# Tiny🔥Torch

Build Your Own Machine Learning Framework
From Tensors to Systems

Vijay Janapa Reddi
Harvard University
Cambridge, MA, USA

tinytorch.ai

## Abstract

Machine learning education typically teaches framework usage without exposing internals, leaving students unable to debug gradient flows, profile memory bottlenecks, or understand optimization tradeoffs. TinyTorch addresses this gap through a build-from-scratch curriculum where students implement PyTorch's core components—tensors, autograd, optimizers, and neural networks—to gain framework transparency.

We contribute three pedagogical patterns for teaching ML as systems engineering. **Progressive disclosure** activates dormant tensor features across modules through monkey-patching, modeling how frameworks evolve from separate abstractions to unified interfaces. **Systems-first curriculum** embeds memory profiling and complexity analysis from the start rather than treating them as advanced topics. **Historical milestone validation** recreates nearly 70 years of ML breakthroughs (1958 Perceptron through modern transformers) using exclusively student-implemented code to validate correctness.

The 20-module curriculum (60–80 hours) provides complete open-source implementation for institutional adoption or self-paced learning at `tinytorch.ai`.

## 1 Introduction

Machine learning deployment faces a critical workforce bottleneck: demand for ML systems engineers outstrips supply by over 3:1, with only 150,000 skilled practitioners worldwide serving an industry where AI job postings have grown 78% year-over-year (Robert Half, 2024; Keller Executive Search, 2025). Forty to fifty percent of executives cite this talent shortage as their primary barrier to AI adoption (Keller Executive Search, 2025).

Unlike algorithmic ML—where automated tools increasingly handle model architecture search and hyperparameter tuning—systems engineering remains bottlenecked by tacit knowledge that resists automation: understanding *why* Adam requires 2× optimizer state memory, *when* attention's $O(N^2)$ scaling becomes prohibitive, *how* to navigate accuracy-latency-memory tradeoffs in production systems. These engineering judgment calls depend on mental models of framework internals (Meadows, 2008), traditionally acquired through years of debugging PyTorch or TensorFlow rather than formal instruction.

Current ML education creates this gap by separating algorithms from systems. Students learn to implement gradient descent without measuring memory consumption, build attention mechanisms without profiling $O(N^2)$ costs, and train models without understanding optimizer state overhead. Introductory courses use high-level APIs (PyTorch, Keras) that abstract away implementation details, while advanced electives teach systems concepts (memory management, performance optimization) in isolation from ML frameworks. This pedagogical divide produces graduates who can *use* `loss.backward()` but cannot explain how computational graphs enable reverse-mode differentiation, or who understand transformers mathematically but miss that KV caching trades $O(N^2)$ memory for $O(N)$ recomputation.

We present TinyTorch, a 20-module curriculum where students build PyTorch's core components from scratch using only NumPy: tensors, automatic differentiation, optimizers, CNNs, transformers, and production optimization techniques. Students transition from framework *users* to framework *engineers* by implementing the

internals that high-level APIs deliberately hide. As a hands-on companion to the *Machine Learning Systems* textbook (Reddi), TinyTorch transforms tacit systems knowledge into explicit pedagogy—students don't just learn *that* Adam requires 4× training memory, they *implement* momentum and variance buffers and *measure* the footprint directly through profiling code they wrote. Figure 1 contrasts this bottom-up approach with traditional top-down API usage.

The curriculum addresses three fundamental questions. First, can students learn systems thinking *alongside* ML fundamentals rather than in separate electives? TinyTorch demonstrates that memory profiling, computational complexity analysis, and performance reasoning integrate naturally when students build components from scratch—Module 01 introduces tensor memory footprints before matrix operations, making systems awareness foundational rather than advanced. Second, how do we manage cognitive load when teaching both algorithms and implementation? Progressive disclosure (Section 4) solves this through runtime feature activation: gradient tracking exists but stays dormant in Modules 01-04, activating only when Module 05 introduces automatic differentiation. Third, can bottom-up implementation compete with top-down API usage for learning efficiency? Historical milestone validation provides evidence: students recreate 70 years of ML breakthroughs (1958 Perceptron → 2024 optimized transformers) using exclusively their own code, demonstrating that implementations work on real tasks.

The curriculum follows the compiler course model (Aho et al., 2006): students build a complete system module-by-module, experiencing how components integrate through direct implementation. Figure 2 illustrates the dependency structure—tensors (Module 01) enable activations (02) and layers (03), which feed into autograd (05), which powers optimizers (06) and training (07). This incremental construction mirrors how compiler courses connect lexical analysis to parsing to code generation, creating systems thinking through component integration. Each completed module becomes immediately usable: after Module 03, students can build neural networks; after Module 05, automatic differentiation enables training; after Module 13, transformers support language modeling.

TinyTorch serves students transitioning from framework *users* to framework *engineers*: those who have completed introductory ML courses (e.g., CS229, fast.ai) and want to understand PyTorch internals, those planning ML systems research or infrastructure careers, or practitioners debugging production deployment issues. The curriculum assumes NumPy proficiency and basic neural network familiarity but teaches framework architecture from first principles. Students needing immediate GPU/distributed training skills are better served by PyTorch tutorials; those preferring project-based application building will find high-level frameworks more appropriate. The 20-module structure supports flexible pacing: intensive bootcamp completion (2-3 weeks), semester integration (parallel with lectures), or self-paced professional development.

The curriculum introduces three pedagogical innovations. First, **progressive disclosure** manages cognitive load through runtime feature activation: `Tensor` gradient attributes exist from Module 01 but remain dormant until Module 05 activates automatic differentiation (Section 4). This monkey-patching technique maintains a unified mental model while revealing complexity gradually, teaching both current framework usage and historical evolution (PyTorch's Variable/Tensor merger). Second, **systems-first integration** embeds memory profiling, FLOPs analysis, and performance reasoning from Module 01 onwards rather than deferring to advanced electives (Section 5). Students measure what they build: Conv2d's 109× parameter efficiency over Dense layers, attention's $O(N^2)$ memory scaling, quantization's 4× compression. Third, **historical milestone validation** provides correctness proof through replication: students recreate nearly 70 years of ML breakthroughs (1958 Perceptron through 2024 Llama-style transformers) using exclusively their own implementations, demonstrating that their code works on real tasks.

This paper makes three primary contributions:

1. **Systems-First Curriculum Architecture**: A 20-module learning path integrating memory profiling, computational complexity, and performance analysis from Module 01 onwards, replacing traditional algorithm-systems separation. Students discover systems constraints through direct measurement (Adam's 2× optimizer state overhead, Conv2d's 109× parameter efficiency, KV caching's $O(n^2) \rightarrow O(n)$ transformation) rather than abstract instruction (Sections 3 and 5). This architecture directly addresses the workforce gap by making tacit systems knowledge explicit through hands-on implementation. Grounded in situated cognition (Lave and Wenger) and constructionism (Wooster and Papert), with systems thinking pedagogy informed by established frameworks (Meadows, 2008).

2. **Progressive Disclosure Pattern**: To make systems-first learning tractable, we introduce a pedagogical technique using monkey-patching (runtime method replacement) to reveal `Tensor` complexity gradually while maintaining a unified mental model. Dormant gradient features exist from Module 01 but

```python
import torch.nn as nn
import torch.optim as optim

model = nn.Linear(784, 10)
optimizer = optim.Adam(
    model.parameters(), lr=0.001)
loss_fn = nn.CrossEntropyLoss()

for epoch in range(10):
    for x, y in dataloader:
        pred = model(x)
        loss = loss_fn(pred, y)
        loss.backward()    # Magic?
        optimizer.step()   # How?
```

(a) PyTorch: Using frameworks as black boxes

```python
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(10,
        input_shape=(784,))
])
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy')

model.fit(dataloader, epochs=10)
# How does it work?
```

(b) TensorFlow: High-level abstractions

```python
class Linear:
    def __init__(self, in_features,
                 out):
        self.weight = Tensor.randn(
            out, in_features)
        self.bias = Tensor.zeros(out)

    def forward(self, x):
        return (x @ self.weight.T +
                self.bias)

class Adam:
    def __init__(self, params,
                 lr=0.001):
        self.params = params
        self.lr = lr
        # 2× optimizer state:
        # momentum + variance
        self.m = [Tensor.zeros_like(p)
                  for p in params]
        self.v = [Tensor.zeros_like(p)
                  for p in params]

    def step(self):
        for p, m, v in zip(self.params,
                           self.m,
                           self.v):
            m = 0.9*m + 0.1*p.grad
            v = 0.999*v + 0.001*p.grad**2
            p.data -= (self.lr * m /
                       (v.sqrt() + 1e-8))
```

(c) TinyTorch: Understanding internals

Figure 1: Learning progression from framework users to engineers. (a-b) PyTorch/TensorFlow: high-level API usage. (c) TinyTorch: building internals reveals optimizer memory costs, computational complexity, and systems constraints.

activate in Module 05, enabling forward-compatible code and teaching how frameworks like PyTorch evolved (Variable/Tensor merger) (Section 4). This pattern solves the cognitive load challenge inherent in teaching both algorithms and systems simultaneously. Grounded in cognitive load theory (Sweller) and cognitive apprenticeship (Collins et al.).

3. **Open Educational Infrastructure**: Both innovations are validated through a complete open-source curriculum with NBGrader assessment infrastructure (Jupyter et al.), three integration models (self-paced learning, institutional courses, team onboarding), historical milestone validation (1958 Perceptron through 2024 optimized transformers), and PyTorch-inspired package architecture. This infrastructure enables community adoption, curricular adaptation, and empirical research into ML systems pedagogy effectiveness (Sections 3, 6 and 7).

**Scope:** These contributions represent demonstrated design patterns and complete educational infrastructure grounded in established learning theory. The curriculum's technical correctness is validated through historical milestone recreation (students train CNNs achieving 75%+ CIFAR-10 accuracy using exclusively their own implementations). Learning outcome claims—that systems-first integration improves debugging skills, that progressive disclosure reduces cognitive load, that graduates achieve production readiness faster—remain testable hypotheses requiring empirical validation through controlled classroom studies. We detail specific research questions and measurement methodologies in Section 7.

**Paper Organization** Before presenting TinyTorch's design, we position our contributions relative to existing educational frameworks and grounding learning theories (Section 2). We then present the systems-first curriculum architecture (Section 3), its integration throughout modules (Section 5), and the progressive disclosure pattern enabling cognitive load management (Section 4). Finally, we discuss limitations, empirical validation plans, and implications for ML education (Sections 7 and 9).

## 2 Related Work

TinyTorch builds upon decades of work in CS education research and recent innovations in ML framework pedagogy. We position our contributions relative to ex-

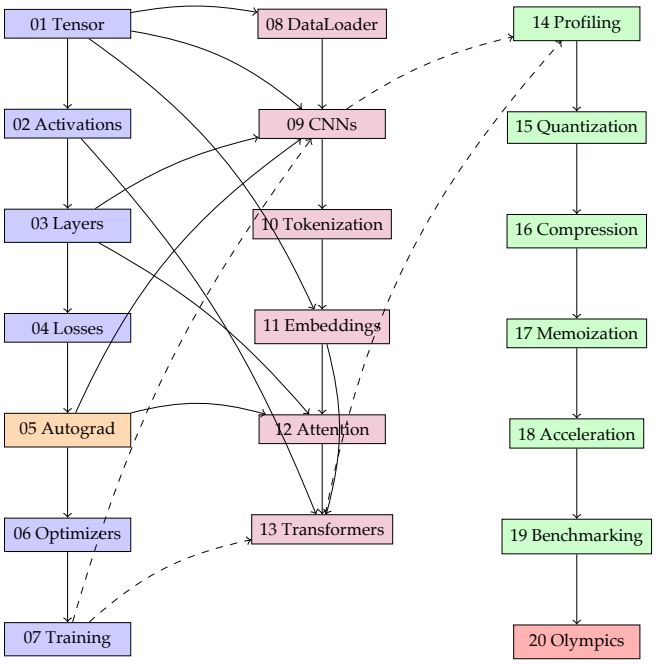


Figure 2: Module dependency flow: How components connect across tiers. Foundation modules (blue) enable architectures (purple), which are optimized (green), culminating in the capstone competition (red). Dotted lines show cross-tier integration.

isting educational frameworks and grounding learning theories.

## 2.1 Educational ML Frameworks

Educational frameworks teaching ML internals occupy different points in the scope-simplicity tradeoff space. **micrograd** (Karpathy, 2022) demonstrates autograd mechanics elegantly in approximately 200 lines of scalar-valued Python, making backpropagation transparent through decomposition into elementary operations. Its pedagogical clarity comes from intentional minimalism: scalar operations only, no tensor abstraction, focused solely on automatic differentiation fundamentals. This design illuminates gradient mechanics but necessarily omits systems concerns (memory profiling, computational complexity, production patterns) and modern architectures.

**MiniTorch** (Rush, 2020) extends beyond autograd to tensor operations, neural network modules, and optional GPU programming, originating from Cornell Tech's Machine Learning Engineering course. The curriculum progresses from foundational autodifferentiation through deep learning with assessment infrastructure (unit tests, visualization tools). While MiniTorch includes an optional GPU module exploring parallel programming concepts and covers efficiency considerations throughout, the core curriculum emphasizes mathematical rigor: students work through detailed exercises building tensor abstractions from first principles. TinyTorch differs through systems-first emphasis (memory profiling and complexity analysis embedded from Module 01), production-inspired package organization, and three integration models supporting diverse deployment contexts.

**tinygrad** (Hotz and contributors, 2023) positions itself between micrograd's simplicity and PyTorch's production capabilities, providing a complete framework (tensor library, IR, compiler, JIT) that emphasizes hackability and transparency. Unlike opaque production frameworks, tinygrad makes "the entire compiler and IR visible," enabling students to understand deep learning compilation internals. While pedagogically valuable through its inspectable design, tinygrad assumes significant background: students must navigate compiler concepts, multiple hardware backends, and production-level architecture without scaffolded progression or automated assessment infrastructure.

**Stanford CS231n** (Johnson et al., 2016), **CMU Deep Learning Systems (CS 10-414)** (Chen and Zheng, 2022), and **Harvard TinyML** (Reddi et al., b) represent university courses that include implementation components with different systems emphases. CS231n's assignments involve NumPy implementations of CNNs, backpropagation, and optimization algorithms, providing hands-on experience with neural network internals. However, assignments are isolated exercises rather than cumulative framework construction, and systems concerns (memory profiling, complexity analysis) are not embedded from the start. CMU's DL Systems course explicitly targets ML systems engineering, covering automatic differentiation, GPU programming, distributed training, and deployment—representing the production systems knowledge TinyTorch provides conceptual foundations for. Harvard's TinyML Professional Certificate focuses on deploying ML to resource-constrained embedded devices (microcontrollers with KB-scale memory), teaching TensorFlow Lite for Microcontrollers through Arduino-based projects. While TinyML emphasizes hardware constraints and embedded deployment (achieving systems thinking through resource limitations), TinyTorch focuses on framework internals and algorithmic understanding (achieving systems thinking through implementation transparency). TinyML students learn *how to optimize for* hardware constraints; TinyTorch students learn *why frameworks work* internally. These approaches complement rather than compete: TinyML prepares students for edge deployment, TinyTorch for framework engineering and infrastructure development.

**Dive into Deep Learning (d2l.ai)** (Zhang et al., 2021)

and **fast.ai** (Howard and Gugger) represent comprehensive ML education but with different pedagogical emphases than framework construction. d2l.ai provides interactive implementations across multiple frameworks (PyTorch, JAX, TensorFlow, MXNet/NumPy) through executable notebooks, teaching algorithmic foundations alongside practical coding. The NumPy implementation track includes from-scratch implementations of key algorithms, though these are presented as educational demonstrations rather than components of a cumulative framework students build. With widespread adoption across hundreds of universities globally, it excels at algorithmic understanding through framework usage. fast.ai's distinctive top-down pedagogy starts with practical applications before foundations, using layered APIs that provide high-level abstractions while enabling deeper exploration through PyTorch. Both resources assume cloud computing access (AWS, Google Colab, SageMaker) for GPU-based training, though provide various deployment options.

## 2.2 Learning Theory Foundations

TinyTorch's pedagogical design draws on established learning theories validated across CS education.

**Constructionism** (Wooster and Papert) argues learning occurs most effectively when students construct artifacts others can examine. TinyTorch realizes this through framework building—students create tangible ML systems enabling peer code review, portfolio demonstration, and conceptual debugging through concrete implementation.

**Cognitive Apprenticeship** (Collins et al.) emphasizes making expert thinking visible through modeling, coaching, and scaffolding. Progressive disclosure (Section 4) models expert framework evolution: students experience how PyTorch's Tensor class grew capabilities (matching PyTorch 0.4's Variable-Tensor merger (Team, 2018)). Module structure provides coaching through connection maps (showing prerequisites and unlocked capabilities) and scaffolding through integration tests validating cross-module composition.

**Productive Failure** (Kapur) demonstrates that struggling with problems before instruction deepens understanding compared to direct teaching. TinyTorch applies this through minimal upfront explanation: students implement components first, encounter integration failures ("My Conv2d passes unit tests but crashes during backpropagation"), then discover why interface design matters. Historical milestones validate eventual success, providing delayed gratification after productive struggle.

**Threshold Concepts** (Meyer and Land, 2003) identify transformative ideas (computational graphs, gradient flow, memory-compute tradeoffs) whose mastery fundamentally changes student thinking. Unlike traditional ML courses presenting these abstractly, TinyTorch makes thresholds concrete through implementation: students cannot complete Module 05 without understanding computational graphs because they must implement the graph data structure. Systems-first integration (Section 5) addresses the "memory reasoning" threshold: calculating VRAM requirements before operations becomes reflexive through repeated practice starting Module 01.

## 2.3 Positioning and Unique Contributions

TinyTorch occupies a distinct pedagogical niche through its **bottom-up, systems-first approach**. Unlike top-down pedagogies (fast.ai: start with applications, descend to details) or algorithm-focused curricula (d2l.ai: master theory through framework usage), TinyTorch employs bottom-up framework construction: students build core abstractions first (tensors, autograd, layers), then compose them into architectures (CNNs, transformers), finally optimizing for production constraints (quantization, compression). This grounds systems thinking in direct implementation rather than abstract instruction. The curriculum serves students post-introductory-ML (ready to transition from framework users to engineers), pre-systems-research (providing foundations before production ML courses like CMU DL Systems), and complementary to algorithm courses (adding systems awareness to mathematical foundations).

Table 1 positions TinyTorch relative to both educational frameworks (micrograd, MiniTorch, tinygrad) and production frameworks (PyTorch, TensorFlow), clarifying that TinyTorch serves as pedagogical bridge between understanding frameworks and using them professionally.

TinyTorch differs from educational frameworks through systems-first integration and from production frameworks through pedagogical transparency:

- **vs. micrograd**: Complete framework scope beyond autograd (tensors, architectures, optimizers, transformers), systems integration (memory/performance from Module 01), automated assessment infrastructure (NBGrader)
- **vs. MiniTorch**: TinyTorch inverts pedagogical priorities—where MiniTorch prioritizes mathematical rigor (students work through tensor abstractions and broadcasting semantics before encountering systems concerns, with GPU optimization as optional advanced module), TinyTorch embeds systems thinking from Module 01 (students calculate memory footprints before matrix multiplication, profile FLOPs during convolution). Progressive disclosure maintains unified API across modules (Ten-

Table 1: Framework comparison: Educational vs. Production

| Framework | Primary Purpose | Scope | Systems Focus | Target Outcome |
|---|---|---|---|---|
| **Educational Frameworks** | | | | |
| micrograd | Teach autograd | Autograd only (scalar) | Minimal | Understand backprop |
| MiniTorch | Teach ML math | Tensors + autograd + optional GPU | Math foundations | Build from first principles |
| tinygrad | Inspectable production | Complete (compiler, IR, JIT) | Advanced (compiler) | Understand compilation |
| **TinyTorch** | **Teach systems** | **Complete (tensors → transformers → optimization)** | **Embedded from Module 01** | **Framework engineers** |
| **Production Frameworks** | | | | |
| PyTorch | Production ML | Complete (GPU, distributed, deployment) | Advanced (implicit) | Train models efficiently |
| TensorFlow | Production ML | Complete (GPU, distributed, deployment, mobile) | Advanced (implicit) | Deploy at scale |

sor class evolves via monkey-patching) while MiniTorch introduces separate abstractions (ScalarTensor □ Tensor □ CudaTensor), modeling production framework evolution versus pedagogical scaffolding

- **vs. tinygrad**: Scaffolded pedagogical progression with assessment versus inspectable production system, accessibility (CPU-only, no compiler background required) versus hackability (multiple backends, IR exploration)

- **vs. d2l.ai**: Framework construction (build internals) versus algorithmic mastery (apply frameworks), systems-first integration versus algorithm-focused curriculum

- **vs. fast.ai**: Bottom-up framework building versus top-down application focus, constructionist artifacts (students create importable framework) versus practitioner training (students use layered APIs)

Empirical validation of learning outcomes remains future work (Section 7), but design grounding in established theory (constructionism, cognitive apprenticeship, productive failure, threshold concepts) provides theoretical justification for pedagogical choices.

# 3 Curriculum Architecture

This section presents the 20-module curriculum structure, organized into four tiers that progressively build a complete ML framework.

Traditional ML education presents algorithms sequentially without revealing how components integrate into working systems. TinyTorch addresses this through a 4-phase curriculum architecture where students build a complete ML framework progressively, with each module enforcing prerequisite mastery.

## 3.1 Prerequisites and Target Audience

TinyTorch targets students ready to transition from framework users to framework engineers. The curriculum assumes intermediate Python proficiency—comfort with classes, functions, and NumPy array operations—alongside mathematical foundations in linear algebra (matrix multiplication, vectors) and basic calculus (derivatives, chain rule). Students should understand complexity analysis (Big-O notation) and basic algorithms. While prior ML coursework (traditional machine learning or deep learning courses) and data structures courses are helpful, they are not strictly required; motivated students can acquire these foundations concurrently.

The primary audience consists of junior and senior computer science undergraduates who have completed

```python
class Tensor:
    def __init__(self, data):
        self.data = np.array(data, dtype=np.
    float32)
        self.shape = self.data.shape

    def memory_footprint(self):
        """Calculate exact memory in bytes"""
        return self.data.nbytes

    def __matmul__(self, other):
        if self.shape[-1] != other.shape[0]:
            raise ValueError(
                f"Shape mismatch: {self.shape} @
     {other.shape}"
            )
        return Tensor(self.data @ other.data)
```

Listing 1: Tensor with memory profiling from Module 01.

introductory ML courses and seek deeper systems understanding. Graduate students transitioning to ML systems research form a secondary audience, while self-learners with strong programming backgrounds represent a tertiary group. The curriculum's flexible pacing accommodates diverse contexts: intensive completion over weeks, semester integration within existing courses, or self-paced professional development.

### 3.2 The 3-Tier Learning Journey + Olympics

TinyTorch organizes modules into three progressive tiers plus a capstone competition (Table 2). Students cannot skip tiers: architectures require foundation mastery, optimization demands training system understanding. The tiers mirror ML systems engineering practice: foundation (core ML mechanics), architectures (domain-specific models), optimization (production deployment), culminating in the AI Olympics (competitive systems engineering).

**Tier 1: Foundation (Modules 01–07).** Students build the mathematical core enabling neural networks to learn. Systems thinking begins immediately—Module 01 introduces `memory_footprint()` before matrix multiplication (Listing 1), making memory a first-class concept. The tier progresses through tensors, activations, layers, and losses to automatic differentiation (Module 05)—where dormant gradient features activate through progressive disclosure (Section 4). Students implement optimizers (Module 06), discovering Adam's memory trade-offs through direct measurement (Section 5). The training loop (Module 07) integrates all components. By tier completion, students recreate three historical milestones: (1958)'s Perceptron, Minsky and Papert's XOR solution, and Rumelhart et al.'s backpropagation achieving 95%+ on MNIST. Students calculate memory before operations: "Matrix multiplication A @ B where both are (1000, 1000) FP32 requires 12MB peak memory: 4MB for A, 4MB for B, and 4MB for the output." This reasoning becomes automatic.

**Tier 2: Architectures (Modules 08–13).** Students apply foundation knowledge to modern architectures for vision and language. Module 08 introduces the Dataset abstraction pattern (implementing `__len__` and `__getitem__` protocols) and DataLoader with batch collation, teaching how PyTorch's data pipeline transforms individual samples into batched tensors through the iterator protocol. TinyTorch ships with two custom educational datasets that install with the repository: **TinyDigits** (5,000 grayscale handwritten digits, curated from public digit datasets) and **TinyTalks** (3,000 synthetically-generated conversational Q&A pairs). These datasets are deliberately small and offline-first: they require no network connectivity during training, consume minimal storage (<50MB combined), and train in minutes on CPU-only hardware. This design ensures accessibility for students in regions with limited internet infrastructure, institutional computer labs with restricted network access, and developing countries where cloud-based datasets create barriers to ML education.

The tier then branches into two paths. **Vision** implements Conv2d with seven explicit nested loops making $O(C_{out} \times H \times W \times C_{in} \times K^2)$ complexity visible before optimization. Students discover weight sharing's dramatic efficiency through direct comparison: Conv2d(3→32, kernel=3) requires 896 parameters while an equivalent dense layer needs 98,336 parameters (3072 input features $\times$ 32 outputs + 32 bias terms)—a 109$\times$ reduction demonstrating how inductive biases enable CNNs to learn spatial patterns without brute-force parameterization. This enables Milestone 4 (1998 CNN Revolution) with 75%+ CIFAR-10 accuracy (Krizhevsky and Hinton, 2009; Lecun et al.).

**Language** progresses through tokenization (character-level and BPE), embeddings (both learned and sinusoidal positional encodings), attention ($O(N^2)$ memory), and complete transformers (Vaswani et al.). Module 10 (Tokenization) teaches a fundamental NLP systems trade-off: vocabulary size controls model parameters (embedding matrix rows $\times$ dimensions), while sequence length determines transformer computation ($O(n^2)$ attention complexity). Students discover why later GPT models increased vocabulary from 50K tokens (GPT-2/GPT-3) to 100K tokens (GPT-3.5/GPT-4)—not for better language understanding, but to reduce sequence lengths for long documents, trading parameter memory for computational efficiency. Students experience quadratic scaling through direct measurement. Milestone 5 (2017 Transformer Era) validates through text generation on

Table 2: Module-by-module ML and Systems concepts (embedded from the start)

| Mod | Tier | Module Name | ML Concept | Systems Concept |
|---|---|---|---|---|
| **Foundation Tier (01–07)** | | | | |
| 01 | Fnd | Tensor | Multidimensional arrays, broadcasting | Memory footprint (nbytes), FP32 storage |
| 02 | Fnd | Activations | ReLU, Sigmoid, Softmax | Numerical stability (exp overflow), vectorization |
| 03 | Fnd | Layers | Linear, parameter initialization | Parameter memory vs activation memory |
| 04 | Fnd | Losses | Cross-entropy, MSE | Stability (log(0) handling), gradient flow |
| 05 | Fnd | Autograd | Computational graphs, backprop | Gradient memory, optimizer state (2× for Adam) |
| 06 | Fnd | Optimizers | SGD, Momentum, Adam | Memory-speed tradeoffs, update rules |
| 07 | Fnd | Training Loop | Epoch/batch iteration | Forward/backward memory lifecycle |
| **Architecture Tier (08–13)** | | | | |
| 08 | Arch | DataLoader | Batching, shuffling, Dataset abstraction | Iterator protocol, batch collation, memory layout |
| 09 | Arch | Spatial (CNNs) | Conv2d, kernels, strides, pooling | $O(B \times C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}} \times C_{\text{in}} \times K_h \times K_w)$ complexity |
| 10 | Arch | Tokenization | BPE (Byte Pair Encoding), vocabulary, encoding | Vocabulary management, OOV handling |
| 11 | Arch | Embeddings | Token/position embeddings | Lookup tables, gradient through indices |
| 12 | Arch | Attention | Scaled dot-product attention | $O(N^2)$ memory scaling, sequence length impact |
| 13 | Arch | Transformers | Multi-head, encoder/decoder | Quadratic memory, KV caching strategies |
| **Optimization Tier (14–19)** | | | | |
| 14 | Opt | Profiling | Time, memory, FLOPs analysis | Bottleneck identification, measurement overhead |
| 15 | Opt | Quantization | INT8, dynamic/static quant | 4× model size reduction, accuracy-speed tradeoff |
| 16 | Opt | Compression | Pruning, distillation | 10× model shrinkage, minimal accuracy loss |
| 17 | Opt | Memoization | KV-cache for transformers | 10–100× inference speedup via caching |
| 18 | Opt | Acceleration | Vectorization, parallelization | 10–100× speedup via NumPy optimization |
| 19 | Opt | Benchmarking | Statistical testing, comparisons | Rigorous performance measurement |
| **AI Olympics (20)** | | | | |
| 20 | Capstone | AI Olympics | Complete production system | MLPerf-style competition, leaderboard |

TinyTalks.

**Tier 3: Optimization (Modules 14–19).** Students transition from "models that train" to "systems that deploy." Profiling (14) teaches measuring time, memory, and FLOPs (floating-point operations), introducing Amdahl's Law: optimizing 70% of runtime by 2× yields only 1.53× overall speedup because the remaining 30% becomes the new bottleneck—teaching that optimization is iterative and measurement-driven. Quantization (15) achieves 4× compression (FP32→INT8) with 1–2% accuracy cost. Compression (16) applies pruning and distillation for 10× shrinkage. Memoization (17) implements KV caching (storing attention keys and values to avoid recomputation), a technique used in production LLM serving: students discover that naive autoregressive generation recomputes attention keys and values at every step—generating 100 tokens requires 5,050 redundant computations (1+2+...+100). By caching these values and reusing them, students transform $O(n^2)$ generation into $O(n)$, achieving 10–100× speedup and understanding why this optimization is essential in systems like ChatGPT and Claude for economically viable inference. Acceleration (18) vectorizes convolution for 10–100× gains. Benchmarking (19) teaches rigorous performance measurement.

**AI Olympics (Module 20).** The capstone integrates all 19 modules into production-optimized systems. Inspired by MLPerf (Reddi et al., a), students optimize prior milestones (CIFAR-10 CNN, transformer generation, or custom architecture) for 10× faster inference, 4× smaller size, and sub-100ms latency while maintaining accuracy. Students compete on the TinyTorch Leaderboard across four tracks: Vision Excellence, Language Quality, Speed, and Compression. This teaches data-driven optimization mirroring real ML systems engineering.

## 3.3 Module Structure

Each module follows a consistent **Build** → **Use** → **Reflect** pedagogical cycle that integrates implementation, application, and systems reasoning. This structure addresses multiple learning objectives: students construct working components (Build), validate integration with prior modules (Use), and develop systems thinking through analysis (Reflect).

**Build: Implementation with Explicit Dependencies** Students implement components in Jupyter notebooks (`*_dev.py`) with scaffolded guidance. Each module begins with *connection maps* visualizing prerequisites, current focus, and unlocked capabilities. For example, Module 05 (Autograd) shows prerequisites (Modules 01–04: Tensor, Activations, Layers, Losses), current implementation goal (computational graph + backward pass), and unlocked future modules (Modules 06–07: Optimizers, Training). These visual dependency chains address cognitive apprenticeship (Collins et al.) by making expert knowledge structures explicit. Students see "why this module matters" before implementation begins, reducing disengagement from seemingly isolated exercises.

**Use: Integration Testing Beyond Unit Tests** Assessment validates both isolated correctness and cross-module integration. Unit tests verify individual component behavior ("Does `Tensor.reshape()` produce correct output?"), while integration tests validate that components compose into working systems ("Can Module 05 Autograd compute gradients through Module 03 Linear layers?"). Integration tests are critical for TinyTorch's pedagogical model because students may pass Module 03 unit tests but fail when autograd activates in Module 05—their layer implementation doesn't properly propagate `requires_grad` through operations or construct computational graphs correctly.

A common failure pattern illustrates this: students implement `Linear.forward()` that passes unit tests (correct output values), but gradients don't flow during backpropagation because they used NumPy operations directly instead of `Tensor` operations. When `x.requires_grad=True` flows into their layer, the computational graph breaks. Students encounter errors like `"AttributeError: 'numpy.ndarray' object has no attribute 'backward'"` and must debug interface contracts: operations must preserve `Tensor` types to maintain gradient connectivity. This teaches *interface design*—components must satisfy contracts enabling composition, not just produce correct outputs in isolation.

Module 09 (Convolutions) integration exemplifies this: convolution must work with Module 05's autograd (gradient flow through kernels), Module 06's optimizers (parameter updates), and Module 07's training loop (forward-backward cycles) simultaneously. Students discover that "passing unit tests" $\neq$ "works in the system" when their Conv2d produces correct outputs but crashes during `loss.backward()` because they forgot to track intermediate activations for gradient computation. This debugging mirrors professional ML engineering: isolated correctness is insufficient; system integration reveals interface failures.

**Reflect: Systems Analysis Questions** Each module concludes with systems reasoning prompts measuring conceptual understanding beyond syntactic correctness.

Memory analysis questions ask students to calculate footprints ("A (256, 256) Conv2d layer with 64 input and 128 output channels requires how much memory?"). Complexity analysis prompts probe asymptotic understanding ("Why is attention $O(N^2)$? Demonstrate by doubling sequence length and measuring memory growth."). Design trade-off questions assess engineering judgment ("Adam requires 2× optimizer state memory (momentum and variance) but converges faster than SGD. When is the 4× total training memory trade-off worth it?"). These open-ended questions assess transfer (Perkins and Salomon, 1992)—can students apply learned concepts to novel scenarios not seen in exercises?

### 3.4 Milestone Arcs

**Why Milestones Matter** Milestones serve dual pedagogical and validation purposes that differentiate TinyTorch from traditional programming assignments. First, **pedagogical motivation through historical framing**: Rather than "implement this function," students "recreate the breakthrough that proved Minsky wrong about neural networks," connecting implementation work to historically significant results. This instantiates Bruner's spiral curriculum (Bruner, 1960)—students train neural networks 6 times with increasing sophistication, each iteration deepening understanding through historical progression from 1958 (Perceptron) through 2024 (production-optimized systems).

Second, **implementation validation beyond unit tests**: Milestones differ from modules pedagogically—modules teach components, milestones validate that components *compose* into functional systems. Students who pass all Module 01–07 unit tests might still fail Milestone 3 (MLP Revival) if their training loop doesn't properly orchestrate forward passes, loss computation, and backpropagation. This mirrors professional ML engineering: individual functions may work, but the system fails due to integration bugs. If student-implemented CNNs successfully classify natural images, convolution, pooling, and backpropagation all work correctly together; if transformers generate coherent text, attention mechanisms integrate properly. Milestone success is measured by achieving performance in the ballpark of historical benchmarks (CNNs with reasonable CIFAR-10 accuracy, transformers generating coherent text), not matching exact published accuracies—the goal is demonstrating implementations work correctly on real tasks, validating framework correctness.

**The Six Historical Milestones** The curriculum includes six milestones spanning 1958–2024, each requiring progressively more components from the growing framework:

1. **1958 Perceptron** (after Module 04): Train Rosenblatt's original single-layer perceptron on linearly separable classification. Students import `from tinytorch.core import Tensor; from tinytorch.nn import Linear, Sigmoid`—their framework now supports single-layer networks.
2. **1969 XOR Solution** (after Module 07): Solve Minsky's "impossible" XOR problem with multi-layer perceptrons, proving critics wrong. Validates that autograd enables non-linear learning.
3. **1986 MLP Revival** (after Module 07): Handwritten digit recognition demonstrating backpropagation's power. Requires Modules 01–07 working together (tensor operations, activations, layers, losses, autograd, optimizers, training). Students import `from tinytorch.optim import SGD; from tinytorch.nn import CrossEntropyLoss`—their framework trains multi-layer networks end-to-end with 95%+ MNIST accuracy.
4. **1998 CNN Revolution** (after Module 09): Image classification demonstrating convolutional architectures' advantage through 75%+ CIFAR-10 accuracy (Krizhevsky and Hinton, 2009; Lecun et al.)—the "north star" achievement validating framework correctness. Students import `from tinytorch.nn import Conv2d, MaxPool2d`, training both MLP and CNN on identical data to measure architectural improvements themselves.
5. **2017 Transformer Era** (after Module 13): Language generation with attention-based architecture. Validates that attention mechanisms, positional embeddings, and autoregressive sampling function correctly through coherent text generation.
6. **2024 MLPerf Benchmark Era** (after Module 20): Production-optimized system integrating all 20 modules, inspired by MLPerf (Reddi et al., a). Students import from every module: `from tinytorch.nn import Transformer; from tinytorch.optim import Adam; from tinytorch.profiling import profile_memory`—demonstrating quantization, compression, and acceleration for 10× faster inference and 4× smaller models.

Each milestone: (1) recreates actual breakthroughs using exclusively student code, (2) uses *only* TinyTorch implementations (no PyTorch/TensorFlow), (3) validates

```python
# Module 01: Foundation Tensor
class Tensor:
    def __init__(self, data, requires_grad=False
    ):
        self.data = np.array(data, dtype=np.
    float32)
        self.shape = self.data.shape
        # Gradient features - dormant
        self.requires_grad = requires_grad
        self.grad = None
        self._backward = None

    def backward(self, gradient=None):
        """No-op until Module 05"""
        pass

    def __mul__(self, other):
        return Tensor(self.data * other.data)
```

Listing 2: Module 01: Dormant gradient features.

```python
def enable_autograd():
    """Monkey-patch Tensor with gradients"""
    def backward(self, gradient=None):
        if gradient is None:
            gradient = np.ones_like(self.data)
        if self.grad is None:
            self.grad = gradient
        else:
            self.grad += gradient
        if self._backward is not None:
            self._backward(gradient)

    # Monkey-patch: replace methods
    Tensor.backward = backward
    print("Autograd activated!")

# Module 05 usage
enable_autograd()
x = Tensor([3.0], requires_grad=True)
y = x * x  # y = 9.0
y.backward()
print(x.grad)  # [6.0] - dy/dx = 2x
```

Listing 3: Module 05: Autograd activation.

success through task-appropriate performance, and (4) demonstrates architectural comparisons showing why new approaches improved over predecessors.

# 4 Progressive Disclosure

The curriculum architecture described above raises a pedagogical challenge: how do students learn complex framework features without cognitive overload? This section presents progressive disclosure, a pattern that manages complexity by revealing `Tensor` capabilities gradually while maintaining a unified mental model.

Traditional ML education faces a pedagogical dilemma: students need to understand complete systems, but introducing all concepts simultaneously overwhelms cognitive capacity. Educational frameworks employ various strategies: some introduce separate classes (fragmenting the conceptual model), others defer advanced features until later courses (leaving gaps). TinyTorch introduces a third approach: **progressive disclosure via monkey-patching**, where a single `Tensor` class reveals capabilities gradually while maintaining conceptual unity.

## 4.1 Pattern Implementation

TinyTorch's `Tensor` class includes gradient-related attributes from Module 01, but they remain dormant until Module 05 activates them through monkey-patching (Listings 2 and 3).

This design serves three pedagogical purposes: (1) **Early interface familiarity**—students learn complete `Tensor` API from start; (2) **Forward compatibility**—Module 01 code doesn't break when autograd activates; (3) **Curiosity-driven learning**—dormant features create questions motivating curriculum progression.

## 4.2 Pedagogical Justification

Progressive disclosure is hypothesized to manage cognitive load through two competing mechanisms. First, early API familiarity may reduce future cognitive load when features activate (students already know the interface). Second, dormant features visible from Module 01 may create split-attention effects requiring students to track "not yet functional" components (Sweller). Whether the net effect reduces or increases cognitive load requires empirical measurement through dual-task methodology or cognitive load self-report scales. The pattern's primary validated benefit is forward compatibility—Module 01 code continues working when autograd activates—rather than proven cognitive load reduction. Empirical testing planned for Fall 2025 will measure actual cognitive load impact across both mechanisms.

The pattern also instantiates threshold concept pedagogy (Meyer and Land, 2003): autograd is transformative and troublesome. By making it visible early (dormant) but activatable later, students may cross this threshold when cognitively ready, though empirical evidence of this progression is needed.

## 4.3 Production Framework Alignment

Progressive disclosure demonstrates how real ML frameworks evolve. Early PyTorch (pre-0.4) separated data (`torch.Tensor`) from gradients (`torch.autograd.Variable`). PyTorch 0.4 (April 2018) (Team, 2018) consolidated functionality into `Tensor`, matching TinyTorch's pattern. Students are exposed to the modern unified interface from Module 01, positioned to understand why PyTorch made this

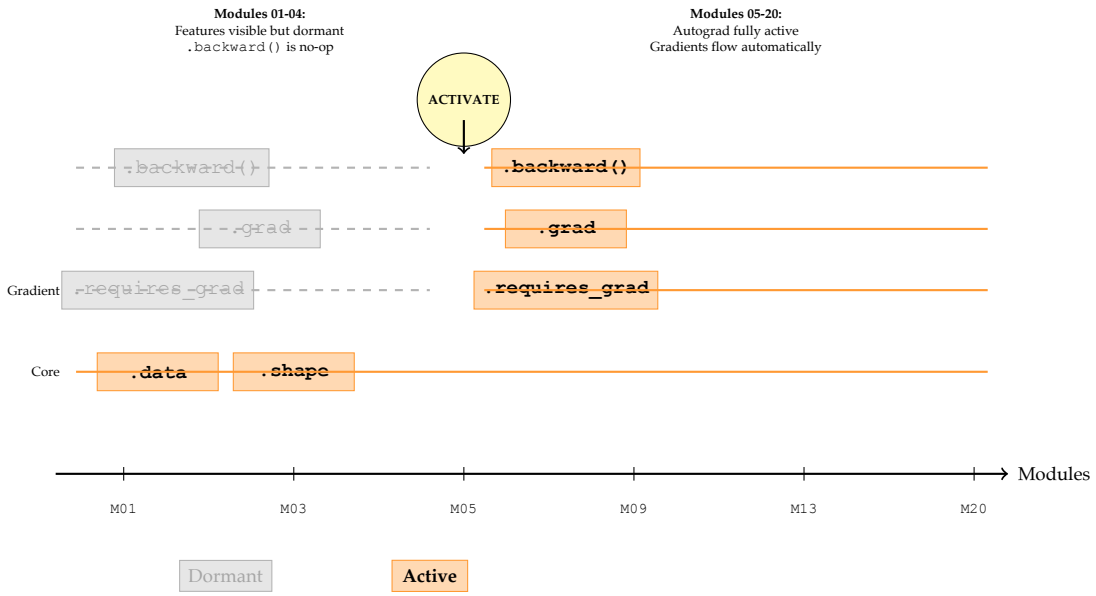


Figure 3: Progressive activation of Tensor gradient features. Dormant placeholders (gray, dashed) exist from Module 01. Module 05 activates full autograd functionality (orange, solid) via runtime method enhancement.

design evolution.

Similarly, TensorFlow 2.0 integrated eager execution by default (Team, 2019), making gradients work immediately—similar to TinyTorch's activation pattern. Students who understand progressive disclosure can grasp why TensorFlow eliminated `tf.Session()`: immediate execution with automatic graph construction aligns with unified API design principles.

# 5 Systems-First Integration

Industry surveys show ML engineers spending more time on memory optimization and debugging than hyperparameter tuning, yet most curricula defer systems thinking to senior electives. TinyTorch applies situated cognition (Lave and Wenger) by integrating systems awareness from Module 01 through a three-phase progression: (1) **understanding memory** through explicit profiling, (2) **analyzing complexity** through transparent implementations, and (3) **optimizing systems** through measurement-driven iteration. This mirrors professional ML engineering workflow: measure resource requirements, understand computational costs, then optimize bottlenecks.

## 5.1 Phase 1: Understanding Memory Through Profiling

Where traditional frameworks abstract away memory concerns, TinyTorch makes memory footprint calculation explicit (Listing 1). Students' first assignment calculates memory for MNIST ($60{,}000 \times 784 \times 4$ bytes $\approx 180$ MB) and ImageNet ($1.2\text{M} \times 224 \times 224 \times 3 \times 4$ bytes $\approx 670$ GB).

This memory-first pedagogy transforms student questions:

- Module 01: "Why does batch size affect memory?" (activations scale with batch size)
- Module 06: "Why does Adam use $2\times$ optimizer state memory?" (momentum and variance buffers)
- Module 13: "How much VRAM for GPT-3 training?" (175B parameters $\times$ 4 bytes $\times$ 4 $\approx$ 2.6 TB: weights + gradients + momentum + variance)

Students learn to distinguish parameter memory (model weights) from optimizer state memory (Adam's $2\times$ for momentum and variance) from activation memory (often 10–100$\times$ larger than parameters). This decomposition enables accurate capacity planning for training runs.

## 5.2 Phase 2: Analyzing Complexity Through Transparent Implementations

Module 09 introduces convolution with seven explicit nested loops (Listing 4), making $O(B \times C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}} \times C_{\text{in}} \times K_h \times K_w)$ complexity visible and countable.

This explicit implementation illustrates TinyTorch's pedagogical philosophy: **minimal NumPy reliance until concepts are established**. While the curriculum builds on NumPy as foundational infrastructure (array

```python
def conv2d_explicit(input, weight):
    """7 nested loops - see the complexity!
    input: (B, C_in, H, W)
    weight: (C_out, C_in, K_h, K_w)"""
    B, C_in, H, W = input.shape
    C_out, _, K_h, K_w = weight.shape
    H_out, W_out = H - K_h + 1, W - K_w + 1
    output = np.zeros((B, C_out, H_out, W_out))

    # Count: 1,2,3,4,5,6,7 loops
    for b in range(B):
        for c_out in range(C_out):
            for h in range(H_out):
                for w in range(W_out):
                    for c_in in range(C_in):
                        for kh in range(K_h):
                            for kw in range(K_w)
    :
                                output[b,c_out,h
    ,w] += \
                                    input[b,c_in
    ,h+kh,w+kw] * \
                                    weight[c_out
    ,c_in,kh,kw]
    return output
```

Listing 4: Explicit convolution showing 7-nested complexity.

storage, broadcasting, element-wise operations), optimized operations like matrix multiplication appear only after students understand computational complexity through explicit loops. Module 03 introduces linear layers with manual weight-input multiplication loops before Module 08 introduces NumPy's `@` operator; Module 09 teaches convolution through seven nested loops before Module 18 vectorizes with NumPy operations. This progression ensures students understand *what* operations do (and their complexity) before learning *how* to optimize them. Pure Python transparency enables this pedagogical sequencing—students can inspect every operation without navigating compiled C extensions or CUDA kernels.

Students calculate: CIFAR-10 batch (128, 3, 32, 32) through 32-filter 5×5 convolution: $128 \times 32 \times 28 \times 28 \times 3 \times 5 \times 5 = 241$M multiply-accumulate operations. This concrete measurement motivates Module 18's vectorization (10–100× speedup) and explains why CNNs require hardware acceleration.

**Experiencing Performance Reality** Table 3 shows TinyTorch's deliberate slowness compared to PyTorch—100–1000× slower through pure Python implementations. This slowness is pedagogically valuable (productive failure (Kapur)): students experience performance problems before learning optimizations, making vectorization meaningful rather than abstract. When students measure their Conv2d taking 8.4 seconds per CIFAR batch versus PyTorch's 0.09 seconds, they understand

Table 3: Runtime comparison: TinyTorch vs PyTorch (CPU).

| Operation | TinyTorch | PyTorch | Ratio |
|---|---|---|---|
| `matmul` (1K×1K) | 890 ms | 2.1 ms | 424× |
| `conv2d` (CIFAR batch) | 8.4 s | 0.09 s | 93× |
| `softmax` (10K elem) | 45 ms | 0.12 ms | 375× |
| CIFAR-10 epoch (LeNet) | 12 min | 8 sec | 90× |

*why* production frameworks obsess over implementation details.

### 5.3 Phase 3: Optimizing Systems Through Measurement-Driven Iteration

The Optimization Tier (Modules 14–19) transforms systems-first pedagogy from *analysis* ("How much memory does this use?") into *optimization* ("How do I reduce memory by 4×?"). Where foundation modules taught calculating footprints and counting FLOPs, optimization modules teach systematic improvement through profiling-driven iteration.

This tier introduces three fundamental optimization concepts that complete the systems-first integration:

**1. Measurement-Driven Optimization.** Students learn the "measure first, optimize second" methodology through systematic profiling. Rather than guessing bottlenecks, they measure time, memory, and FLOPs to identify where optimization efforts yield maximum impact. This mirrors production ML engineering: profiling reveals that convolution consumes 80%+ training time, directing optimization focus appropriately.

**2. Trade-off Reasoning.** Optimization involves balancing competing objectives—accuracy, speed, memory, model size. Students measure these trade-offs empirically: quantization achieves 4× compression with 1–2% accuracy cost; pruning removes 90% of parameters with minimal accuracy impact; KV-caching achieves 10–100× speedup but increases memory. This reinforces that systems engineering requires navigating trade-offs, not absolutes.

**3. Implementation Matters.** Identical algorithms exhibit 100× performance differences based on implementation choices. Students experience this through vectorization: seven explicit loops (pedagogically transparent) versus NumPy matrix operations (production efficient). This teaches why production frameworks obsess over seemingly minor implementation details—performance differences compound across operations.

The Optimization Tier completes the systems-first integration arc: students who calculate memory in Module 01, count FLOPs in Module 09, and optimize deployment

in Modules 14–19 develop reflexive systems thinking. When encountering new ML techniques, they automatically ask: "How much memory? What's the computational complexity? What are the trade-offs?" These questions become automatic, not afterthoughts.

# 6 Deployment and Infrastructure

Translating curriculum design into effective classroom practice requires addressing integration models, infrastructure accessibility, and student support structures. This section presents deployment patterns validated through pilot implementations and institutional feedback.

## 6.1 Integration Models

TinyTorch supports three deployment models for different institutional contexts, ranging from standalone courses to supplementary tracks in existing curricula.

**Model 1: Self-Paced Learning (Primary Use Case)** serves individual learners, professionals, and researchers wanting framework internals understanding. Students work through modules at their own pace, selecting depth based on goals: complete all 20 modules for comprehensive systems knowledge, focus on Foundation (01–07) for autograd understanding, or target specific topics (Module 12 for attention mechanisms, Module 15 for quantization). The curriculum provides immediate feedback through local NBGrader validation, historical milestones for correctness proof, and progressive complexity enabling both intensive study (weeks) and distributed learning (months). This model requires zero infrastructure beyond Python and 4GB RAM, making it accessible worldwide.

**Model 2: Institutional Integration** enables universities to incorporate TinyTorch into existing ML courses. Options include: standalone 4-credit course (all 20 modules, complete systems coverage), half-semester module (Modules 01–09, foundation + CNN architectures), or optional honors track (selected modules for extra credit). Institutional deployment provides NBGrader autograding infrastructure, connection maps showing prerequisite dependencies, and milestone validation scripts. Lecture materials remain future work; current release supports lab-based or flipped-classroom formats where students implement concepts from textbook readings.

**Model 3: Team Onboarding** addresses industry use cases where ML teams want members to understand PyTorch internals. Companies can use TinyTorch for: new hire bootcamps (2–3 week intensive), internal training programs (distributed over quarters), or debugging workshops (focused modules like 05 Autograd, 12 Attention). The framework's PyTorch-inspired package structure and systems-first approach prepare engineers for understanding production frameworks and optimization workflows.

**Available Resources**: Current release provides module notebooks, NBGrader test suites, milestone validation scripts, and connection maps. Lecture slides for institutional courses remain future work (**??**), though self-paced learning requires no additional materials beyond the modules themselves.

## 6.2 Infrastructure and Accessibility

ML systems education faces an accessibility challenge: production ML courses typically require expensive GPU hardware ($500+ gaming laptops or cloud credits), 16GB+ RAM, CUDA-compatible environments, and Linux/WSL systems. These requirements create barriers for community college students, international learners in regions with limited cloud access, K-12 educators exploring ML internals, and institutions with modest computing budgets. Widening access to ML systems education requires reducing infrastructure barriers while maintaining pedagogical effectiveness (Reddi et al., b).

TinyTorch addresses this through CPU-only, pure Python implementation. The curriculum requires only dual-core 2GHz+ CPUs (no GPU needed), 4GB RAM (sufficient for CIFAR-10 training with batch size 32), 2GB storage (modules plus datasets), and any operating system supporting Python 3.8+ (Windows, macOS, or Linux). This enables deployment on Chromebooks via Google Colab, five-year-old budget laptops, and institutional computer labs. The pure Python design (NumPy-only, no compiled extensions) ensures cross-platform compatibility and pedagogical transparency: students can inspect every line of framework code without navigating CUDA kernels or hardware-specific optimizations. Included offline-first datasets (TinyDigits, TinyTalks) eliminate network dependencies during training, addressing connectivity challenges in developing countries and institutional environments with restricted internet access. Text-based ASCII connection maps further enhance accessibility for visually impaired students using screen readers.

### 6.2.1 Jupyter Environment Options

TinyTorch supports three deployment environments: **JupyterHub** (institutional server, 8-core/32GB supports 50 students), **Google Colab** (zero installation, best for MOOCs), and **local installation** (`pip install tinytorch`, best for self-paced learning).

### 6.2.2 NBGrader Autograding Workflow

**Student Submission Process**: (1) Student works in Jupyter notebook (local or cloud), (2) runs `nbgrader`

```
# Cell metadata defines grading parameters:
# nbgrader = {
#   "grade": true,
#   "grade_id": "tensor_memory",
#   "points": 2,
#   "locked": false,
#   "solution": true
# }

def memory_footprint(self):
    """Calculate tensor memory in bytes"""
    ### BEGIN SOLUTION
    return self.data.nbytes
    ### END SOLUTION
```

Listing 5: NBGrader cell metadata and solution structure.

`validate module_01.ipynb` for local correctness checking, (3) submits via LMS (Canvas/Blackboard) or Git (GitHub Classroom), (4) instructor runs `nbgrader autograde` on submitted notebooks, (5) grades and feedback posted to LMS.

**NBGrader Module Structure Example**: Each module uses NBGrader markdown cells to define assessment points and structure. For example, Module 01's memory profiling exercise:

This scaffolding makes educational objectives explicit while enabling automated grading. The `name` field identifies the exercise, `points` assigns weight, and the description provides context before students see code cells.

**Handling Autograder Edge Cases**: Pure Python convolution (Module 09) may exceed default 30-second timeout on slower hardware; we set 5-minute timeouts and provide vectorized reference solutions for comparison. Critical modules (05 Autograd, 09 CNNs) include manual review of 20% of submissions to catch conceptual errors missed by unit tests. All modules include `assert numpy.__version__ >= '1.20'` dependency validation.

**Projected Scalability**: Small courses (30 students) can grade in approximately 10 minutes per module on instructor laptop, medium courses (100 students) require approximately 30 minutes on dedicated grading server, while MOOCs (1000+ students) can achieve 2-hour turnaround via parallelized cloud autograding. These projections assume average grading time of 45 seconds per module submission on 4-core systems. Full-scale deployment validation planned for Fall 2025 (Section 7).

### 6.3 Automated Assessment Infrastructure

TinyTorch integrates NBGrader (Jupyter et al.) for scalable automated assessment. Each module contains:

- **Solution cells**: Scaffolded implementations with grade metadata
- **Test cells**: Locked autograded tests preventing modification
- **Immediate feedback**: Students validate correctness locally before submission
- **Point allocation**: Reflects pedagogical priorities (Module 05 Autograd: 100 points; Module 01 Tensor: 60 points)

This infrastructure enables deployment in MOOCs and large classrooms where manual grading proves infeasible. Instructors configure NBGrader to collect submissions, execute tests in sandboxed environments, and generate grade reports automatically.

**Important caveat**: NBGrader scaffolding exists but remains unvalidated at scale (Section 7). Automated assessment validity requires empirical investigation: Do tests measure conceptual understanding or syntax correctness? We scope this as "curriculum with autograding infrastructure" rather than "validated assessment system."

### 6.4 Package Organization

Unlike tutorial-style notebooks creating isolated code, TinyTorch modules export to a package structure inspired by PyTorch's API organization. Critically, *each completed module becomes immediately usable*—students build a working framework progressively, not isolated exercises. Module 01 exports to `tinytorch.core.tensor`, Module 09 to `tinytorch.nn.conv`, enabling import patterns familiar to PyTorch users that grow with each module completed.

As students complete modules, their framework accumulates capabilities. After Module 03, students can import and use layers; after Module 05, autograd enables training; after Module 09, CNNs become available. This progressive accumulation creates tangible evidence of progress—students see their framework grow from basic tensors to a complete ML system. Listing 6 illustrates how imports expand as modules are completed:

This design bridges educational and professional contexts. Students aren't "solving exercises"—they're building a framework they could ship. The package structure reinforces systems thinking: understanding how `torch.nn.Conv2d` relates to `torch.Tensor` requires grasping module organization, not just individual algorithms. More importantly, students experience the satisfaction of watching their framework grow from a single `Tensor` class to a complete system capable of training transformers—each module completion adds new capabilities they can immediately use.

```
# After Module 01: Basic tensors
from tinytorch.core import Tensor

# After Module 09: CNNs available
from tinytorch.nn import Conv2d, MaxPool2d,
    Linear
# Autograd active - gradients flow!

# After Module 13: Complete framework
from tinytorch.nn import Transformer, Embedding,
     Attention
```

Listing 6: Progressive imports: Framework capabilities grow module-by-module.

```
## Prerequisites & Progress
You've Built: Tensor, activations, layers,
    losses
You'll Build: Autograd system
You'll Enable: Training loops, optimizers

Connection Map:
Modules 01-04 → Autograd → Training (06-07)
(forward pass)   (backward)   (learning loops)
```

Listing 7: Module 05 connection map.

Export happens via nbdev (Howard and Gugger) directives (`#| default_exp core.tensor`) embedded in module notebooks. Students work in Jupyter's interactive environment while TinyTorch maintains source-of-truth in version-controlled Python files, enabling professional development workflows (Git, code review, CI/CD) within pedagogical context.

## 6.5 Connection Maps and Knowledge Integration

Every module begins with a **Connection Map** showing prerequisite modules, current module focus, and enabled future capabilities. This addresses Collins et al.'s cognitive apprenticeship (Collins et al.) by making expert knowledge structures visible:

Connection maps transform isolated modules into coherent curriculum. Students see *why* each module matters before implementation begins, reducing "I don't see the point" disengagement. Early feedback suggests these maps help students maintain big-picture understanding while working through implementation details.

## 6.6 Open Source Infrastructure

TinyTorch is released as open source to enable community adoption and evolution.[1] The repository includes instructor resources: `CONTRIBUTING.md` (guidelines for bug reports and curriculum improvements), `INSTRUCTOR.md` (30-minute setup guide, grading rubrics, common student errors), and `MAINTENANCE.md` (support commitment through 2027, succession planning for community governance).

**Maintenance Commitment**: The author commits to bug fixes and dependency updates through 2027, community pull request review within 2 weeks, and annual releases incorporating educator feedback. Community governance transition (2026–2027) will establish an educator advisory board and document succession planning to ensure long-term sustainability beyond single-author maintenance.

**Customization Support**: TinyTorch's modular design enables institutional adaptation: replacing datasets with domain-specific data (medical images, time series), adding modules (diffusion models, graph neural networks), adjusting difficulty through scaffolding modifications, or changing assessment approaches. Forks should maintain attribution (CC-BY-SA requirement) and ideally contribute improvements upstream.

## 6.7 Teaching Assistant Support

Effective deployment requires structured TA support beyond instructor guidance.

**TA Preparation**: TAs should develop deep familiarity with critical modules where students commonly struggle—Modules 05 (Autograd), 09 (CNNs), and 13 (Transformers)—by completing these modules themselves and intentionally introducing bugs to understand common error patterns. The repository provides `TA_GUIDE.md` documenting frequent student errors (gradient shape mismatches, disconnected computational graphs, broadcasting failures) and debugging strategies.

**Office Hour Demand Patterns**: Student help requests are expected to cluster around conceptually challenging modules, with autograd (Module 05) likely generating higher office hour demand than foundation modules. Instructors should anticipate demand spikes by scheduling additional TA capacity during critical modules, providing pre-recorded debugging walkthroughs, and establishing async support channels (discussion forums with guaranteed response times).

**Grading Infrastructure**: While NBGrader automates 70-80% of assessment, critical modules benefit from manual review of implementation quality and conceptual understanding. TAs should focus manual grading on: (1) code clarity and design choices, (2) edge case handling, (3) computational complexity analysis, and (4) memory profiling insights. Sample solutions and grading rubrics in `INSTRUCTOR.md` calibrate evaluation standards.

**Boundaries and Scaffolding**: TAs should guide students toward solutions through structured debugging

[1] Code released under MIT License, curriculum materials under Creative Commons Attribution-ShareAlike 4.0 (CC-BY-SA). Repository: https://github.com/harvard-edge/TinyTorch

questions rather than providing direct answers. When students reach unproductive frustration, TAs can suggest optional scaffolding modules (numerical gradient checking before autograd implementation, scalar autograd before tensor autograd) to build confidence through intermediate steps.

### 6.8 Student Learning Support

TinyTorch embraces productive failure (Kapur)—learning through struggle before instruction—while providing guardrails against unproductive frustration.

**Recognizing Productive vs Unproductive Struggle**: Productive struggle involves trying different approaches, making incremental progress (passing additional tests), and developing deeper understanding of error messages. Unproductive frustration manifests as repeated identical errors without new insights, random code changes hoping for success, or inability to articulate the problem. Students experiencing unproductive frustration should seek help rather than persisting solo.

**Structured Help-Seeking**: The repository provides debugging workflows: (1) self-debug using print statements and simple test cases, (2) consult common errors documentation for the module, (3) search discussion forums for similar issues, (4) post structured help requests with error messages and attempted solutions, (5) attend office hours with specific questions. This progression encourages independence while ensuring timely intervention.

**Flexible Pacing and Optional Scaffolding**: Students learn at different rates depending on background, learning style, and external commitments. TinyTorch supports multiple pacing modes—intensive (weeks), semester (distributed coursework), self-paced (professional development)—without prescriptive timelines. Students struggling with conceptual jumps can access optional intermediate modules providing additional scaffolding. No penalty attaches to slower pacing or scaffolding use; depth of understanding matters more than completion speed.

**Diverse Student Contexts**: The curriculum acknowledges students balance learning with work, caregiving, or health challenges. Flexible pacing enables participation from community college students, working professionals, international learners, and non-traditional students who might be excluded by rigid timelines or high-end hardware requirements. Pure Python deployment on modest hardware (4GB RAM, dual-core CPU) and screen-reader-compatible ASCII diagrams further broaden accessibility.

## 7 Discussion and Limitations

Building TinyTorch revealed insights about teaching ML systems from first principles. This section reflects on design lessons, acknowledges scope boundaries honestly, and outlines concrete empirical validation plans.

### 7.1 Scope: What's NOT Covered

TinyTorch prioritizes framework internals understanding over production ML completeness. The curriculum explicitly omits several critical production skills that require substantial additional complexity orthogonal to framework internals. GPU programming and hardware acceleration—including CUDA kernel optimization, memory hierarchies, tensor cores, mixed precision training with FP16/BF16/INT8 and gradient scaling, and hardware-specific optimization for TPUs and Apple Neural Engine—represent substantial domains requiring parallel programming expertise that would overwhelm students still mastering tensor operations and automatic differentiation.

Distributed training fundamentals including data parallelism (DistributedDataParallel, gradient synchronization), model parallelism (pipeline parallelism, tensor parallelism), and large-scale training systems (FSDP, DeepSpeed, Megatron-LM) similarly introduce communication complexity that extends beyond framework internals. Production deployment and serving skills—model compilation through TorchScript, ONNX, and TensorRT, serving infrastructure including batching, load balancing, and latency optimization, and MLOps tooling for experiment tracking, model versioning, and A/B testing—represent deployment infrastructure concerns distinct from framework understanding.

Advanced systems techniques such as gradient checkpointing (trading computation for memory), operator fusion and graph compilation, Flash Attention and memory-efficient attention variants, and dynamic versus static computation graphs, while important for production systems, introduce optimization complexity that obscures pedagogical transparency. These topics require substantial additional complexity—parallel programming semantics, hardware knowledge, deployment infrastructure—that would shift focus away from framework internals understanding.

TinyTorch teaches framework internals as foundation for GPU and distributed work, not as replacement. Complete production ML engineer preparation requires TinyTorch (internals) followed by PyTorch Distributed (GPU/multi-node), deployment courses (serving), and on-the-job experience. Students completing TinyTorch should pursue GPU and distributed training through PyTorch tutorials, NVIDIA Deep Learning Institute courses,

or advanced ML systems courses. The CPU-only design offers three pedagogical benefits: accessibility (students in regions with limited cloud computing access can complete curriculum on modest hardware), reproducibility (no GPU availability variability across institutions), and pedagogical focus (internals learning not confounded with hardware optimization).

### 7.2 Limitations: Understanding Scope

**Assessment infrastructure**: NBGrader scaffolding works in development but remains unvalidated for large-scale deployment. Grading validity requires investigation: Do tests measure conceptual understanding or syntax? Future work should validate through item analysis and transfer task correlation.

**Performance**: Pure Python executes 100–1000× slower than PyTorch (Table 3)—deliberate trade-off for transparency. Seven explicit loops reveal convolution complexity; unoptimized operations demonstrate why vectorization matters. Not for production use; students graduate to PyTorch/TensorFlow with internals understanding.

**Energy consumption**: While TinyTorch covers optimization techniques with significant energy implications (quantization achieving 4× compression, pruning enabling 10× model shrinkage), the curriculum does not explicitly measure or quantify energy consumption for students. The optimization modules discuss memory reduction and computational efficiency, implicitly touching on energy-saving benefits, but lack direct energy profiling tools or measurements. This omission means students understand that quantization reduces model size and pruning decreases computation, but may not connect these optimizations to concrete energy savings (joules/inference, watt-hours/training epoch). Future iterations could integrate energy profiling libraries or simulation tools to make energy efficiency an explicit learning objective alongside memory and latency optimization, particularly relevant for edge deployment and sustainable ML practices.

**Language**: Materials exist exclusively in English. Modular structure facilitates translation; community contributions welcome.

## 8 Future Work

TinyTorch's current implementation emphasizes hands-on measurement within the framework—students profile actual TinyTorch code, measure real memory consumption, and time genuine operations. This direct measurement approach teaches systems thinking through observable behavior. However, extending systems education beyond TinyTorch's CPU-only scope requires complementary approaches: analytical models for reasoning about hardware we don't have access to, and simulators for exploring distributed systems we cannot physically deploy. We organize future directions into three categories reflecting different pedagogical goals.

### 8.1 Systems Extensions: Analytical Models and Simulators

TinyTorch's CPU-only design prioritizes pedagogical transparency, but students benefit from understanding GPU acceleration and distributed training without requiring expensive hardware. We propose integrating **analytical performance models** and **systems simulators** to enable hardware-agnostic systems education.

**Roofline Models for GPU Performance Analysis** Future extensions could enable students to compare TinyTorch CPU implementations against PyTorch GPU equivalents through roofline models (Williams et al.). Rather than writing CUDA code, students would profile existing implementations to understand: (1) memory hierarchy differences (CPU cache levels L1/L2/L3 versus GPU global/shared/register memory), (2) parallelism benefits (sequential CPU loops versus massively parallel GPU execution with thousands of threads), (3) roofline analysis techniques (plotting achieved performance against hardware limits to identify compute-bound versus memory-bound operations), and (4) mixed precision advantages (profiling FP32 versus FP16 training speed/memory tradeoffs). Students would run instrumented PyTorch code alongside TinyTorch implementations, measuring wall-clock time, memory usage, and FLOPs utilization. The roofline model visualization shows why GPUs excel at ML workloads: high arithmetic intensity operations (matrix multiplication) approach peak FLOPs, while memory-bound operations (element-wise activations) hit bandwidth limits. This awareness without implementation maintains TinyTorch's accessibility while preparing students for GPU programming courses.

**ASTRA-sim for Distributed Training Simulation** Understanding distributed training communication patterns and scalability challenges requires simulation-based pedagogy, not multi-GPU clusters. Future extensions could integrate ASTRA-sim (Rashidi et al.; Samajdar et al.), a distributed ML training simulator enabling single-machine exploration of multi-device concepts. Rather than requiring 8-GPU clusters, students would simulate multi-device training, exploring: (1) data parallelism basics (gradient synchronization via all-reduce across virtual workers, analyzing communication over-

head versus compute time), (2) scalability analysis (measuring weak versus strong scaling, identifying communication bottlenecks as worker count increases), (3) network topology impact (comparing ring all-reduce, tree all-reduce, and hierarchical strategies through ASTRA-sim's topology modeling), and (4) pipeline parallelism introduction (simulating model partitioning across devices, analyzing pipeline bubbles and micro-batching strategies). This simulation-based approach maintains TinyTorch's pedagogical principle: understanding systems through transparent implementation and measurement, not black-box hardware access. Students would understand why gradient synchronization limits distributed training scalability, how network bandwidth affects multi-node training, and when to apply different parallelism strategies based on model and hardware characteristics.

**Energy and Power Profiling** Edge deployment and sustainable ML require understanding energy consumption. Future extensions could integrate power profiling tools enabling students to measure energy costs (joules per inference, watt-hours per training epoch) alongside latency and memory. Students would profile TinyTorch implementations to understand: (1) energy-memory tradeoffs (quantization's 4× memory reduction translates to proportional energy savings), (2) sparse computation benefits (structured sparsity reducing both FLOPs and energy), and (3) deployment platform differences (comparing CPU, GPU, mobile NPU energy profiles). This connects optimization techniques (already taught in Modules 15–18) to concrete sustainability metrics, particularly relevant for edge AI where battery life constrains deployment.

**The Three-Tier Systems Pedagogy** These extensions complete a three-tier systems education approach: (1) **Direct measurement** (current TinyTorch): profile actual code, measure real memory, time genuine operations on accessible hardware; (2) **Analytical models** (roofline, energy models): reason about hardware behavior through first-principles performance bounds without requiring physical access; (3) **Simulation** (ASTRA-sim, distributed training): explore distributed systems and communication patterns impossible to deploy on single machines. This progression mirrors computer architecture education: students first measure real systems, then learn analytical modeling for design space exploration, finally simulate systems too complex or expensive to build. Additional extensions could include cache simulators for understanding memory hierarchy effects, custom accelerator modeling for hardware-software co-design exploration, and sparse tensor operation analysis for structured pruning patterns.

## 8.2 Empirical Validation

While systems extensions represent the primary technical future work, empirical validation remains important for establishing pedagogical effectiveness. Planned studies include deploying TinyTorch in university curricula to measure whether embedding systems concepts from the start improves production ML readiness compared to algorithm-only approaches, and whether students who build frameworks transfer knowledge to PyTorch/TensorFlow more effectively than students who only use these frameworks. Specific research questions include evaluating progressive disclosure's impact on cognitive load, measuring transfer effectiveness through debugging tasks, and assessing long-term retention of systems concepts (memory profiling, complexity reasoning) six months post-course. These studies require validated instruments, control groups, and longitudinal data collection as part of future research programs.

## 8.3 Curriculum Extensions: Fundamentals vs. Production Scope

TinyTorch's 20-module curriculum deliberately stops at fundamental systems concepts (tensors through optimized transformers). Extending beyond this scope presents a fundamental pedagogical tension: **teaching framework internals versus becoming a production framework**. Every additional topic risks diluting the core mission—building systems understanding from scratch—by adding API surface without proportional pedagogical depth.

Potential extensions exist (graph neural networks, diffusion models, reinforcement learning, federated learning), but each must justify inclusion through systems pedagogy rather than completeness. The question is not "Can TinyTorch implement this?" but "Does implementing this teach fundamental systems concepts students cannot learn through existing modules?" Graph convolutions, for example, might teach sparse tensor operations and message-passing patterns; diffusion models might illuminate iterative refinement and noise scheduling trade-offs. However, these risk becoming feature additions rather than conceptual foundations.

Community forks demonstrate extensibility within this philosophy: quantum ML variants replace classical tensors with quantum state vectors (teaching quantum circuit depth vs classical memory); robotics-focused forks add RL infrastructure emphasizing simulation overhead and real-time constraints. These extensions succeed when they maintain TinyTorch's core princi-

ple: **every line of code teaches a systems concept**. The curriculum remains intentionally incomplete as a production framework—its completeness lies in covering foundational systems thinking applicable across all ML architectures, not in implementing every contemporary model family.

### 8.4 Community Building and Adoption

TinyTorch serves as the hands-on companion to the Machine Learning Systems textbook, providing practical implementation experience alongside theoretical foundations. Adoption will be measured through multiple channels: (1) **Educational adoption**: tracking course integrations, student enrollment, and instructor feedback across institutions; (2) **AI Olympics community**: inspired by MLPerf benchmarking, the AI Olympics leaderboard would create competitive systems engineering challenges where students submit optimized implementations competing across accuracy, speed, compression, and efficiency tracks—building community engagement and peer learning; (3) **Open-source metrics**: GitHub stars, forks, contributions, and community discussions indicating active use beyond formal coursework. This multi-faceted approach recognizes that educational impact extends beyond traditional classroom metrics to include community building, peer learning, and long-term skill development. The AI Olympics platform particularly enables students to see how their implementations compare globally, fostering systems thinking through competitive optimization while maintaining educational focus on understanding internals rather than achieving state-of-the-art performance.

## 9 Conclusion

Machine learning systems engineering benefits from understanding framework internals—why `loss.backward()` traverses computational graphs, why Adam requires 2× optimizer state memory (momentum and variance), why attention scales $O(N^2)$. TinyTorch addresses this through three pedagogical contributions: progressive disclosure managing complexity via monkey-patching, systems-first integration embedding memory profiling from Module 01, and historical milestone validation proving correctness through recreating 70 years of ML breakthroughs.

**For practitioners**: TinyTorch offers framework internals education through building PyTorch components from scratch. Understanding autograd implementation aids debugging gradient flow issues. Understanding optimizer memory costs informs deployment decisions. Understanding attention complexity guides architecture choices. This systems knowledge should transfer to production framework usage.

**For researchers**: TinyTorch provides replicable infrastructure for studying ML systems pedagogy. The curriculum embodies testable hypotheses: Does progressive disclosure reduce cognitive load? Does systems-first integration improve production readiness? Do historical milestones increase engagement? Open-source release enables empirical investigation across institutions.

**For educators**: TinyTorch supports three integration models—self-paced learning (primary use case, zero infrastructure), institutional courses (classroom deployment with NBGrader), and team onboarding (industry training). The modular structure enables selective adoption based on learning goals and institutional constraints.

The complete codebase, curriculum materials, and assessment infrastructure are openly available at `tinytorch.ai`. We invite the community to adopt, adapt, extend, and empirically evaluate these pedagogical patterns. ML systems education remains an open research problem; TinyTorch contributes one approach grounded in learning theory and accessible to diverse audiences worldwide.

## References


F , ROSENBLATT. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386–408, 1958. ISSN 0033-295X. doi: 10.1037/h0042519.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, 2nd edition, 2006.

Jerome S. Bruner. *The Process of Education*. Harvard University Press, Cambridge, MA, 1960.

Tianqi Chen and Zico Zheng. Cs 10-414/614: Deep learning systems, 2022. URL https://dlsyscourse.org/.

Allan Collins, John Seely Brown, and Susan E. Newman. Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics. In *Knowing, Learning, and Instruction*, pages 453–494. Routledge. ISBN 9781315044408. doi: 10.4324/9781315044408-14. URL https://doi.org/10.4324/9781315044408-14.

George Hotz and contributors. tinygrad: A simple and powerful neural network framework, 2023. URL https://github.com/tinygrad/tinygrad.

Jeremy Howard and Sylvain Gugger. Fastai: A layered api for deep learning. *Information*, 11(2):108. ISSN

2078-2489. doi: 10.3390/info11020108. URL https://doi.org/10.3390/info11020108.

Justin Johnson, Andrej Karpathy, and Li Fei-Fei. Cs231n: Convolutional neural networks for visual recognition, 2016. URL http://cs231n.stanford.edu/.

Project Jupyter, Douglas Blank, David Bourgin, Alexander Brown, Matthias Bussonnier, Jonathan Frederic, Brian Granger, Thomas Griffiths, Jessica Hamrick, Kyle Kelley, M Pacer, Logan Page, Fernando Pérez, Benjamin Ragan-Kelley, Jordan Suchow, and Carol Willing. nbgrader: A tool for creating and grading assignments in the jupyter notebook. *Journal of Open Source Education*, 2(11):32. ISSN 2577-3569. doi: 10.21105/jose.00032. URL https://doi.org/10.21105/jose.00032.

Manu Kapur. Productive failure. *Cognition and Instruction*, 26(3):379–424. ISSN 0737-0008,1532-690X. doi: 10.1080/07370000802212669. URL https://doi.org/10.1080/07370000802212669.

Andrej Karpathy. micrograd: A tiny scalar-valued autograd engine and neural net library, 2022. URL https://github.com/karpathy/micrograd.

Keller Executive Search. Ai & machine-learning talent gap 2025, 2025. URL https://www.kellerexecutivesearch.com/intelligence/ai-machine-learning-talent-gap-2025/.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Jean Lave and Etienne Wenger. *Situated Learning*. Cambridge University Press. ISBN 9780521413084,9780521423748,9780511815355. doi: 10.1017/cbo9780511815355. URL https://doi.org/10.1017/cbo9780511815355.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324. ISSN 0018-9219. doi: 10.1109/5.726791. URL https://doi.org/10.1109/5.726791.

Donella H. Meadows. *Thinking in Systems: A Primer*. Chelsea Green Publishing, White River Junction, VT, 2008.

Jan H. F. Meyer and Ray Land. Threshold concepts and troublesome knowledge: Linkages to ways of thinking and practising within the disciplines. In C. Rust, editor, *Improving Student Learning: Theory and Practice Ten Years On*, pages 412–424. Oxford Centre for Staff and Learning Development, Oxford, 2003.

David N. Perkins and Gavriel Salomon. Transfer of learning. In Torsten Husen and T. Neville Postlethwaite, editors, *International Encyclopedia of Education*. Pergamon Press, Oxford, UK, 2nd edition, 1992.

Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. Astra-sim: Enabling sw/hw co-design exploration for distributed dl training platforms. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, volume 43, pages 81–92. IEEE. doi: 10.1109/ispass48437.2020.00018. URL https://doi.org/10.1109/ispass48437.2020.00018.

Vijay Janapa Reddi. Mlsysbook.ai: Principles and practices of machine learning systems engineering. In *2024 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 41–42. IEEE, IEEE. doi: 10.1109/codes-isss60120.2024.00015. URL https://doi.org/10.1109/codes-isss60120.2024.00015.

Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. Mlperf inference benchmark. *arXiv preprint arXiv:1911.02549*, a. URL http://arxiv.org/abs/1911.02549v2.

Vijay Janapa Reddi, Brian Plancher, Susan Kennedy, Laurence Moroney, Pete Warden, Anant Agarwal, Colby Banbury, Massimo Banzi, Matthew Bennett, Benjamin Brown, Sharad Chitlangia, Radhika Ghosal, Sarah Grafman, Rupert Jaeger, Srivatsan Krishnan, Maximilian Lam, Daniel Leiker, Cara Mann, Mark Mazumder, Dominic Pajak, Dhilan Ramaprasad, J. Evan Smith, Matthew Stewart, and Dustin Tingley. Widening access to applied machine learning with tinyml. *arXiv preprint arXiv:2106.04008*, b. URL http://arxiv.org/abs/2106.04008v2.

Robert Half. New robert half research reveals severity of the technology skills gap amid talent shortage, May 2024. URL https://press.roberthalf.com/2024-05-08-New-Robert-Half-Research-Reveals

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536. ISSN 0028-0836,1476-4687. doi: 10.1038/323533a0. URL https://doi.org/10.1038/323533a0.

Sasha Rush. Minitorch: A diy teaching library for machine learning engineers, 2020. URL https://minitorch.github.io/.

Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. A systematic methodology for characterizing scalability of dnn accelerators using scale-sim. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 1–14. IEEE. doi: 10.1109/ispass48437.2020.00016. URL https://doi.org/10.1109/ispass48437.2020.00016.

John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2): 257–285. ISSN 0364-0213,1551-6709. doi: 10.1207/s15516709cog1202_4. URL https://doi.org/10.1207/s15516709cog1202_4.

PyTorch Team. Pytorch 0.4.0 release notes: Tensor and variable merge, 2018. URL https://github.com/pytorch/pytorch/releases/tag/v0.4.0.

TensorFlow Team. Tensorflow 2.0: Easy model building with keras and eager execution, 2019. URL https://www.tensorflow.org/guide/effective_tf2.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N.Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. URL https://doi.org/10.65215/pc26a033.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline. *Communications of the ACM*, 52 (4):65–76. ISSN 0001-0782,1557-7317. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

Judith S. Wooster and Seymour Papert. Mindstorms: Children, computers, and powerful ideas. *The English Journal*, 71(8):60. ISSN 0013-8274. doi: 10.2307/816450. URL https://doi.org/10.2307/816450.

Aston Zhang, Zachary C. Lipton, Mu Li 0003, and Alexander J. Smola. Dive into deep learning., 2021. URL https://arxiv.org/abs/2106.11342.