



Build Your Own Machine Learning Framework

From Tensors to Systems

Vijay Janapa Reddi
Harvard University

tinytorch.ai

Abstract

Machine learning systems engineering requires understanding framework internals: why optimizers consume memory, when computational complexity becomes prohibitive, how to navigate accuracy-latency-memory tradeoffs. Yet current ML education separates algorithms from systems—students learn gradient descent without measuring memory, attention mechanisms without profiling costs, training without understanding optimizer overhead. This divide leaves graduates unable to debug production failures or make informed engineering decisions. We present TinyTorch, a build-from-scratch curriculum where students implement PyTorch’s core components (tensors, autograd, optimizers, neural networks) to gain framework transparency. Three pedagogical patterns address the gap: **progressive disclosure** gradually reveals complexity (gradient features exist from Module 01, activate in Module 05); **systems-first curriculum** embeds memory profiling from the start; **historical milestone validation** recreates nearly 70 years of ML breakthroughs (1958–2025) using exclusively student-implemented code. These patterns are grounded in learning theory (situated cognition, cognitive load theory) but represent testable hypotheses requiring empirical validation. The 20-module curriculum provides complete open-source infrastructure at mlsysbook.ai/tinytorch.

1 Introduction

In “The Bitter Lesson,” Rich Sutton observes that the history of artificial intelligence teaches a counterintuitive truth: general methods that leverage computation ultimately defeat cleverly-designed, domain-specific ap-

proaches (Sutton, 2019). Deep learning surpassed hand-crafted features in computer vision. Large language models outperformed linguistic rule systems. AlphaZero mastered games through self-play rather than encoded heuristics. A fundamental driver behind each breakthrough is computational efficiency—the ability to effectively scale learning systems to leverage available hardware. Yet while we have learned this lesson algorithmically, building ever-larger models that demonstrate its truth, we have not embedded it pedagogically. Our educational systems continue to separate the teaching of machine learning algorithms from the systems knowledge required to achieve computational scale.

This pedagogical gap creates a systems efficiency crisis. Modern ML models often fail not from algorithmic limitations but from hitting computational walls. Transformer attention mechanisms scale as $O(N^2)$ with sequence length (Vaswani et al.), causing memory exhaustion before accuracy plateaus. Distributed training requires understanding gradient synchronization overhead that can eliminate parallel speedup. Production deployments crash from subtle memory leaks in tensor caching and reference cycles. The paradox is stark—we know empirically that scale wins, that computational efficiency enables the breakthroughs Sutton documents, yet we do not teach students how to achieve this scale. They can train models but cannot explain why gradient accumulation reduces memory usage or when activation checkpointing becomes necessary.

This crisis directly impacts the ML workforce. Industry surveys indicate significant demand-supply imbalances for ML systems engineers (Heffernan; Search, 2025), with surveys suggesting that a substantial portion of executives cite talent shortage as their primary barrier to AI adoption (Search, 2025). These are not the

scientists who invent new architectures but the engineers who make existing architectures computationally viable—who understand when mixed precision training preserves accuracy, how gradient checkpointing trades compute for memory, and why distributed training introduces synchronization bottlenecks. They are often the bottleneck to realizing the promise of computational scaling that enables general methods to triumph.

The knowledge these engineers need is fundamentally about systems mental models. Understanding *why* Adam requires $2\times$ optimizer state memory requires visualizing optimizer state buffers. Predicting *when* batch sizes must shrink to fit GPU memory requires internalizing memory hierarchy latencies. Navigating accuracy-latency-memory tradeoffs in production systems requires understanding collective communication patterns and their overhead (Meadows, 2008). This tacit systems knowledge—how frameworks manage memory, schedule operations, and optimize execution—cannot be developed through high-level API usage alone. It emerges from building these systems, from implementing tensor operations that reveal memory access patterns, from constructing computational graphs that expose optimization opportunities.

Current ML education often fails to develop these mental models through a strict separation: algorithms courses teach backpropagation mathematics while systems courses teach distributed computing, with limited bridges between them. Students learn gradient descent’s convergence properties but not its memory footprint. They implement neural networks using framework APIs but never see how those APIs translate to memory allocations and kernel launches. They can mathematically derive the chain rule but cannot explain how autograd implements it efficiently through dynamic graph construction and topological traversal. This separation leaves students unprepared for the systems engineering roles industry desperately needs.

We present TinyTorch, a 20-module curriculum that teaches machine learning as computational systems engineering by building a PyTorch-compatible framework from pure Python primitives. Designed as a hands-on companion to the *Machine Learning Systems* textbook (Reddi), TinyTorch makes systems efficiency tangible—students implement tensor operations while measuring memory consumption, build autograd while profiling computational graphs, create optimizers while tracking state overhead. Each module reinforces insights inspired by the systems imperative the Bitter Lesson reveals: computational efficiency, not algorithmic cleverness alone, drives ML progress. Students don’t just learn *that* Conv2d achieves $109\times$ parameter efficiency over dense layers, they *implement* sliding window convolu-

tion and *measure* the difference directly through profiling code they wrote. Figure 1 illustrates this progression: from PyTorch’s black-box APIs, through building internals like optimizers, to training transformers where every import is student-implemented code.

Building systems knowledge alongside ML fundamentals presents three pedagogical challenges: teaching systems thinking early without overwhelming beginners (Section 5), managing cognitive load when teaching both algorithms and implementation (Section 4), and validating student understanding through concrete milestones (Section 3.4). TinyTorch addresses these through curriculum design inspired by compiler courses (Aho et al., 2006)—students build a complete system incrementally, with each module adding functionality while maintaining a working implementation. Figure 2 illustrates this progression: tensors (Module 01) enable activations (02) and layers (03), which feed into autograd (05), powering optimizers (06) and training (07). Each completed module becomes immediately usable: after Module 03, students build neural networks; after Module 05, automatic differentiation enables training; after Module 13, transformers support language modeling. This structure enables students to construct mental models gradually while seeing immediate results.

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 completion, semester integration (parallel with lectures), or self-paced professional development.

This paper makes three contributions, each inspired by the systems imperative the Bitter Lesson reveals:

1. **Systems-First Curriculum Architecture** (Sections 3 and 5): A 20-module learning path integrating memory profiling, computational complexity, and performance analysis from Module 01 onwards. Students discover systems constraints through direct measurement rather than abstract instruction. Grounded in situated cognition (Lave and Wenger) and constructionism (Wooster and Papert).
2. **Progressive Disclosure Pattern** (Section 4): A scaf-

```

1 import torch.nn as nn
2 import torch.optim as optim
3
4 # How much memory?
5 model = nn.Linear(784, 10)
6
7 # Why does Adam need more
8 # memory than SGD?
9 optimizer = optim.Adam(
10     model.parameters())
11 loss_fn = nn.CrossEntropyLoss()
12
13 for epoch in range(10):
14     for x, y in dataloader:
15         pred = model(x)
16         loss = loss_fn(pred, y)
17         loss.backward() # Magic?
18         optimizer.step() # How?

```

(a) PyTorch: Black box usage

```

1 class Adam:
2     def __init__(self, params,
3                  lr=0.001):
4         self.params = params
5         self.lr = lr
6         # 2x optimizer state:
7         # momentum + variance
8         self.m = [zeros_like(p)
9                   for p in params]
10        self.v = [zeros_like(p)
11                  for p in params]
12
13    def step(self):
14        for i, p in enumerate(
15            self.params):
16            self.m[i] = 0.9 * self.m[i]
17                + 0.1 * p.grad
18            self.v[i] = 0.99 * self.v[i]
19                + 0.001
20                    * p.grad**2
21            p.data -= self.lr *
22                self.m[i] /
23                (self.v[i].sqrt() + 1e-8)

```

(b) TinyTorch: Build internals

```

1 # After Module 13: train
2 # transformers with YOUR code
3 from tinytorch.nn import (
4     Transformer, Embedding)
5 from tinytorch.optim import
6     Adam
7 from tinytorch.data import
8     DataLoader
9
10 model = Transformer(
11     vocab=1000, d_model=64,
12     n_heads=4, n_layers=2)
13 opt = Adam(model.parameters())
14
15 for batch in DataLoader(data):
16     loss = model(batch.x,
17                  batch.y)
18     loss.backward() # Yours!
19     opt.step() # Yours!
# You understand WHY it works
# because you built it all!

```

(c) TinyTorch: The culmination

Figure 1. From framework user to engineer. (a) PyTorch: high-level APIs hide internals. (b) TinyTorch: students implement components like Adam, understanding memory costs and update rules. (c) After Module 13, students train transformers using exclusively their own code—every import is something they built.

folding technique using monkey-patching to reveal Tensor complexity gradually while maintaining a unified mental model. Module 01 implementations continue working when autograd activates in Module 05. Grounded in cognitive load theory (Sweller) and cognitive apprenticeship (Collins et al.).

3. **Open Educational Infrastructure** (Sections 3, 6 and 7): Complete open-source curriculum with NBGrader assessment (Jupyter et al.), historical milestone validation, and PyTorch-inspired package architecture enabling community adoption and empirical research.

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 targeting 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 existing 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

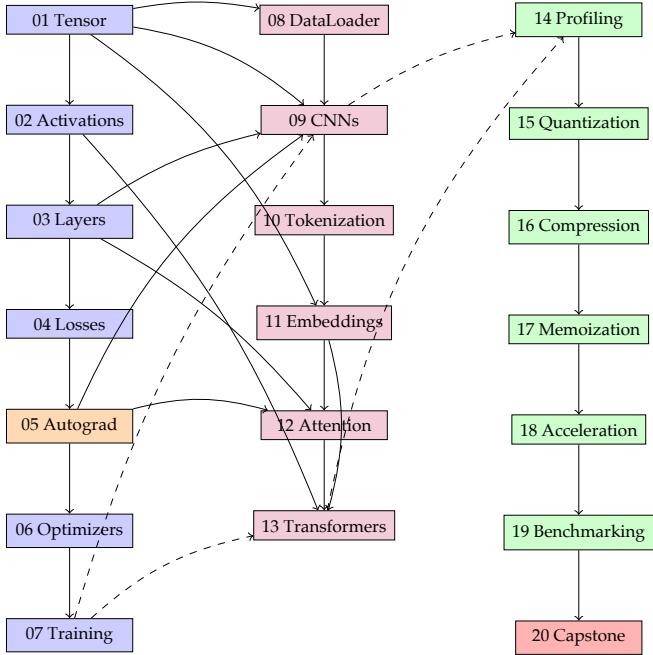


Figure 2. Module dependency flow shows how students build a complete ML framework incrementally. Foundation modules (blue, M01-07) provide core tensor operations and training infrastructure. These enable architecture modules (purple, M08-13) where students implement CNNs and transformers using only their own code. Optimization modules (green, M14-19) teach production concerns: profiling, quantization, and deployment. Solid arrows show direct dependencies (e.g., autograd M05 requires tensors M01); dotted lines show cross-tier integration (e.g., benchmarking M19 requires all architectures). This structure mirrors compiler courses: each module builds on previous work, creating systems thinking through component integration.

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 hack-

ability 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.](#)) 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 implementation and practical application.

mic 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 back-propagation”), 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. Key distinctions: versus micrograd, complete framework scope with assessment infrastructure; versus MiniTorch, systems-first emphasis with unified API evolution; versus tinygrad, scaffolded progression without compiler prerequisites; versus d2l.ai and fast.ai, bottom-up framework construction rather than algorithm mastery or top-down application focus.

2.4 Pedagogical Precedent: Build-to-Understand Systems Education

TinyTorch joins a long pedagogical tradition in systems education: teaching complex systems by having students build simplified, transparent implementations. This approach emerged because production systems optimize for performance rather than comprehension, making their internals inaccessible to learners.

MINIX ([Tanenbaum, 1987](#)) exemplifies this pattern. Tanenbaum created a teaching operating system despite Unix’s existence because production kernels are too complex for students to hold in their heads. Students who implement a microkernel understand scheduler mechanics, memory management, and process isolation in ways that remain opaque to Linux users. MINIX shaped generations of systems engineers and famously inspired Linux itself. The parallel to TinyTorch is direct: PyTorch is

Table 1. Framework comparison positions TinyTorch’s pedagogical role. Educational frameworks (micrograd, MiniTorch, tinygrad) prioritize learning over production use. Production frameworks (PyTorch, TensorFlow) prioritize performance and scalability. TinyTorch bridges both: students learn framework internals through implementation, then transfer that knowledge to production frameworks with deeper systems understanding.

| Framework | 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 |

Unix; TinyTorch is MINIX.

Compiler education follows identical logic. Despite LLVM and GCC representing decades of engineering excellence, compiler courses continue teaching from-scratch implementation (Aho et al., 2006; Appel and Palsberg, 2002). Students build lexical analyzers, construct abstract syntax trees, implement type checkers, and generate code. The Tiger compiler (Appel and Palsberg, 2002) became canonical precisely because industrial compilers are too vast for pedagogical dissection. Students who build compilers understand optimization passes and code generation in ways that framework users cannot.

Operating systems education institutionalized this approach through purpose-built teaching systems. Nachos (Christopher et al., 1993) at Berkeley and Pinotos (Pfaff et al., 2009) at Stanford provided hackable kernels where students implement threads, schedulers, virtual memory, and file systems. These systems never aimed to replace Linux; they aimed to reveal what Linux hides. The pedagogical insight transfers directly: TinyTorch is to PyTorch what Nachos was to Unix.

Perhaps most fundamentally, SICP (Abelson et al., 1996) taught programming through building interpreters and evaluators from scratch, despite real languages already existing. The core philosophy: to understand a system, build one. This principle, validated across five

decades of CS education, grounds TinyTorch’s design.

These canonical precedents share common characteristics: (1) production systems grew too complex to learn from directly, (2) lightweight reconstructions provided genuine insight unavailable through usage alone, (3) educational versions outlived production systems in pedagogical value, and (4) they trained systems thinkers rather than mere users. TinyTorch applies this proven pattern to ML frameworks at the precise moment when PyTorch and TensorFlow have matured beyond pedagogical accessibility, and when industry desperately needs engineers who understand framework internals.

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 TinyTorch Architecture

This section presents the 20-module curriculum structure, organized into four tiers that progressively build a complete ML framework. Each module enforces prerequisite mastery, ensuring students build on solid foundations.

Integration with ML Systems Curriculum. TinyTorch serves as the hands-on implementation companion to

the *Machine Learning Systems* textbook (Reddi, 2025) (`mlsysbook.ai`), creating synergy between theoretical foundations and systems engineering practice. While the textbook covers the full ML lifecycle—data engineering, training architectures, deployment monitoring, robust operations, and sustainable AI—TinyTorch provides the complementary experience of building core infrastructure from first principles. This integration enables a complete educational pathway: students study production ML systems architecture in the textbook (Chapter 4: distributed training patterns, Chapter 7: quantization strategies), then implement those same abstractions in TinyTorch (Module 05: autograd for backpropagation, Module 15: INT8 quantization). The two resources address different aspects of the same educational gap: understanding both *how production systems work* (textbook’s systems architecture perspective) and *how to build them yourself* (TinyTorch’s implementation depth). This theory-practice coupling mirrors industry workflows where ML systems engineers both understand architectural patterns and implement infrastructure components.

3.1 Prerequisites

As established in Section 1, TinyTorch targets students transitioning 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.

3.2 The 3-Tier Learning Journey + Capstone

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 Capstone (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 multi-

```

1 class Tensor:
2     def __init__(self, data):
3         self.data = np.array(data, dtype=np.float32)
4         self.shape = self.data.shape
5
6     def memory_footprint(self):
7         """Calculate exact memory in bytes"""
8         return self.data.nbytes
9
10    def __matmul__(self, other):
11        if self.shape[-1] != other.shape[0]:
12            raise ValueError(
13                f"Shape mismatch: {self.shape} @ {other.
14                shape}")
15        return Tensor(self.data @ other.data)

```

Listing 1. Tensor with memory profiling from Module 01.

plication (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: Rosenblatt’s Perceptron, Minsky and Papert’s XOR solution, and Rumelhart et al.’s backpropagation targeting 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. While Module 07 implements basic training loops with manual batching (simple iteration over pre-batched arrays), Module 08 refactors this into production-quality data loading, a pedagogical pattern of “make it work, then make it right.” Students first understand training mechanics (forward pass, loss, backward, update), then learn proper data pipeline engineering. 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-

Table 2. Module progression integrates ML algorithms with systems concepts from the start. Each module teaches both “what” (ML technique) and “how much” (memory / compute costs). Foundation tier (M01-07) establishes core operations with explicit resource tracking. Architecture tier (M08-13) applies these foundations to CNNs and transformers. Optimization tier (M14-19) adds production concerns: profiling, quantization, deployment. This dual-concept approach ensures students never learn algorithms without understanding their systems implications.

| Mod | Tier | Module Name | ML Concept | Systems Concept |
|----------------------------------|------|----------------|---|--|
| Foundation Tier (01–07) | | | | |
| 01 | Fnd | Tensor | Multidimensional arrays, broadcasting | Memory footprint (nbytes), dtype sizes, contiguous layout |
| 02 | Fnd | Activations | ReLU, Sigmoid, Tanh, GELU, Softmax | Numerical stability (exp overflow), vectorization |
| 03 | Fnd | Layers | Linear, Xavier initialization | Parameter vs activation memory, weight layout |
| 04 | Fnd | Losses | Cross-entropy, MSE, log-sum-exp trick | Numerical stability (log(0)), gradient magnitude |
| 05 | Fnd | Autograd | Computational graphs, chain rule, backprop | Gradient memory ($2 \times$ momentum, $3 \times$ Adam) |
| 06 | Fnd | Optimizers | SGD, Momentum, Adam, AdamW | Optimizer state memory, in-place updates |
| 07 | Fnd | Training | Cosine scheduling, gradient clipping | Peak memory lifecycle, checkpoint tradeoffs |
| Architecture Tier (08–13) | | | | |
| 08 | Arch | DataLoader | Dataset abstraction, batching, shuffling | Iterator protocol, batch collation overhead |
| 09 | Arch | Spatial (CNNs) | Conv2d, pooling, padding, stride | im2col expansion, 7-loop $O(B \times C \times H \times W \times K^2)$ |
| 10 | Arch | Tokenization | BPE, vocabulary, special tokens | Vocab size \leftrightarrow sequence length tradeoff |
| 11 | Arch | Embeddings | Token + positional (sinusoidal/learned) | Sparse gradient updates, embedding table memory |
| 12 | Arch | Attention | Scaled dot-product, causal masking | $O(N^2)$ memory, attention score materialization |
| 13 | Arch | Transformers | Multi-head attention, LayerNorm, MLP | KV cache sizing, per-layer memory profile |
| Optimization Tier (14–19) | | | | |
| 14 | Opt | Profiling | Time / memory / FLOPs measurement | Bottleneck identification, measurement overhead |
| 15 | Opt | Quantization | INT8, scale / zero-point calibration | 4 \times compression, quantization error propagation |
| 16 | Opt | Compression | Magnitude pruning, knowledge distillation | Sparsity patterns, teacher-student memory |
| 17 | Opt | Memoization | KV-cache for autoregressive generation | $O(n^2) \rightarrow O(n)$ caching, memory-compute tradeoff |
| 18 | Opt | Acceleration | Vectorization, memory access patterns | Cache locality, SIMD utilization |
| 19 | Opt | Benchmarking | Statistical comparison, multiple runs | Confidence intervals, warm-up protocols |
| Capstone (20) | | | | |
| 20 | Cap | Capstone | End-to-end optimized system, benchmark submission | MLPerf-style metrics, JSON validation, leaderboard integration |

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($3 \rightarrow 32$, 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) targeting 65–75% 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 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 $2\times$ yields only $1.53\times$ overall speedup because the remaining 30% becomes the new bottleneck. This teaches that optimization is iterative and measurement-driven. Quantization (15) achieves $4\times$ compression (FP32 \rightarrow INT8) with 1–2% accuracy cost. Compression (16) applies pruning and distillation for $10\times$ 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 ($1+2+\dots+100$). By caching these values and reusing them, students transform $O(n^2)$ gen-

eration into $O(n)$, achieving $10\text{--}100\times$ speedup and understanding why this optimization is essential in systems like ChatGPT and Claude for economically viable inference. Acceleration (18) vectorizes convolution for $10\text{--}100\times$ gains. Benchmarking (19) teaches rigorous performance measurement.

Capstone (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 $10\times$ faster inference, $4\times$ smaller size, and sub-100ms latency while maintaining accuracy. Students learn professional ML systems workflow: benchmark baseline model, apply optimizations from Modules 14–19 (quantization, pruning, acceleration), benchmark optimized version, measure improvement metrics (speedup, compression ratio, accuracy delta), and analyze optimization tradeoffs. This teaches data-driven optimization mirroring real ML systems engineering: profiling to identify bottlenecks, principled trade-offs between accuracy and efficiency, and reproducible benchmarking practices essential for production deployment.

3.3 Module Structure

Each module follows a consistent **Build** \rightarrow **Use** \rightarrow **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 ([Meadows, 2008](#)) 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. To maintain consistency across all 20 modules, Module 08 (DataLoader) serves as the canonical reference implementation that all modules follow, reducing maintenance burden and enabling consistent community contribution.

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 \times optimizer state memory (momentum and variance) but converges faster than SGD. When is the 4 \times total training memory trade-off worth it?”). These open-ended questions assess transfer ([Burstein et al.](#)): 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 ([Frolli et al.](#)): 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 address what we call the “implementation-example gap”—students can pass all unit tests but fail milestone tasks due to composition errors. This gap arises because unit tests validate isolated correctness (“Does `Linear.forward()` produce correct output?”), while milestones validate system composition (“Does the training loop properly orchestrate forward passes, loss computation, and backpropagation?”). Students who pass all Module 01–07 unit tests might still fail Milestone 3 (MLP Revival) if their components don’t compose correctly into functional systems. 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 on MNIST digits.
 4. **1998 CNN Revolution** (after Module 09): Image classification demonstrating convolutional architectures’ advantage (Krizhevsky and Hinton, 2009; Lecun et al.). Students import `from tinytorch.nn import Conv2d, MaxPool2d`, training both MLP and CNN on CIFAR-10 to measure architectural improvements themselves through direct comparison.
 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. **2018 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 (`tinytorch.nn, tinytorch.optim, tinytorch.profiling, tinytorch.optimization`) demonstrating quantization, compression, and acceleration for 10× faster inference and 4× smaller models while maintaining accuracy. This validates that students can apply optimization techniques systematically and measure impact through reproducible benchmarking.

Each milestone: (1) recreates actual breakthroughs using exclusively student code, (2) uses *only* TinyTorch implementations (no PyTorch/TensorFlow), (3) validates success through task-appropriate performance, and (4) demonstrates architectural comparisons showing why new approaches improved over predecessors.

Validation Approach: While milestones provide pedagogical motivation through historical framing, they simultaneously serve a technical validation purpose: demonstrating implementation correctness through real-world task performance. Success criteria for each milestone:

- **Milestone 1 (1958 Perceptron):** Solves linearly separable problems (e.g., 4-point OR/AND tasks), demonstrating basic gradient descent convergence.
- **Milestone 2 (1969 XOR Solution):** Solves XOR classification, proving multi-layer networks handle non-linear problems that single layers cannot.
- **Milestone 3 (1986 MLP Revival):** Achieves strong MNIST digit classification accuracy, validating backpropagation through all layers of deep networks.
- **Milestone 4 (1998 CNN Revolution):** Demonstrates meaningful CIFAR-10 learning (substantially better than random 10% baseline), showing convolutional feature extraction works correctly.
- **Milestone 5 (2017 Transformer):** Generates coherent multi-token text continuations on TinyTalks dataset, demonstrating functional attention mechanisms and autoregressive generation.
- **Milestone 6 (Capstone):** Student-selected challenge across Vision/Language/Speed/Compression tracks with self-defined success metrics, demonstrating production systems integration.

Performance targets differ from published state-of-the-art due to pure-Python constraints (no GPU acceleration, simplified architectures). Correctness matters more than speed: if a student’s CNN learns meaningful CIFAR-10 features, their convolution, pooling, and backpropagation implementations compose correctly into a functional vision system. This approach mirrors professional debugging where implementations prove correct by solving real tasks, not by passing synthetic unit tests alone.

4 Progressive Disclosure

This section details how TinyTorch implements progressive disclosure: a pattern that manages cognitive load by revealing Tensor capabilities gradually through monkey-patching while maintaining a unified mental model. Unlike approaches that introduce separate classes or defer features entirely, students work with a single `Tensor` class throughout the curriculum.

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). Figure 3 visualizes this activation timeline across the curriculum.

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

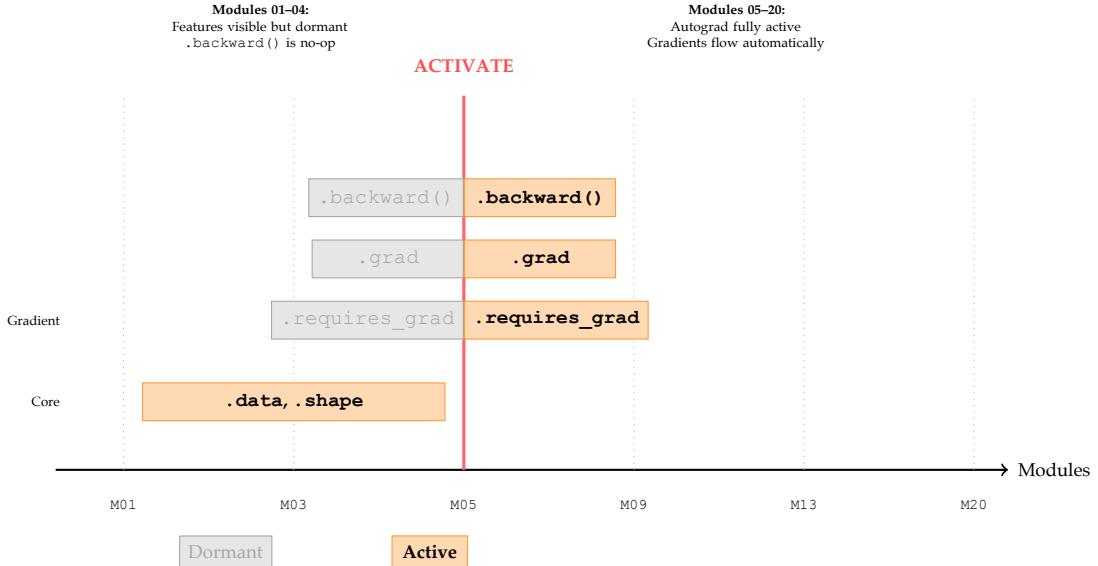


Figure 3. Progressive disclosure manages cognitive load through runtime feature activation. From Module 01, students see the complete Tensor API including gradient methods (`.backward()`, `.grad`, `.requires_grad`), but these features remain dormant (gray, dashed)—they exist as placeholders that return gracefully. In Module 05, runtime method enhancement activates full autograd functionality (orange, solid) without breaking earlier code. This creates three learning benefits: (1) students learn the complete API early, avoiding interface surprise later; (2) Module 01 code continues working unchanged when autograd activates, demonstrating forward compatibility; (3) visible but inactive features create curiosity-driven questions (“Why does `.backward()` exist if we can’t use it yet?”) that motivate curriculum progression.

```

1 # Module 01: Foundation Tensor
2 class Tensor:
3     def __init__(self, data, requires_grad=False):
4         self.data = np.array(data, dtype=np.float32)
5         self.shape = self.data.shape
6         # Gradient features - dormant
7         self.requires_grad = requires_grad
8         self.grad = None
9         self.__backward = None
10
11    def backward(self, gradient=None):
12        """No-op until Module 05"""
13        pass
14
15    def __mul__(self, other):
16        return Tensor(self.data * other.data)

```

Listing 2. Module 01: Dormant gradient features.

questions motivating curriculum progression.

4.2 Pedagogical Justification

Progressive disclosure is grounded in cognitive load theory’s principle of managing element interactivity through progressive complexity revelation (Sweller). The pattern provides two established benefits: (1) **forward compatibility**: Module 01 code continues working when autograd activates, and (2) **unified mental model**: students work with one Tensor class throughout. The cognitive load hypothesis (early API familiarity reduces future load when features activate) competes with po-

```

1 def enable_autograd():
2     """Monkey-patch Tensor with gradients"""
3     def backward(self, gradient=None):
4         if gradient is None:
5             gradient = np.ones_like(self.data)
6         if self.grad is None:
7             self.grad = gradient
8         else:
9             self.grad += gradient
10        if self.__backward is not None:
11            self.__backward(gradient)
12
13    # Monkey-patch: replace methods
14    Tensor.backward = backward
15    print("Autograd activated!")
16
17    # Module 05 usage
18    enable_autograd()
19    x = Tensor([3.0], requires_grad=True)
20    y = x * x # y = 9.0
21    y.backward()
22    print(x.grad) # [6.0] - dy/dx = 2x

```

Listing 3. Module 05: Autograd activation.

tential split-attention effects from visible but dormant features. Empirical measurement planned for Fall 2025 (Section 8) will quantify the net cognitive load impact across both mechanisms through dual-task methodology and self-report scales.

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.

Implementation Choice: Monkey-Patching vs. Inheritance. Alternative designs include inheritance (TensorV1/TensorV2) or composition. We chose monkey-patching for three pedagogical reasons: (1) **Unified mental model:** students work with one `Tensor` class throughout, not switching between versions; (2) **Historical accuracy:** models PyTorch 0.4’s Variable-Tensor merger via runtime consolidation rather than design-time hierarchy; (3) **Forward compatibility demonstration:** Module 01 code continues working unchanged when features activate, teaching interface stability principles. The software engineering trade-off (global state modification) is explicitly discussed in Module 05’s reflection questions, making students aware of both the pedagogical benefit and production code concerns.

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 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

Having established TinyTorch’s systems-first architecture (Section 3), this section details how systems awareness manifests 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 progression applies situated cognition (Lave and Wenger) by mirroring professional ML engineering workflow: measure resource requirements, understand computational costs, then optimize bottlenecks.

```

1 def conv2d_explicit(input, weight):
2     """7 nested loops - see the complexity!
3     input: (B, C_in, H, W)
4     weight: (C_out, C_in, K_h, K_w)"""
5     B, C_in, H, W = input.shape
6     C_out, K_h, K_w = weight.shape
7     H_out, W_out = H - K_h + 1, W - K_w + 1
8     output = np.zeros((B, C_out, H_out, W_out))
9
10    # Count: 1,2,3,4,5,6,7 loops
11    for b in range(B):
12        for c_out in range(C_out):
13            for h in range(H_out):
14                for w in range(W_out):
15                    for c_in in range(C_in):
16                        for kh in range(K_h):
17                            for kw in range(K_w):
18                                output[b,c_out,h,w] += \
19                                    input[b,c_in,h+kh,w+kw] * \
20                                    weight[c_out,c_in,kh,kw]
21
22    return output

```

Listing 4. Explicit convolution showing 7-nested complexity.

5.1 Phase 1: Understanding and Characterizing Memory Usage

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 ≈ 180 MB) and ImageNet ($1.2M \times 224 \times 224 \times 3 \times 4$ bytes ≈ 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 $10\text{--}100\times$ 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 $O(B \times C_{out} \times H_{out} \times W_{out} \times C_{in} \times K_h \times K_w)$ complexity visible and countable.

This explicit implementation illustrates TinyTorch’s pedagogical philosophy of minimal NumPy reliance until concepts are established. While the curriculum

Table 3. Runtime comparison: TinyTorch vs PyTorch (CPU). Pure Python implementations with explicit loops are $100\text{--}10,000\times$ slower, making performance costs visible and optimization meaningful. Benchmarks measured on Intel i7 CPU; PyTorch uses optimized BLAS libraries while TinyTorch uses pure Python for pedagogical transparency.

| Operation | TinyTorch | PyTorch | Ratio |
|----------------------|-----------|---------|----------------|
| matmul (1K×1K) | 1.0 s | 0.9 ms | $1,090\times$ |
| conv2d (CIFAR batch) | 97 s | 10 ms | $10,017\times$ |
| softmax (10K elem) | 6 ms | 0.05 ms | $134\times$ |

builds on NumPy as foundational infrastructure (array 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: $128 \times 32 \times 28 \times 28 \times 3 \times 5 \times 5 = 241\text{M}$ multiply-accumulate operations. This concrete measurement motivates Module 18’s vectorization ($10\text{--}100\times$ speedup) and explains why CNNs require hardware acceleration.

Experiencing Performance Reality. Table 3 shows TinyTorch’s deliberate slowness compared to PyTorch ($100\text{--}10,000\times$ 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 97 seconds per CIFAR batch versus PyTorch’s 10 milliseconds, they understand *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 $4\times$?”). Where foundation modules taught calculating footprints and counting FLOPs, optimiza-

tion 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 $4\times$ 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 $100\times$ 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 are designed to develop reflexive systems thinking. When encountering new ML techniques, the curriculum aims to train them to automatically ask: “How much memory? What’s the computational complexity? What are the trade-offs?” Whether this design successfully makes these questions automatic rather than afterthoughts requires empirical validation.

6 Course Deployment

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 (Section 8), though self-paced learning requires no additional materials beyond the modules themselves.

6.2 Tier-Based Curriculum Configurations

TinyTorch’s three-tier architecture (Foundation, Architecture, Optimization) enables flexible deployment matching diverse course objectives and time constraints. Instructors can deploy complete tiers or selectively focus on specific learning goals:

Configuration 1: Foundation Only (Modules 01–07). Students build core framework internals from scratch: tensors, activations, layers, losses, autograd, optimizers,

and training loops. This configuration suits introductory ML systems courses, undergraduate capstone projects, or bootcamp modules focusing on framework fundamentals. Students complete Milestones 1–3 (Perceptron, XOR, MLP Revival) demonstrating functional autograd and training infrastructure. Upon completion, students understand `loss.backward()` mechanics, can debug gradient flow, and profile memory usage. Ideal for courses prioritizing systems fundamentals over architectural breadth.

Configuration 2: Foundation + Architecture (Modules 01–13). Extends Configuration 1 with modern deep learning architectures: datasets/dataloaders, convolution, pooling, embeddings, attention, and transformers. This configuration enables comprehensive ML systems courses or graduate-level deep learning seminars. Students complete Milestones 4–5 (CNN Revolution, Transformer Era) demonstrating working vision and language models. Upon completion, students implement production architectures from scratch, understand memory scaling ($O(N^2)$ attention), and recognize architectural trade-offs (109 \times parameter efficiency from Conv2d weight sharing). Suitable for semester-long courses covering both internals and modern ML.

Configuration 3: Optimization Focus (Modules 14–19 only). Students import pre-built `tinytorch.nn` and `tinytorch.optim` packages from Configurations 1–2, implementing only production optimization techniques: profiling, quantization, compression, memoization, acceleration, and benchmarking. This configuration targets production ML courses, TinyML workshops, or edge deployment seminars where students already understand framework basics but need systems optimization depth. Students complete Milestone 6 (MLPerf-inspired benchmark) demonstrating 10 \times speedup and 4 \times compression. Upon completion, students optimize existing models for deployment constraints. Addresses key pedagogical limitation: students interested in quantization shouldn’t need to re-implement autograd first.

These configurations support “build what you’re learning, import what you need” pedagogy. Configuration 3 students focus on optimization while treating Foundation / Architecture as trusted dependencies, mirroring professional practice where engineers specialize rather than rebuilding entire stacks. The three-tier structure also enables multi-semester deployments aligned with academic terms, and hybrid integration where TinyTorch modules augment PyTorch-first courses by revealing framework internals (e.g., implementing Module 05 autograd to understand `loss.backward()`, or Module 09 convolution to demystify `torch.nn.Conv2d`).

6.3 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. Text-based ASCII connection maps enhance accessibility for visually impaired students using screen readers, while offline-first datasets (Section 3) eliminate network dependencies during training.

6.3.1 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).

6.3.2 NBGrader Autograding Workflow

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

```
1 # Cell metadata defines grading parameters:
2 # nbgrader = {
3 #     "grade": true,
4 #     "grade_id": "tensor_memory",
5 #     "points": 2,
6 #     "locked": false,
7 #     "solution": true
8 # }
9
10 def memory_footprint(self):
11     """Calculate tensor memory in bytes"""
12     ### BEGIN SOLUTION
13     return self.data.nbytes
14     ### END SOLUTION
```

Listing 5. NBGrader cell metadata and solution structure.

son. 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).

6.4 Automated Assessment Infrastructure

TinyTorch integrates NBGrader (Jupyter et al.) for scalable automated assessment (Thompson et al., 2008). 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.

NBGrader integration uses BEGIN SOLUTION/END SOLUTION markers and test cells, providing automated assessment infrastructure that scales with class size.

```

1 # After Module 01: Basic tensors
2 from tinytorch.core import Tensor
3
4 # After Module 09: CNNs available
5 from tinytorch.nn import Conv2d, MaxPool2d, Linear
6 # Autograd active - gradients flow!
7
8 # After Module 13: Complete framework
9 from tinytorch.nn import Transformer, Embedding,
  Attention

```

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

6.5 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.

TinyTorch implements a literate programming workflow where source files serve dual purposes: executable Python code and educational documentation. Export happens via nbdev (Howard and Gugger) directives (`# | export`) embedded in module source files, enabling automatic package generation via `nbdev_export`. Critically, TinyTorch modules are developed as Python source files using Jupyter percent format (`src/*/*.py`), with Jupyter notebooks generated for student distribution. The build system maintains single source of truth: developers edit `src/*/*.py`

```

1 ## Prerequisites & Progress
2 You've Built: Tensor, activations, layers, losses
3 You'll Build: Autograd system
4 You'll Enable: Training loops, optimizers
5
6 Connection Map:
7 Modules 01-04 → Autograd → Training (06-07)
8 (forward pass)   (backward)  (learning loops)

```

Listing 7. Module 05 connection map.

literate programming files containing both code and documentation, nbdev exports marked functions to `tinytorch/*` package structure (gitignored as generated artifact), and Jupytext converts source files to student-facing `.ipynb` notebooks. Thirteen modules (01, 05–09, 11–12, 15–19) currently use `# | export` directives for automatic package generation, enabling students to import from `tinytorch.core`, `tinytorch.nn`, `tinytorch.optim`, and `tinytorch.profiling` as they complete modules. This resolves the tension between version-controllable development (Python files enable proper diffs, merges, and code review) and notebook-based learning (students work in familiar Jupyter environments). Educators building similar curricula can adopt this pattern: maintain source-of-truth in version-controlled Python files while delivering interactive notebooks to students.

6.6 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.

6.7 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 `TA_GUIDE.md` (teaching assistant preparation and debugging strate-

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

gies).

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.

6.8 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 questions rather than providing direct answers. When students reach unproductive frustration, TAs can suggest optional scaffolding modules (numerical gradient checking before autograd implementation, scalar auto-

grad before tensor autograd) to build confidence through intermediate steps.

6.9 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

This section reflects on TinyTorch’s design through three lenses: pedagogical scope as deliberate design decision, flexible curriculum configurations enabling diverse institutional deployment, and honest assessment of limita-

tions requiring future work.

7.1 Pedagogical Scope as Design Decision

TinyTorch’s CPU-only, framework-internals-focused scope represents deliberate pedagogical constraint, not technical limitation. This scoping embodies three design principles:

Accessibility over performance: Pure Python eliminates GPU dependency, prioritizing equitable access over execution speed (pedagogical transparency detailed in Section 5). GPU access remains inequitably distributed: cloud credits favor well-funded institutions, personal GPUs favor affluent students. The $100\text{--}1000\times$ slowdown versus PyTorch (Table 3) is acceptable when pedagogical goal is understanding internals, not training production models.

Incremental complexity management: GPU programming introduces memory hierarchy (registers, shared memory, global memory), kernel launch semantics, race conditions, and hardware-specific tuning. Teaching GPU programming simultaneously with autograd would violate cognitive load constraints. TinyTorch enables “framework internals now, hardware optimization later” learning pathway. Students completing TinyTorch should pursue GPU acceleration through PyTorch tutorials, NVIDIA Deep Learning Institute courses, or advanced ML systems courses, building on internals understanding to comprehend optimization techniques.

Similarly, distributed training (data parallelism, model parallelism, gradient synchronization) and production deployment (model serving, compilation, MLOps) introduce substantial additional complexity orthogonal to framework understanding. These topics remain important but beyond current pedagogical scope. Future extensions could address distributed systems through simulation-based pedagogy (Section 8), maintaining accessibility while teaching concepts.

7.2 Limitations

TinyTorch’s current implementation contains gaps requiring future work.

Performance transparency tradeoff: Pure Python executes $100\text{--}1000\times$ slower than PyTorch (Table 3), a deliberate choice for pedagogical clarity. Seven explicit convolution loops reveal algorithmic complexity better than optimized C++ kernels, but slow execution limits practical experimentation. Students complete milestones (65–75% CIFAR-10 accuracy, transformer text generation) but cannot iterate rapidly on architecture search.

Energy consumption measurement: While TinyTorch covers optimization techniques with significant energy implications (quantization achieving $4\times$ compression,

pruning enabling $10\times$ model shrinkage), the curriculum does not explicitly measure or quantify energy consumption. 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 to make sustainability an explicit learning objective alongside memory and latency optimization, particularly relevant for edge deployment.

Language and accessibility: Materials exist exclusively in English. Modular structure facilitates translation; community contributions welcome. Code examples omit type annotations (e.g., `def forward(self, x: Tensor) -> Tensor:)` to reduce visual complexity for students learning ML concepts simultaneously. While this prioritizes pedagogical clarity, it means students don’t practice type-driven development increasingly standard in production ML codebases. Future iterations could introduce type hints progressively: omitting them in early modules (01–05), then adding them in optimization modules (14–18) where interface contracts become critical.

Forward dependency prevention: A recurring curriculum maintenance challenge is forward dependency creep—advanced concepts leaking into foundational modules through helper functions, error messages, or test cases that assume knowledge students haven’t yet acquired. For example, an error message in Module 03 (Layers) that mentions “computational graph” assumes Module 05 (Autograd) knowledge. Maintaining pedagogical ordering requires vigilance during curriculum updates; future work could automate this validation through CI/CD checks that flag cross-module dependencies violating prerequisite ordering.

8 Future Directions

TinyTorch’s current implementation establishes a foundation for three extension directions: empirical validation to test pedagogical hypotheses, curriculum evolution to expand systems coverage beyond CPU-only scope, and community adoption to measure educational impact through deployment at scale.

8.1 Empirical Validation Roadmap

While TinyTorch’s design is grounded in established learning theory (cognitive load (Sweller), progressive disclosure, cognitive apprenticeship (Collins et al.)), its pedagogical effectiveness requires empirical validation through controlled classroom studies. We commit to the following validation roadmap:

Phase 1: Pilot Deployment (Fall 2025, $n = 30\text{--}50$ stu-

dents). Deploy at 2–3 universities in introductory ML systems courses as primary hands-on framework alongside theory lectures. Cognitive load measurement uses Paas Mental Effort Rating Scale ([Paas](#)) administered after Modules 05 (autograd) and 09 (CNNs) to test progressive disclosure hypothesis: does dormant feature activation reduce cognitive load compared to introducing autograd as separate framework? Time-to-completion tracking instruments each module to measure actual completion time across different student backgrounds and pacing modes. Formative assessment identifies common struggle points, prerequisite gaps, and module pacing issues through instructor interviews, student feedback surveys, and learning analytics from NBGrader submissions.

Phase 2: Comparative Study (Spring 2026, $n = 100\text{--}150$ students). Randomized controlled trial compares TinyTorch (systems-first, build-from-scratch) versus PyTorch-only (application-first, use-existing frameworks) versus lecture-only (control) across 3 sections of same ML course with identical theory content. Conceptual understanding measured through ML systems concept inventory (adapted from program visualization assessment ([Sorva, 2012](#)) for systems thinking) administered pre-course and post-course, assessing autograd mechanics, memory profiling, computational complexity, and optimization tradeoffs. Transfer performance evaluated through post-course debugging task requiring PyTorch profiling and optimization on novel CNN architecture (e.g., “This training loop runs out of memory: identify bottlenecks and fix”). Does building TinyTorch improve debugging transfer to production frameworks? Code quality analysis evaluates student-written training loops for memory efficiency (batch size tuning, gradient accumulation awareness), vectorization (avoiding Python loops), and systems awareness (profiling-informed decisions versus trial-and-error).

Phase 3: Longitudinal Tracking (2026–2027, $n = 200+$ students). Re-administer concept inventory at 6 months and 12 months post-course to measure long-term retention of systems thinking: does implementation-based learning persist better than lecture-based learning? Track TinyTorch cohort versus control group performance in subsequent ML systems courses (e.g., CMU Deep Learning Systems ([Chen and Zheng, 2022](#)), distributed training courses) to measure preparation effectiveness. Survey employment placement in ML engineering roles (requiring systems knowledge) versus ML application roles (using frameworks as black boxes) at 1–2 years post-graduation. Does systems-first education influence career trajectory?

Open Science Commitment: All validation studies will be pre-registered on Open Science Framework (OSF) with hypotheses, instruments, and analysis plans pub-

lished before data collection. Datasets (anonymized student performance, survey responses, code submissions) and analysis code will be released openly under CC-BY-4.0 license. Results (positive or negative) will be published regardless of outcome, avoiding publication bias. Validation data will inform iterative curriculum refinement through evidence-based design updates, ensuring continuous improvement grounded in empirical pedagogy research rather than assumption.

8.2 Curriculum Evolution

TinyTorch’s CPU-only design prioritizes pedagogical transparency, but students benefit from understanding GPU acceleration and distributed training concepts without requiring expensive hardware. Future curriculum extensions would maintain TinyTorch’s core principle (understanding through transparent implementation) while expanding systems coverage through complementary pedagogical approaches.

Performance Analysis Through Analytical Models. Future extensions could enable students to compare TinyTorch CPU implementations against PyTorch GPU equivalents through roofline modeling ([Williams et al.](#)). Rather than writing CUDA code, students would profile existing implementations to understand memory hierarchy differences, parallelism benefits, and compute versus memory bottlenecks. The roofline approach maintains TinyTorch’s accessibility (no GPU hardware required) while preparing students for GPU programming by teaching first-principles performance analysis.

Distributed Training Through Simulation. Understanding distributed training communication patterns requires simulation-based pedagogy rather than multi-GPU clusters. Future extensions could integrate distributed training simulation enabling single-machine exploration of multi-device concepts: gradient synchronization overhead, scalability analysis across worker counts, network topology impact on communication patterns, and pipeline parallelism trade-offs. This simulation-based approach maintains pedagogical transparency: students understand distributed systems through measurement and analysis, not black-box hardware access.

Energy and Power Profiling. Edge deployment and sustainable ML ([Strubell et al.; Patterson et al.](#)) 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. This connects existing optimization techniques (quantization, pruning) taught in Modules 15–18 to concrete sustainability metrics, particularly relevant for edge AI ([Banbury et al.](#)) where battery life constrains deployment.

Hardware Simulation Integration. TinyTorch’s current profiling infrastructure—memory tracking (tracemalloc), FLOP counting, and performance benchmarking—provides algorithmic-level performance analysis. A natural extension would integrate architecture simulators (e.g., scale-sim (Samajdar et al., 2018), timeloop (Parashar et al., 2019), astra-sim (Kannan et al., 2022)) to connect high-level ML operations with cycle-accurate hardware models. This layered approach mirrors real ML systems engineering: students first understand algorithmic complexity and memory patterns in TinyTorch, then trace those operations down to microarchitectural performance in simulators. Such integration would complete the educational arc from algorithmic implementation → systems profiling → hardware realization, enabling first-principle analysis of how model architecture, system configuration (memory hierarchy, compute units, interconnects), and hardware substrate jointly determine production performance. Early discussions with students and collaborators suggest strong pedagogical value in this systems-to-hardware pipeline, maintaining TinyTorch’s accessibility (no GPU hardware required) while preparing students for hardware-aware optimization through measurement-driven analysis rather than black-box experimentation.

Architecture Extensions. Potential additions (graph neural networks, diffusion models, reinforcement learning) must justify inclusion through systems pedagogy rather than completeness. The question is not “Can TinyTorch implement this?” but rather “Does implementing this teach fundamental systems concepts unavailable through existing modules?” Graph convolutions might teach sparse tensor operations; diffusion models might illuminate iterative refinement trade-offs. However, extensions succeed only when maintaining TinyTorch’s principle: **every line of code teaches a systems concept**. Community forks demonstrate this philosophy: quantum ML variants replace tensors with quantum state vectors (teaching circuit depth versus memory); robotics forks emphasize RL simulation overhead and real-time constraints. The curriculum remains intentionally incomplete as a production framework: completeness lies in foundational systems thinking applicable across all ML architectures.

8.3 Community and Sustainability

As part of the ML Systems Book ecosystem (mlsysbook.ai), TinyTorch benefits from and contributes to broader educational infrastructure. Integration with the textbook’s theoretical foundations (Reddi, 2025) enables a complete pedagogical pathway: students study production ML systems architecture (distributed training patterns, quantization

strategies, deployment considerations), then implement those abstractions in TinyTorch (autograd for backpropagation, INT8 quantization, profiling infrastructure). The open-source model (MIT license) and community-driven development enable collaborative refinement across institutions: instructor discussion forums for pedagogical exchange, shared teaching resources, and empirical validation of learning outcomes.

Module 20 (Capstone) culminates the curriculum with competitive systems engineering challenges. Inspired by MLPerf benchmarking (Reddi et al., a), students optimize their implementations across accuracy, speed, compression, and efficiency dimensions, comparing results globally through standardized benchmarking infrastructure. This competitive element reinforces systems thinking: optimization requires measurement-driven decisions (profiling bottlenecks), principled trade-offs (accuracy versus compression), and reproducible methodology (standardized metrics collection). The focus remains pedagogical—understanding *why* optimizations work—rather than achieving state-of-the-art performance, but the competitive framing increases engagement and mirrors real ML engineering workflows.

9 Conclusion

Machine learning education faces a fundamental choice: teach students to *use* frameworks as black boxes, or teach them to *understand* what happens inside `loss.backward()`, why Adam requires $2\times$ optimizer state memory, why attention scales $O(N^2)$. TinyTorch demonstrates that systems understanding (building autograd, profiling memory, debugging gradient flow) is accessible without requiring GPU clusters or distributed infrastructure. This accessibility matters: students worldwide can develop framework internals knowledge on modest hardware, transforming production debugging from trial-and-error into systematic engineering.

Three pedagogical contributions enable this transformation. **Progressive disclosure** manages complexity through gradual feature activation: students work with unified Tensor implementations that gain capabilities across modules rather than replacing code mid-semester. **Systems-first integration** embeds memory profiling from Module 01, preventing “algorithms without costs” learning where students optimize accuracy while ignoring deployment constraints. **Historical milestone validation** proves correctness through recreating nearly 70 years of ML breakthroughs (1958–2025, from Perceptron through Transformers), making abstract implementations concrete through reproducing published results.

For ML practitioners: Building TinyTorch’s 20 modules transforms how you debug production failures. When PyTorch training crashes with OOM errors, you understand memory allocation across parameters, optimizer states, and activation tensors. When gradient explosions occur, you recognize backpropagation numerical instability from implementing it yourself. When choosing between Adam and SGD under memory constraints, you know the $4\times$ total memory multiplier from building both optimizers. This systems knowledge transfers directly to production framework usage: you become an engineer who understands *why* frameworks behave as they do, not just *what* they do.

For CS education researchers: TinyTorch provides replicable infrastructure for testing pedagogical hypotheses about ML systems education. Does progressive disclosure reduce cognitive load compared to teaching autograd as separate framework? Does systems-first integration improve production readiness versus algorithms-only instruction? Do historical milestones increase engagement and retention? The curriculum embodies design patterns amenable to controlled empirical investigation. Open-source release with detailed validation roadmap enables multi-institutional studies to establish evidence-based best practices for teaching framework internals.

For educators and bootcamp instructors: TinyTorch supports flexible integration: self-paced learning requiring zero infrastructure (students run locally on laptops), institutional courses with automated NBGrader assessment, or industry team onboarding for ML engineers transitioning from application development to systems work. The modular structure enables selective adoption: foundation tier only (Modules 01–07, teaching core concepts), architecture focus (adding CNNs and Transformers through Module 13), or complete systems coverage (all 20 modules including optimization and deployment). No GPU access required, no cloud credits needed, no infrastructure barriers.

The complete codebase, curriculum materials, and assessment infrastructure are openly available at mlsysbook.ai/tinytorch (or tinytorch.ai) under permissive open-source licensing. We invite the global ML education community to adopt TinyTorch in courses, contribute curriculum improvements, translate materials for international accessibility, fork for domain-specific variants (quantum ML, robotics, edge AI), and empirically evaluate whether implementation-based pedagogy achieves its promise. The difference between engineers who know *what* ML systems do and engineers who understand *why* they work begins with understanding what’s inside `loss.backward()`, and TinyTorch makes that

understanding accessible to everyone.

Acknowledgments

Colby Banbury.

References

- Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 2nd edition, 1996.
- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, 2nd edition, 2006.
- Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, UK, 2nd edition, 2002.
- Colby R. Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, David Patterson, Danilo Pau, Jae-sun Seo, Jeff Sieracki, Urmish Thakker, Marian Verhelst, and Poonam Yadav. Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*. URL <http://arxiv.org/abs/2003.04821v4>.
- Roy Burstein, Nathaniel J. Henry, Michael L. Collison, Laurie B. Marczak, Amber Sligar, Stefanie Watson, Neal Marquez, Mahdieh Abbasalizad-Farhangi, Masoumeh Abbasi, Foad Abd-Allah, Amir Abdoli, Mohammad Abdollahi, Ibrahim Abdollahpour, Rizwan Suliankatchi Abdulkader, Michael R. M. Abrigo, Dilaram Acharya, Oladimeji M. Adebayo, Victor Adekanmbi, Davoud Adham, Mahdi Afshari, Mohammad Aghaali, Keivan Ahmadi, Mehdi Ahmadi, Ehsan Ahmadpour, Rushdia Ahmed, Chalachew Genet Akal, Joshua O. Akinyemi, Fares Alahdab, Noore Alam, Genet Melak Alamene, Kefyalew Addis Alene, Mehran Alijanzadeh, Cyrus Alinia, Vahid Alipour, Syed Mohamed Aljunid, Mohammed J. Almalki, Hesham M. Al-Mekhlafi, Khalid Altirkawi, Nelson Alvis-Guzman, Adeladza Kofi Amegah, Saeed Amini, Arianna Maever Loreche Amit, Zohreh Anbari, Sofia Androudi, Mina Anjomshoa, Fereshteh Ansari, Carl Abelardo T. Antonio, Jalal Arabloo, Zohreh Arefi, Olatunde Aremu, Bahram Armoon, Amit Arora, Al Artaman, Anvar Asadi, Mehran Asadi-Aliabadi, Amir Ashraf-Ganjouei, Reza Assadi, Bahar Ataeinia, Sachin R. Atre, Beatriz Paulina Ayala Quintanilla, Martin Amogre Ayanore, Samad Azari, Ebrahim Babaee, Arefeh Babazadeh,

Alaa Badawi, Soghra Bagheri, Mojtaba Bagherzadeh, Nafiseh Baheiraei, Abbas Balouchi, Aleksandra Barac, Quique Bassat, Bernhard T. Baune, Mohsen Bayati, Neeraj Bedi, Ettore Beghi, Masoud Behzadifar, Meysam Behzadifar, Yared Belete Belay, Brent Bell, Michelle L. Bell, Dessalegn Ajema Berbada, Robert S. Bernstein, Natalia V. Bhattacharjee, Suraj Bhattarai, Zulfiqar A. Bhutta, Ali Bijani, Somayeh Bohlouli, Nicholas J. K. Breitborde, Gabrielle Britton, Annie J. Browne, Sharath Burugina Nagaraja, Reinhard Busse, Zahid A. Butt, Josip Car, Rosario Cárdenas, Carlos A. Castañeda-Orjuela, Ester Cerin, Wagaye Fentahun Chanie, Pranab Chatterjee, Dinh-Toi Chu, Cyrus Cooper, Vera M. Costa, Koustuv Dalal, Lalit Dandona, Rakhi Dandona, Farah Daoud, Ahmad Daryani, Rajat Das Gupta, Ian Davis, Nicole Davis Weaver, Dragos Virgil Davitoiu, Jan-Walter De Neve, Feleke Mekonnen Demeke, Gebre Teklemariam Demoz, Kebede Deribe, Rupak Desai, Aniruddha Deshpande, Hanna Demelash Desyibelew, Sagnik Dey, Samath Dhamminda Dharmaratne, Meghnath Dhimal, Daniel Diaz, Leila Doshmangir, Andre R. Duraes, Laura Dwyer-Lindgren, Lucas Earl, Roya Ebrahimi, Soheil Ebrahimpour, Andem Effiong, Aziz Eftekhari, Elham Ehsani-Chimeh, Iman El Sayed, Maysaa El Sayed Zaki, Maha El Tantawi, Ziad El-Khatib, Mohammad Hassan Emamian, Shymaa Enany, Sharareh Eskandarieh, Oghenowede Eyawo, Maha Ezalarab, Mahbobeh Faramarzi, Mohammad Fareed, Roghiyeh Faridnia, Andre Faro, Ali Akbar Fazaeli, Mehdi Fazlzadeh, Netsanet Fentahun, Seyed-Mohammad Fereshtehnejad, João C. Fernandes, Irina Filip, Florian Fischer, Nataliya A. Foigt, Masoud Foroutan, Joel Msafiri Francis, Takeshi Fukumoto, Nancy Fullman, Silvano Gallus, Destalle Gebremedhin Gebre, Tsegaye Tewelde Gebrehiwot, Gebreamlak Gebremedhn Gebremeskel, Bradford D. Gessner, Birhanu Geta, Peter W. Gething, Reza Ghadimi, Keyghobad Ghadiri, Mahsa Ghajarzadeh, Ahmad Ghashghaee, Paramjit Singh Gill, Tiffany K. Gill, Nick Golding, Nelson G. M. Gomes, Philimon N. Gona, Sameer Vali Gopalani, Giuseppe Gorini, Bárbara Niegia Garcia Goulart, Nicholas Graetz, Felix Greaves, Manfred S. Green, Yuming Guo, Arvin Haj-Mirzaian, Arya Haj-Mirzaian, Brian James Hall, Samer Hamidi, Hamidreza Haririan, Josep Maria Haro, Milad Hasankhani, Edris Hasanpoor, Amir Hasanzadeh, Hadi Hassankhani, Hamid Yimam Hassen, Mohamed I. Hegazy, Delia Hendrie, Fatemeh Heydarpour, Thomas R. Hird, Chi Linh Hoang, Gillian Hollerich, Enayatollah Homaie Rad, Mojtaba Hoseini-Ghahfarokhi, Naznin Hossain, Mostafa Hosseini, Mehdi Hosseinzadeh, Mihaela Hostiuc,

Sorin Hostiuc, Mowafa Househ, Mohamed Hsairi, Olayinka Stephen Ilesanmi, Mohammad Hasan Imani-Nasab, Usman Iqbal, Seyed Sina Naghibi Irvani, Nazrul Islam, Sheikh Mohammed Shariful Islam, Mikk Jürisson, Nader Jafari Balalami, Amir Jalali, Javad Javidnia, Achala Upendra Jayatileke, Ensiyeh Jenabi, John S. Ji, Yash B. Jobanputra, Kimberly Johnson, Jost B. Jonas, Zahra Jorjoran Shushtari, Jacek Jerzy Jozwiak, Ali Kabir, Amaha Kahsay, Hamed Kalani, Rohollah Kalhor, Manoochehr Karami, Surendra Karki, Amir Kasaean, Nicholas J. Kassebaum, Peter Njenga Keiyoro, Grant Rodgers Kemp, Roghayeh Khabiri, Yousef Saleh Khader, Morteza Abdullatif Khafaie, Ejaz Ahmad Khan, Junaid Khan, Muhammad Shahzeb Khan, Young-Ho Khang, Khaled Khatab, Amir Khater, Mona M. Khater, Alireza Khatony, Mohammad Khazaei, Salman Khazaei, Maryam Khazaei-Pool, Jagdish Khubchandani, Neda Kianipour, Yun Jin Kim, Ruth W. Kimokoti, Damaris K. Kinyoki, Adnan Kisa, Sezer Kisa, Tufa Kolola, Soewarta Kosen, Parvaiz A. Koul, Ai Koyanagi, Moritz U. G. Kraemer, Kewal Krishan, Kris J. Krohn, Nuworza Kugbey, G. Anil Kumar, Manasi Kumar, Pushpendra Kumar, Desmond Kuupiel, Ben Lacey, Sheetal D. Lad, Faris Hasan Lami, Anders O. Larsson, Paul H. Lee, Mostafa Leili, Aubrey J. Levine, Shanshan Li, Lee-Ling Lim, Stefan Listl, Joshua Longbottom, Jaifred Christian F. Lopez, Stefan Lorkowski, Sameh Magdeldin, Hassan Magdy Abd El Razek, Muhammed Magdy Abd El Razek, Azeem Majeed, Afshin Maleki, Reza Malekzadeh, Deborah Carvalho Malta, Abdullah A. Mamun, Navid Manafi, Ana-Laura Manda, Morteza Mansourian, Francisco Rogerlândio Martins-Melo, Anthony Masaka, Benjamin Ballard Massenburg, Pallab K. Maulik, Benjamin K. Mayala, Mohsen Mazidi, Martin McKee, Ravi Mehrotra, Kala M. Mehta, Gebrekirios Gebremichael Meles, Walter Mendoza, Ritesh G. Menezes, Atte Meretoja, Tuomo J. Meretoja, Tomislav Mestrovic, Ted R. Miller, Molly K. Miller-Petrie, Edward J. Mills, George J. Milne, G. K. Mini, Seyed Mostafa Mir, Hamed Mirjalali, Erkin M. Mirrakhimov, Efat Mohamadi, Dara K. Mohammad, Aso Mohammad Darwesh, Naser Mohammad Gholi Mezerji, Ammas Siraj Mohammed, Shafiu Mohammed, Ali H. Mokdad, Mariam Molokhia, Lorenzo Monasta, Yoshan Moodley, Mahmood Moosazadeh, Ghobad Moradi, Masoud Moradi, Yousef Moradi, Maziar Moradi-Lakeh, Mehdi Moradinazar, Paula Moraga, Lidia Morawska, Abbas Mosapour, Seyyed Meysam Mousavi, Ulrich Otto Mueller, Atalay Goshu Muluneh, Ghulam Mustafa, Behnam Nabavizadeh, Mehdi Naderi, Ahamar-

shan Jayaraman Nagarajan, Azin Nahvijou, Farid Najafi, Vinay Nangia, Duduzile Edith Ndwandwe, Nahid Neamati, Ionut Negoi, Ruxandra Irina Negoi, Josephine W. Ngunjiri, Huong Lan Thi Nguyen, Long Hoang Nguyen, Son Hoang Nguyen, Katie R. Nielsen, Dina Nur Anggraini Ningrum, Yirga Legesse Nirayo, Molly R. Nixon, Chukwudi A. Nnaji, Marzieh Nojomi, Mehdi Noroozi, Shirin Nosratnejad, Jean Jacques Noubiap, Soraya Nouraei Motlagh, Richard Ofori-Asenso, Felix Akpojene Ogbo, Kelechi E. Oladimeji, Andrew T. Olagunju, Meysam Olfatifar, Solomon Olum, Bolajoko Olubukunola Olusanya, Mojisola Morenike Oluwasanu, Obinna E. Onwujekwe, Eyal Oren, Doris D. V. Ortega-Altamirano, Alberto Ortiz, Osayomwanbo Osarenotor, Frank B. Osei, Aaron E. Osgood-Zimmerman, Stanislav S. Otstavnov, Mayowa Ojo Owolabi, Mahesh P. A., Abdol Sattar Pagheh, Smita Pakhale, Songhomitra Panda-Jonas, Animika Pandey, Eun-Kee Park, Hadi Parsian, Tahereh Pashaei, Sangram Kishor Patel, Veincent Christian Filipino Pepito, Alexandre Pereira, Samantha Perkins, Brandon V. Pickering, Thomas Pilgrim, Majid Pirestani, Bakhtiar Piroozi, Meghdad Pirsahab, Oleguer Plana-Ripoll, Hadi Pourjafar, Parul Puri, Mostafa Qorbani, Hedley Quintana, Mohammad Rabiee, Navid Rabiee, Amir Radfar, Alireza Rafiei, Fakher Rahim, Zohreh Rahimi, Vafa Rahimi-Movaghah, Shadi Rahimzadeh, Fatemeh Rajati, Sree Bhushan Raju, Azra Ramezankhani, Chhabi Lal Ranabhat, Davide Rasella, Vahid Rashedi, Lal Rawal, Robert C. Reiner Jr, Andre M. N. Renzaho, Satar Rezaei, Aziz Rezapour, Seyed Mohammad Riahi, Ana Isabel Ribeiro, Leonardo Roever, Elias Merdassa Roro, Max Roser, Gholamreza Roshandel, Daem Roshani, Ali Rostami, Enrico Rubagotti, Salvatore Rubino, Siamak Sabour, Nafis Sadat, Ehsan Sadeghi, Reza Saeedi, Yahya Safari, Roya Safari-Faramani, Mahdi Safdarian, Amirhossein Sahebkar, Mohammad Reza Salahshoor, Nasir Salam, Payman Salamat, Farkhonde Salehi, Saleh Salehi Zahabi, Yahya Salimi, Hamideh Salimzadeh, Joshua A. Salomon, Evanson Zondani Sambala, Abdallah M. Samy, Milena M. Santric Milicevic, Bruno Piassi Sao Jose, Sivan Yegnanarayana Iyer Saraswathy, Rodrigo Sarmiento-Suárez, Benn Sartorius, Brijesh Sathian, Sonia Saxena, Alyssa N. Sbarra, Lauren E. Schaeffer, David C. Schwebel, Sadaf G. Sepanlou, Seyedmojtoba Seyedmousavi, Faramarz Shaahmadi, Masood Ali Shaikh, Mehran Shams-Beyranvand, Amir Shamshirian, Morteza Shamsizadeh, Kiomars Sharafi, Mehdi Sharif, Mahdi Sharif-Alhoseini, Hamid Sharifi, Jayendra Sharma, Rajesh Sharma, Aziz Sheikh, Chloe Shields, Mika Shigematsu, Rahman Shiri, Ivy

Shiue, Kerem Shuval, Tariq J. Siddiqi, João Pedro Silva, Jasvinder A. Singh, Dhirendra Narain Sinha, Malede Mequanent Sisay, Solomon Sisay, Karen Sliwa, David L. Smith, Ranjani Somayaji, Moslem Soofi, Joan B. Soriano, Chandrashekhar T. Sreeramareddy, Agus Sudaryanto, Mu’awiyyah Babale Sufiyan, Bryan L. Sykes, P. N. Sylaja, Rafael Tabarés-Seisdedos, Karen M. Tabb, Takahiro Tabuchi, Nuno Taveira, Mohamad-Hani Temsah, Abdullah Sulieman Terkawi, Zemenu Tadesse Tessema, Kavumpurathu Raman Thankappan, Sathish Thirunavukkarasu, Quyen G. To, Marcos Roberto Tovani-Palone, Bach Xuan Tran, Khanh Bao Tran, Irfan Ullah, Muhammad Shariq Usman, Olalekan A. Uthman, Amir Vahedian-Azimi, Pascual R. Valdez, Job F. M. van Boven, Tommi Juhani Vasankari, Yasser Vasseghian, Yousef Veisani, Narayanaswamy Venketasubramanian, Francesco S. Violante, Sergey Konstantinovich Vladimirov, Vasily Vlassov, Theo Vos, Giang Thu Vu, Isidora S. Vujcic, Yasir Waheed, Jon Wakefield, Haidong Wang, Yafeng Wang, Yuan-Pang Wang, Joseph L. Ward, Robert G. Weintraub, Kidu Gidey Weldegwergs, Girmay Teklay Weldesamuel, Ronny Westerman, Charles Shey Wiysonge, Dawit Zewdu Wondafrash, Lauren Woycynski, Ai-Min Wu, Gelin Xu, Abbas Yadegar, Tomohide Yamada, Vahid Yazdi-Feyzabadi, Christopher Sabo Yilgwan, Paul Yip, Naohiro Yonemoto, Javad Yoosefi Lebni, Mustafa Z. Younis, Mahmoud Yousefifard, Hebat-Allah Salah A. Yousof, Chuanhua Yu, Hasan Yusefzadeh, Erfan Zabeh, Telma Zahirian Moghadam, Sojib Bin Zaman, Mohammad Zamani, Hamed Zandian, Alireza Zangeneh, Taddese Alemu Zerfu, Yunquan Zhang, Arash Ziapour, Sanjay Zodpey, Christopher J. L. Murray, and Simon I. Hay. Mapping 123 million neonatal, infant and child deaths between 2000 and 2017. *Nature*, 574(7778):353–358. ISSN 0028-0836,1476-4687. doi: 10.1038/s41586-019-1545-0. URL <https://doi.org/10.1038/s41586-019-1545-0>.

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

Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. The nachos instructional operating system. In *Proceedings of the USENIX Winter 1993 Conference*, pages 481–488, San Diego, CA, 1993. USENIX Association.

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>.
- Alessandro Frolli, Francesco Cerciello, Sonia Ciotola, Maria Carla Ricci, Clara Esposito, and Luigia Simona Sica. Narrative approach and mentalization. *Behavioral Sciences*, 13(12):994. ISSN 2076-328X. doi: 10.3390/bs13120994. URL <https://doi.org/10.3390/bs13120994>.
- Margaret Heffernan. Earning half the minimum wage: new report reveals pressures on artists to sustain creative life. URL <https://doi.org/10.64628/ab.waecqna7u>.
- 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>.
- Saeed Rashidi Kannan, Saeed Rashidi, Srinivas Sheng, Changhai Asghari, Tuowen Zhao, Siva Rajamanickam, Tushar Kumar, Kartik Melesse, Zhangxi Jia, et al. Astra-sim: Enabling sw/hw co-design exploration for distributed dl training platforms. *arXiv preprint arXiv:2006.14479*, 2022.
- Manu Kapur. Productive failure. *Cognition and Instruction*, 26(3):379–424. ISSN 0737-0008,1532-690X. doi: 10.1080/0737000802212669. URL <https://doi.org/10.1080/0737000802212669>.
- Andrej Karpathy. micrograd: A tiny scalar-valued autograd engine and neural net library, 2022. URL <https://github.com/karpathy/micrograd>.
- 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.
- Fred G. W. C. Paas. Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive-load approach. *Journal of Educational Psychology*, 84(4):429–434. ISSN 1939-2176,0022-0663. doi: 10.1037/0022-0663.84.4.429. URL <https://doi.org/10.1037/0022-0663.84.4.429>.
- Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315. IEEE, 2019.
- David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*. URL <http://arxiv.org/abs/2104.10350v3>.
- Ben Pfaff, Anthony Romano, and Godmar Back. The pintos instructional operating system kernel. Technical report, Stanford University, 2009. Used in Stanford CS140 and adopted at 50+ universities worldwide.
- 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-issss60120.2024.00015. URL <https://doi.org/10.1109/codes-issss60120.2024.00015>.
- Vijay Janapa Reddi. *Machine Learning Systems: Design and Implementation*. MIT Press, 2025. URL <https://mlsysbook.ai>. Forthcoming. Early access at <https://mlsysbook.ai>.
- Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejasve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 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>.
- F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408. ISSN 1939-1471,0033-295X. doi: 10.1037/h0042519. URL <https://doi.org/10.1037/h0042519>.
- 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, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic cnn accelerator simulator. In *arXiv preprint arXiv:1811.02883*, 2018.
- Keller Executive Search. Ai & machine-learning talent gap 2025, 2025. URL <https://www.kellerexecutivesearch.com/intelligence/ai-machine-learning-talent-gap-2025/>.
- Juha Sorva. *Visual program simulation in introductory programming education ; Visuaalinen ohjelmasimulaatio ohjelmoinnin alkeisopetuksessa*. Espoo, Finland, 2012. URL <https://aaltodoc.aalto.fi/handle/123456789/3534>.
- Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. pages 3645–3650. URL <http://arxiv.org/abs/1906.02243v1>.
- Rich Sutton. The bitter lesson, 2019. URL <http://www.incompleteideas.net/IncIdeas/BitterLesson.html>. Accessed: 2024-01-15.
- 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.
- Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Englewood Cliffs, NJ, 1987. Introduced MINIX, a teaching operating system.
- 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.
- Errol Thompson, Andrew Luxton-Reilly, Jacqueline L. Whalley, Minjie Hu, and Phil Robbins. Bloom’s taxonomy for cs assessment. In *Proceedings of the Tenth Conference on Australasian Computing Education*, pages 155–161, 2008.
- 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>.

L. S. VYGOTSKY. *Mind in Society*. Harvard University Press. ISBN 9780674076686,9780674576285. doi: 10.2307/j.ctvjf9vz4. URL <https://doi.org/10.2307/j.ctvjf9vz4>.

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, and Alexander J. Smola. Dive into deep learning. *CoRR*, abs/2106.11342. URL <http://arxiv.org/abs/2106.11342v5>.