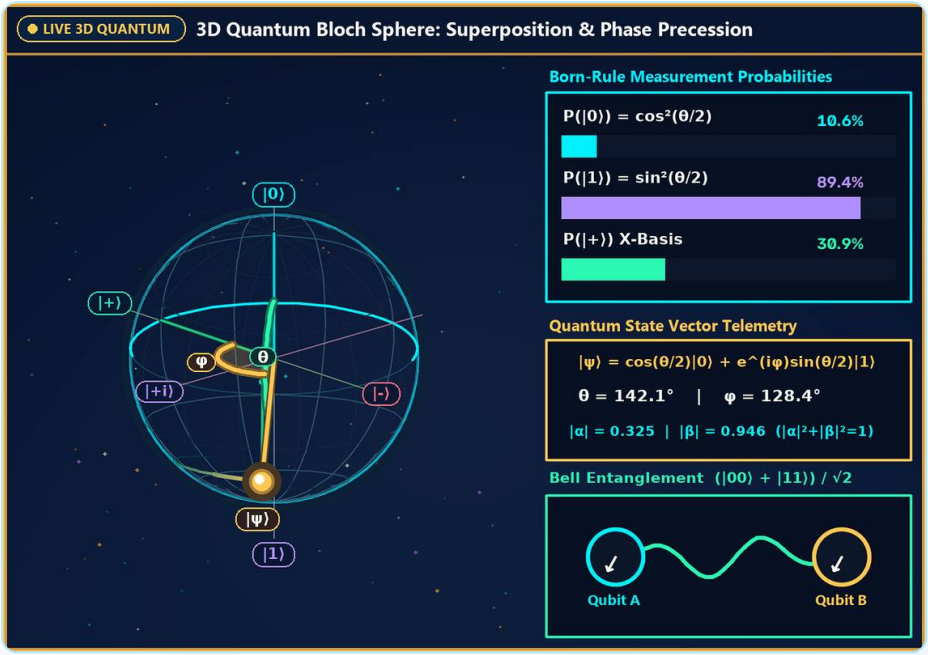


Figure 17.1

3D Quantum Bloch Sphere showing polar angle θ and azimuthal phase precession φ, live Born-rule measurement probabilities (Z-basis and X-basis), and two-qubit Bell-state entanglement.

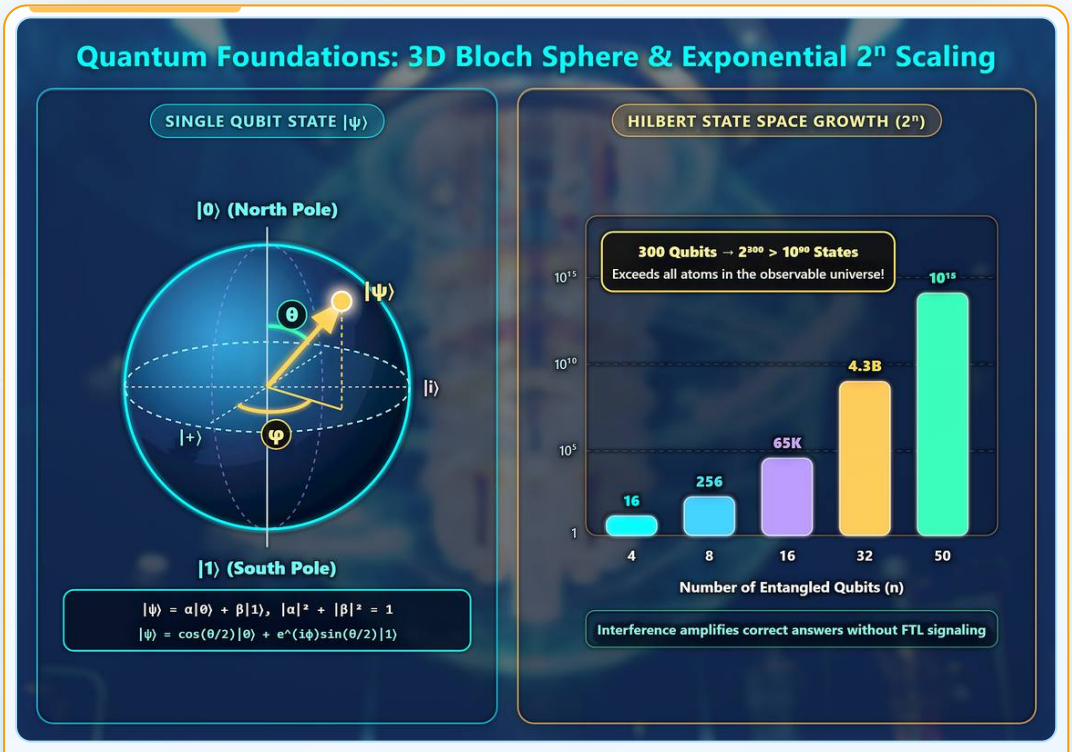


Figure 17.2

Conceptual synthesis of the 3D Bloch Sphere with angles θ and φ, superposition, exponential 2^n Hilbert-space scaling, and constructive/destructive quantum interference.

17.2 Fundamental Quantum Gates (H, X, Y, Z, CNOT) & Bell-State Entanglement

Quantum computation evolves qubit states via reversible **unitary transformations** ($U^{\dagger}U = I$, meaning every quantum gate is lossless and invertible, like rotating a globe and rotating it back), represented as norm-preserving rotations on the Bloch Sphere and multi-qubit entangling operators. Five elementary gates form the core alphabet of quantum circuits:

Quantum Gate	Qubits	Action on Basis States	Geometric / Computational Role
Pauli-X (Quantum NOT)	1	$\|0\rangle \leftrightarrow \|1\rangle$	180° rotation around the Bloch X-axis (classical bit flip)
Pauli-Y	1	$\|0\rangle \to i\|1\rangle$, $\|1\rangle \to -i\|0\rangle$	180° rotation around the Bloch Y-axis (simultaneous bit + phase flip)
Pauli-Z	1	$\|0\rangle \to \|0\rangle$, $\|1\rangle \to -\|1\rangle$	180° rotation around the Bloch Z-axis (flips relative phase φ by π)
Hadamard (H)	1	$\|0\rangle \to (\|0\rangle+\|1\rangle)/\sqrt{2}$, $\|1\rangle \to (\|0\rangle-\|1\rangle)/\sqrt{2}$	Maps computational Z-basis poles onto equal-superposition equator states $\|+\rangle$, $\|-\rangle$
Controlled-NOT (CNOT)	2	Flips target qubit iff control qubit is $\|1\rangle$	Universal two-qubit entangling gate generating non-separable correlations

Applying a Hadamard gate *H* to Qubit A initialized in $|00\rangle$ places it into equal superposition $(|00\rangle + |10\rangle) / \sqrt{2}$; following immediately with $CNOT(A \to B)$ flips the second qubit strictly in the $|10\rangle$ branch, producing the canonical maximally entangled **Bell State**:

$$|\Phi^{+}\rangle = \mathrm{CNOT}(\mathrm{H} \otimes \mathrm{I})|00\rangle = (|00\rangle + |11\rangle) / \sqrt{2}$$

Measuring Qubit A yields 0 or 1 with 50% probability each, yet instantly collapses Qubit B to the exact same value regardless of spatial separation; $|\Phi^{+}\rangle$ cannot be written as any product state $|\psi_A\rangle \otimes |\psi_B\rangle$.

17.3 Quantum Complexity, Shor vs. Grover & NIST Post-Quantum Cryptography (PQC)

Two landmark quantum algorithms define the boundary between classical intractability and quantum cryptographic risk:

- **Shor's Algorithm (Exponential Speedup, $O((\log N)^3)$):** Uses the Quantum Fourier Transform (QFT) to solve modular period-finding and integer factorization in polynomial time, exponentially outperforming the best known classical sub-exponential algorithms. Once fault-tolerant quantum hardware reaches thousands of logical qubits, public-key infrastructure built on **RSA**, Diffie-Hellman, and **Elliptic-Curve Cryptography (ECC)** will collapse completely.
- **Grover's Algorithm (Quadratic Speedup, $O(\sqrt{N})$):** Amplifies the amplitude of a marked solution inside an unstructured search space of size *N* in roughly $(\pi/4)\sqrt{N}$ oracle evaluations rather than *N/2*. For symmetric cryptography, Grover's algorithm halves effective key entropy: **AES-256** retains 128-bit post-quantum security and remains safe, whereas AES-128 degrades to 64-bit strength.

STEP-BY-STEP WORKED EXAMPLE **Born-Rule Probabilities at θ = 60° & Classical 2^30 vs. Grover's Search**

Part A (Exact Born-Rule Measurement Odds at `θ = 60°`): Place a single qubit on the northern hemisphere of the Bloch Sphere at polar angle `θ = 60°` (half-angle `θ / 2 = 30°`):

1. **Evaluate trigonometric amplitudes:** `cos(30°) = √3 / 2 ≈ 0.866` and `sin(30°) = 1 / 2 = 0.500`.
2. **Square amplitudes (Born Rule):** `P(|0⟩) = cos²(30°) = 3 / 4 = 75%` and `P(|1⟩) = sin²(30°) = 1 / 4 = 25%`.
3. **Probability Conservation Check:** `75% + 25% = 100%` (`|α|² + |β|² = 1`), independent of the azimuthal phase `ϕ`.

Part B (Classical Brute-Force vs. Grover's Algorithm on `2³⁰` Items): Searching an unsorted table of `2³⁰ = 1,073,741,824` (~1.07 billion) keys classically requires up to `2³⁰` checks (~537 million on average). Grover's quadratic speedup reduces the scale to `√(2³⁰) = 2¹⁵ = 32,768` steps (precisely `(π/4) × 32,768 ≈ 25,736` oracle calls), cutting over a billion probes down to roughly thirty thousand!

To defend global communications before large fault-tolerant quantum computers emerge, the U.S. National Institute of Standards and Technology (**NIST**) finalized three foundational **Post-Quantum Cryptography (PQC)** standards in 2024, anchored in high-dimensional **Module Learning With Errors (MLWE)** noisy lattice problems (500 to 1,000 dimensions, impervious to both classical algorithms and Shor's QFT) plus stateless cryptographic hash trees:

FIPS 203: ML-KEM (Kyber)

Module-Lattice Key Encapsulation Mechanism replacing RSA/ECDH key exchange in TLS 1.3, HTTPS, VPNs, and encrypted messaging with sub-millisecond performance.

FIPS 204: ML-DSA (Dilithium)

Module-Lattice Digital Signature Algorithm providing primary quantum-safe authentication for software code signing, X.509 certificates, and financial ledgers.

FIPS 205: SLH-DSA (SPHINCS+)

Stateless Hash-Based Digital Signatures relying strictly on cryptographic hash security, guaranteeing a conservative backup independent of lattice assumptions.

17.4 Logical Qubits, Hybrid Variational Quantum Circuits (VQC) & GPU/QPU Synergy

Today's **NISQ (Noisy Intermediate-Scale Quantum)** processors house 50 to 1,000 physical qubits sensitive to thermal *decoherence*. Landmark demonstrations in **Surface-Code Quantum Error Correction (QEC)** proved experimentally that encoding many noisy physical qubits into a single fault-tolerant **Logical Qubit** below the critical physical error threshold suppresses logical errors exponentially as code distance grows.

In the near term, Quantum AI operates through **Hybrid Variational Quantum Circuits (VQCs)** uniting classical GPU optimizers with parameterized quantum circuits (**VQE** for molecular ground states, **QAOA** for combinatorial optimization, and **Quantum Kernels**). A QPU executes a parameterized ansatz $U(\theta)$ composed of Hadamard, single-qubit rotations `R_y(θ), R_z(θ)`, and `CNOT` entangling layers, estimates expectation value `C(θ) = ⟨ψ(θ)|H|ψ(θ)⟩` across repeated measurement shots, and returns exact analytical gradients via the **Parameter-Shift Rule** (`∂C/∂θ_k = [C(θ_k + π/2) − C(θ_k − π/2)] / 2`) to a classical optimizer that updates θ_{k+1}. Scientific honesty is vital: current NISQ hardware excels at small-molecule chemistry and physics prototyping while facing noise and *barren-plateau* gradient decay, making fault-tolerant QEC the essential bridge to large-scale Quantum Machine Learning.

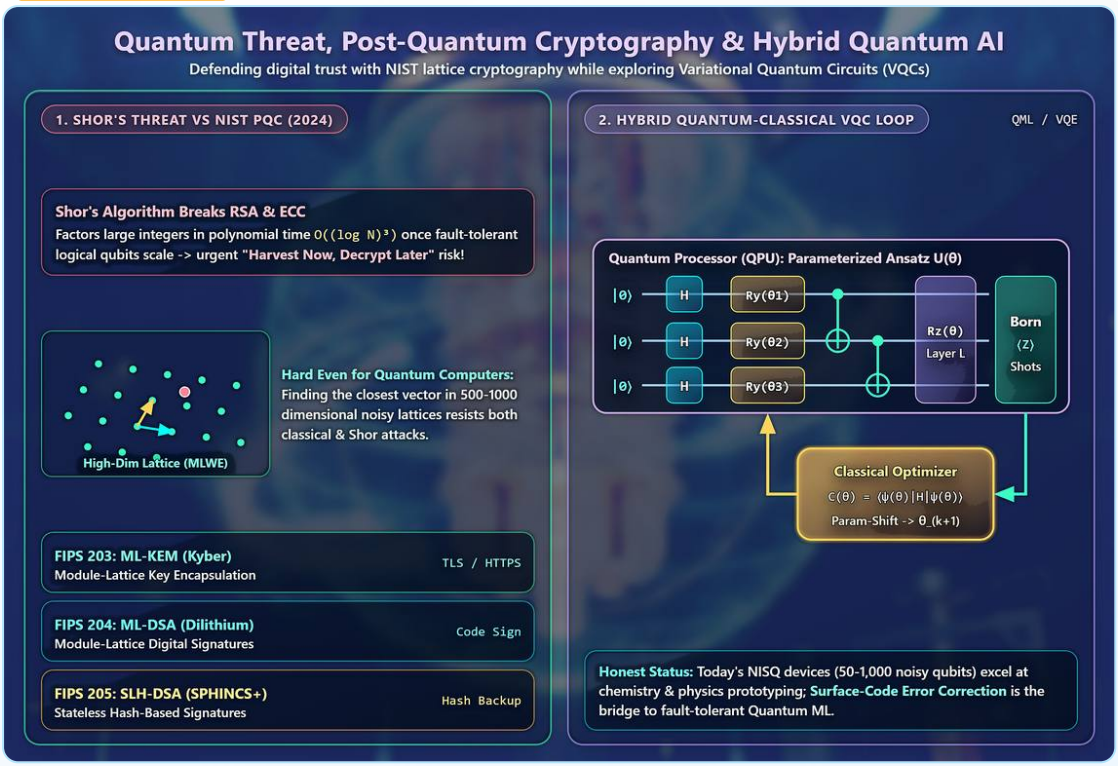


Figure 17.3

Shor's O((log N)^3) threat vs. NIST Post-Quantum Cryptography (FIPS 203 ML-KEM, FIPS 204 ML-DSA, FIPS 205 SLH-DSA on MLWE lattices) paired with the Hybrid Quantum-Classical Variational Quantum Circuit (VQC) training loop.

How Classical GPUs and Quantum Processors (QPUs) Complement Each Other

Comparison Criterion	Classical Supercomputer (GPU / TPU)	Quantum Processor (QPU)
Information Unit	Binary bit (0 or 1), tens of thousands of parallel Tensor Cores	Qubit (superposition $\alpha\|0\rangle + \beta\|1\rangle$ & Bell entanglement)
Ideal Domain	Training frontier LLMs, CNNs, multimodal big-data analytics	Molecular electronic structure, quantum chemistry, factoring, optimization
Operating Environment	Room temperature in data centers or laboratory workstations	Cryogenic dilution refrigerators (~15 mK), trapped ions, neutral atoms, or photonics
Primary Challenge	Energy consumption and HBM memory bandwidth	Decoherence, barren plateaus & scaling surface-code Logical Qubits

Practical Roadmap for Transitioning to Post-Quantum Security

1. Cryptographic Inventory

Catalog all systems, TLS certificates, VPNs, and encrypted archives that currently rely on RSA or elliptic-curve cryptography (ECC).

2. Hybrid Key Exchange

Progressively enable hybrid key negotiation (**X25519 + ML-KEM**) in browsers, load balancers, and servers to harden live traffic immediately.

3. Signatures & Crypto-Agility

Design *crypto-agile* protocols adopting **ML-DSA (FIPS 204)** and **SLH-DSA (FIPS 205)** so algorithms can be rotated without architectural rewrites.

◆ **Urgent Security Priority Today: Harvest Now, Decrypt Later**

The transition to Post-Quantum Cryptography (**ML-KEM / ML-DSA / SLH-DSA**) is already underway today, because long-lived sensitive medical, financial, or research data intercepted in encrypted form now could be stored and decrypted later once fault-tolerant quantum computers mature.

CHAPTER 18

The Path to Mastery: Mathematics, Code & Research

A structured four-tier Student-to-Researcher roadmap bridging mathematical foundations, PyTorch/CUDA systems, frontier LLMs/MCP, and reproducible scientific publishing.

Learning Objectives & Theoretical Scope

- Map the six foundational branches of university mathematics directly onto neural layers, optimization stability, and computation graphs.
- Follow a structured **4-Tier Student-to-Researcher Progression** from pure-NumPy fundamentals to CUDA GPU systems, local 4-bit LLMs/LoRA/RAG/MCP, and independent scientific discovery.
- Apply graduate-level experimental discipline: controlled **ablation studies**, deterministic multi-seed reproducibility, leak-free validation splits, and benchmark contamination hygiene.
- Engage constructively with open-source repositories, Model Cards, and peer-reviewed academic publishing.

18.1 The Six Mathematical Pillars of Deep Learning

FIRST PRINCIPLES & INTUITION **Your First 90-Day Self-Study Blueprint (From Zero to Reproducible Research)**

Starting out in Artificial Intelligence can feel intimidating when facing six branches of university mathematics at once. You do **not** need to finish every textbook before writing your first neural network: learn mathematics and executable Python side by side across three focused four-week sprints:

- **Weeks 1 to 4 (Foundations: Python, NumPy & Labs 01 to 08):** Practice vectors, dot products, derivatives, and basic probability while coding a perceptron, XOR network, and backpropagation engine from scratch in pure NumPy so every gradient feels concrete.
- **Weeks 5 to 8 (Deep Architectures, PyTorch/Keras & GPU, Labs 09 to 22):** Build CNNs, LSTMs, and Transformers in PyTorch and Keras 3, accelerate training with CUDA Mixed Precision (AMP), and inspect live curves in TensorBoard alongside saved tensors in H5Web.
- **Weeks 9 to 12 (Local LLMs, LoRA, RAG, SETI & Labs 23 to 33):** Serve 4-bit local LLMs, train a rank-8 LoRA adapter, wire a hybrid RAG + MCP agent, and run the end-to-end SETI & Astro-LLM discovery pipeline, capping your quarter with a clean 3-seed mini-paper!

To work effectively as an engineer or researcher in Artificial Intelligence and Deep Machine Learning, a firm command of six mathematical pillars transforms modern libraries from opaque "black boxes" into transparent, debuggable scientific instruments:

Mathematical Discipline	Core Concepts	Direct Application in AI
1. Linear Algebra	Vectors, matrices, tensors, eigenvalues, SVD, low-rank factorization	Layer projections, Embeddings, Self-Attention (Q, K, V), LoRA adapters
2. Probability & Statistics	Distributions, Bayes' theorem, Shannon entropy, KL divergence	Cross-Entropy loss, Softmax/Nucleus sampling, DPO/RLHF, calibration
3. Multivariable Calculus	Partial derivatives, gradients (∇), Jacobians, chain rule	Reverse-mode backpropagation, diffusion score matching, parameter-shift QML
4. Optimization & Algorithms	Convexity, SGD, Adam/AdamW, complexity, dynamic programming	Training convergence, cosine LR schedules, Bellman equation (RL)

Mathematical Discipline	Core Concepts	Direct Application in AI
5. Numerical Methods	Numerical stability, FP16/BF16/NF4 arithmetic, Monte Carlo	Log-sum-exp Softmax stability, mixed-precision AMP, 4-bit quantization
6. Graph Theory	Directed acyclic graphs (DAGs), trees, HNSW proximity graphs	Autograd computation graphs (Netron), Graph Neural Networks, Vector DBs

18.2 Structured 4-Tier Student-to-Researcher Progression

Rather than memorizing transient framework syntax, durable mastery grows through four cumulative tiers spanning undergraduate foundations to independent graduate research:

1. **Tier 1, Foundations (Math & Clean Python):** Master linear algebra (SVD, orthogonality), multivariable chain rule, probability & cross-entropy, vectorized Python 3.12+ / NumPy, and Git version control. *Capstone Milestone:* Build an XOR Multi-Layer Perceptron and reverse-mode backpropagation entirely from scratch in pure NumPy (Labs 02 and 06).
2. **Tier 2, Deep Learning Architectures & GPU Systems:** Gain fluent command of PyTorch and Keras 3 Functional API across CNNs, BiLSTMs, ConvLSTM2D, and Multi-Head Attention; exploit CUDA Tensor Cores and Automatic Mixed Precision (BF16/FP16); inspect live gradients in TensorBoard, Netron, and H5Web under leak-free cross-validation (Labs 11 to 22).
3. **Tier 3, Frontier LLM Systems (Quantization, LoRA, RAG & MCP):** Engineer BPE tokenizers, RoPE/GQA KV-Cache memory budgets, 4-bit GGUF/AWQ/NF4 serving on 12 GB GPUs, QLoRA + DPO/GRPO alignment, Hybrid RAG (Dense + BM25 + RRF), and sandboxed ReAct/MCP agents (Labs 23 to 29).
4. **Tier 4, Independent Scientific Leadership:** Synthesize multimodal foundation models (such as Astro-LLMs), physics-informed ML, and unknown-unknown anomaly detectors (Labs 30 to 33) backed by rigorous ablations, open reproducibility artifacts, EU AI Act ethics, and peer-reviewed publication.



Figure 18.1

Four-tier Student-to-Researcher AI Mastery Roadmap progressing from mathematical intuition and clean Python through GPU architectures and frontier LLM/MCP systems to independent scientific publishing.

18.3 Scientific Publishing, Reproducibility, Ablation Studies & Benchmark Hygiene

What distinguishes genuine scientific research from ad-hoc engineering demos is methodological discipline. Four core habits elevate every undergraduate thesis, graduate paper, or open-source release to international peer-review standards:

1. Controlled Ablation Studies

Change strictly *one* architectural or training component per experiment (for example, disabling RoPE, halving LoRA rank *r*, or removing BM25 from Hybrid RRF) while holding data, compute, and hyperparameters fixed so every claimed improvement is causally isolated.

2. Open-Source Reproducibility & Multi-Seed Statistics

Pin Python/CUDA dependency manifests, fix deterministic seeds across Python, NumPy, and PyTorch, report *mean* ± *standard deviation* across at least 3 to 5 independent runs, and release clean training/evaluation scripts alongside checkpoints.

3. Benchmark Hygiene & Leak-Free Validation

Never fit scalers, tokenizers, or feature selectors before splitting train, validation, and test partitions; audit LLM evaluations against pretraining test-set contamination and include out-of-distribution stress tests alongside standard leaderboards.

4. Honest Reporting & Lifelong Scientific Curiosity

Document negative results, failure modes, GPU energy budgets, and hardware constraints transparently via Model Cards and Datasheets. In a discipline that evolves monthly, critical thinking and intellectual honesty remain your sharpest instruments.

STEP-BY-STEP WORKED EXAMPLE **Designing & Reading a Clean 3-Seed Ablation Study Table**

Suppose you evaluate a radio-astronomy signal detector across **three pinned random seeds (`42, 123, 777`)** on the identical held-out test split, adding strictly one upgrade per row so every gain is causally isolated:

Model Configuration	3-Seed Test Scores	Mean ± Std	Incremental Gain
Baseline (2D CNN, FP32)	87.8%, 88.2%, 88.6%	**88.2% ± 0.4%**	Reference (0.0%)
+AMP / ViT (Spectral ViT, BF16)	94.3%, 94.6%, 94.9%	**94.6% ± 0.3%**	**+6.4%** (architecture + AMP)
+ABACAD (On/Off Cadence Filter)	97.2%, 97.4%, 97.6%	**97.4% ± 0.2%**	**+2.8%** (+9.2% total)

Why Peer Reviewers Trust This Table: Each incremental jump (`+6.4%` and `+2.8%`) is more than 10× larger than the cross-seed standard deviation (`±0.2%` to `±0.4%`), proving that both the Vision Transformer upgrade and the physical ABACAD interference filter contribute genuine, repeatable gains rather than lucky random-seed noise.

A Concise Compass for the Emerging Researcher

Never treat large models as magic: start by implementing small networks from scratch (such as Labs `02`, `06`, `22`, and `27` in `code.zip`), observe how weights and gradients evolve inside TensorBoard and H5Web, run disciplined ablations, and build your own research intuition step by step.

EPILOGUE

'The Empty Seat' & Humanity and Intelligence Toward the Cosmos

Beyond algorithms and computational power: human presence, empathy, and our shared responsibility for the future.

E.1 The Empty Seat: When Knowledge Meets Humanity

Throughout the journey of this seminar, we have explored synaptic weights, gradients, convolutions, self-attention mechanisms, GPU accelerators, and quantum gates. We have watched machines recognize images, translate languages, compose music, and even touch the symbolism of a poem. Yet at the end of every scientific voyage, it is worth standing in quiet reflection before a question that no algorithm can answer on our behalf.



Figure E.1 *'The Empty Seat' | What is it worth to know everything, if we do not learn to live together?*

> *"What is it worth to know everything, if we do not learn to live together?"*
>
> Technology multiplies knowledge, yet only human coexistence, empathy, and care give it meaning.

Algorithms can process trillions of parameters, yet they know nothing of the warmth of a human presence, the weight of a promise, or the joy of shared inquiry. Our duty, as an academic community, as engineers, and as citizens, is to ensure that Artificial Intelligence never empties the human seats around us, but serves human dignity, justice, and unity.

E.2 Humanity and Intelligence Reaching Across the Cosmos

When scientific curiosity walks hand in hand with ethical responsibility, Artificial Intelligence becomes a bridge across generations: an instrument that helps humanity heal disease, protect our fragile planet, and turn its gaze toward the stars with wonder and humility.

This seminar is not the end of a journey, but the starting point for your own questions, your own experiments, and new ideas. Study the algorithms with precision, run and modify the scripts in `code.zip`, challenge what you know creatively, and always keep humanity at the center of your thinking.



Figure E.2 *Humanity and Intelligence Reaching Across the Cosmos | Guided by knowledge, ethics, and the light of discovery.*

"Knowledge illuminates the path, yet humanity defines the destination."

Seminar on Artificial Intelligence, Machine Learning & Deep Learning • University of the Peloponnese

Electrical & Computer Engineering Department
UNIVERSITY OF PELOPONNESE

Acknowledgments & Monograph Colophon

Dedicated to the faculty, staff, and students of the University of the Peloponnese, and to everyone seeking the truth of our world through research, curiosity, and creative expression.

Evangelos V. Pefanis | Special Technical Laboratory Staff (E.T.E.P.), Dept. of Electrical & Computer Engineering, University of the Peloponnese
Companion Laboratory Codebase: `code.zip` (33 Python 3.12+ Labs)