Tiny🔥Torch

Build Your Own Machine Learning Framework
From Tensors to Systems

Vijay Janapa Reddi
Harvard University

tinytorch.ai

Last updated: December 14, 2025

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 06); **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 approaches (Sutton, 2019). Deep learning surpassed handcrafted 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., 2017), 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 (Robert Half, 2024; 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 convolution and *measure* the difference directly through profiling code they wrote.

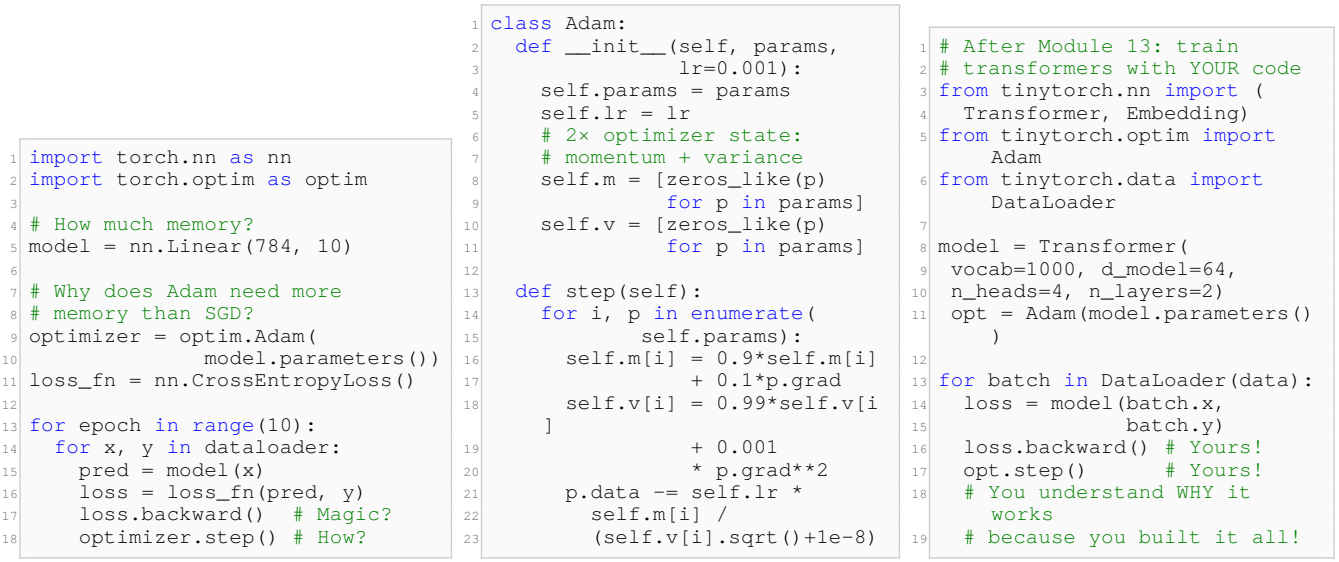
This pedagogical approach mirrors computer engineering education, where processor design progresses from transistors to logic gates to arithmetic units to complete CPUs—each layer a working system that enables the next. TinyTorch applies the same principle: tensors enable operations, operations enable layers, layers enable networks, networks enable complete ML systems. 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 dataloader (05) and autograd (06), powering optimizers (07) and training (08). Each completed module becomes immediately usable: after Module 03, students build neural networks; after Module 06, 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 addressing the systems imperative the Bitter Lesson reveals:

1. **Build-to-Validate Curriculum** (Section 3): A 20-



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

# How much memory?
model = nn.Linear(784, 10)

# Why does Adam need more
# memory than SGD?
optimizer = optim.Adam(
              model.parameters())
loss_fn = nn.CrossEntropyLoss()

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

(a) PyTorch: Black box usage

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

  def step(self):
    for i, p in enumerate(
            self.params):
      self.m[i] = 0.9*self.m[i]
                  + 0.1*p.grad
      self.v[i] = 0.99*self.v[i
    ]
                  + 0.001
                  * p.grad**2
      p.data -= self.lr *
        self.m[i] /
        (self.v[i].sqrt()+1e-8)
```

(b) TinyTorch: Build internals

```python
# After Module 13: train
# transformers with YOUR code
from tinytorch.nn import (
  Transformer, Embedding)
from tinytorch.optim import
    Adam
from tinytorch.data import
    DataLoader

model = Transformer(
 vocab=1000, d_model=64,
 n_heads=4, n_layers=2)
 opt = Adam(model.parameters()
    )

for batch in DataLoader(data):
  loss = model(batch.x,
               batch.y)
  loss.backward() # Yours!
  opt.step()      # Yours!
  # You understand WHY it
    works
  # because you built it all!
```

(c) TinyTorch: The culmination

Figure 1. From User to Engineer. (a) PyTorch's high-level APIs hide framework internals. (b) TinyTorch students implement components like Adam, learning memory costs and update rules firsthand. (c) By Module 13, every import is student-built code—transformers train on infrastructure they fully understand.

module learning path where students validate their implementations by recreating historical ML milestones—from Rosenblatt's Perceptron (1958) to modern transformers—using exclusively their own code. This provides concrete correctness criteria and grounds abstract concepts in tangible achievements.

2. **Progressive Disclosure of Complexity** (Section 4): A scaffolding technique that reveals `Tensor` internals gradually while maintaining a unified mental model. Gradient tracking infrastructure exists from Module 01 but activates only in Module 06, preventing premature cognitive overload while enabling seamless backpropagation later.

3. **Systems from Day One** (Section 5): Memory profiling, computational complexity, and performance analysis are embedded starting in Module 01—not deferred to advanced topics. Students discover that Adam requires 2× optimizer state memory by implementing it, not by reading documentation.

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 curriculum architecture (Section 3), the progressive disclosure pattern enabling cognitive load management (Section 4), and systems-first integration throughout modules (Section 5). Finally, we discuss deployment, limitations, empirical validation plans, and implications for ML education (Sections 6, 7 and 9).

2 Related Work

TinyTorch builds upon decades of work in CS education research and recent innovations in ML framework pedagogy. We first establish the pedagogical tradition that grounds our approach, then position TinyTorch relative to existing ML educational frameworks.

2.1 Pedagogical Precedents in Systems Edu

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 Unix; TinyTorch is MINIX.

Compiler education follows identical logic. Despite

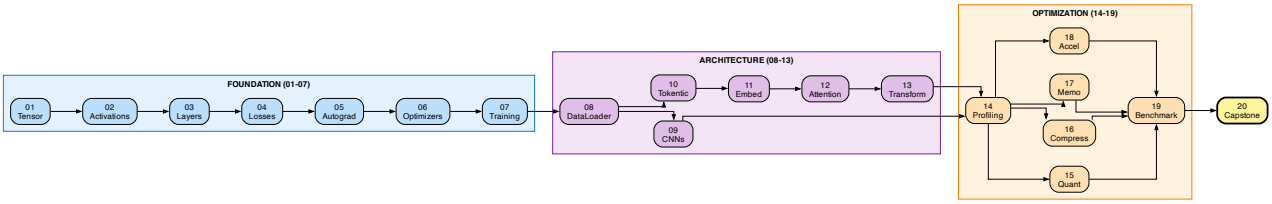


Figure 2. Module Flow. Foundation (blue): core tensor operations through training loop. Architecture (purple): branches into vision (CNNs) and language (Tokenization→Transformers) paths. Optimization (orange): profiling feeds parallel techniques—model-level (quantization, compression) and runtime (acceleration, memoization)—all converging at benchmarking before the capstone.

LLVM and GCC representing decades of engineering excellence, compiler courses continue teaching from-scratch implementation (Aho et al., 2006; Appel and Palsberg). Students build lexical analyzers, construct abstract syntax trees, implement type checkers, and generate code. The Tiger compiler (Appel and Palsberg) 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 Pintos (Pfaff et al.) 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 and Sussman, 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.

2.2 Educational ML Frameworks

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

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

tinygrad (Hotz, 2023) positions itself between micrograd's simplicity and PyTorch's production capabilities, providing a complete framework (tensor library, IR, compiler, JIT) that emphasizes hackability and transparency. Unlike opaque production frameworks, tinygrad makes "the entire compiler and IR visible," enabling students to understand deep learning compilation internals. While pedagogically valuable through its inspectable design,

tinygrad assumes significant background: students must navigate compiler concepts, multiple hardware backends, and production-level architecture without scaffolded progression or automated assessment infrastructure.

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

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

2.3 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.

Empirical validation of learning outcomes remains future work (Section 7), but design grounding in established learning theory provides theoretical justification for pedagogical choices.

3 TinyTorch Architecture

This section presents TinyTorch's curriculum architecture. The design is grounded in established learning theory—constructionism (Papert, 1980) (learning through building artifacts), cognitive apprenticeship (Collins et al., 1989) (making expert thinking visible), productive failure (Kapur) (struggling before instruction), and threshold concepts (Meyer and Land, 2003) (transformative ideas requiring mastery). These principles shaped the competency framework and mod-

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

ule progression that follow.

3.1 The ML Systems Competency Matrix

The learning theories above informed a concrete competency framework that operationalizes pedagogical goals into measurable outcomes. What must a student master to understand ML systems fundamentals? We decompose this into 8 knowledge areas progressing from foundational (tensors) to production (deployment), crossed with 5 capability levels progressing from conceptual understanding to optimization mastery (Table 2). This matrix preceded module design: we first identified the 40 competency cells, then engineered modules to systematically address them.

The matrix embodies two design principles. First, **rows are progressive**: each knowledge area builds on those above. Students cannot understand optimization without first understanding gradients; deployment concerns assume architecture mastery. This ordering directly informs TinyTorch's module sequencing. Second, **columns are progressive**: capability levels build left-to-right. Students must understand concepts (Knows) before quantifying costs (Measures), must measure before implementing (Implements), must implement before diagnosing failures (Debugs), and must debug before systematically improving (Optimizes). This progression shapes each module's internal structure through the Build → Use → Reflect cycle (Section 3.3).

Not every cell requires equal depth. An educational framework prioritizes foundational cells (Tensors through Training, columns 1–4) over production-specific concerns (Deployment-Optimizes). Section 6.3 maps our actual coverage, including intentional gaps where production infrastructure exceeds accessibility-first scope.

3.2 The 3-Tier Learning Journey + Capstone

TinyTorch organizes modules into three progressive tiers plus a capstone competition (Table 3). 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 01–08). Students build the mathematical core enabling neural networks to learn, following a deliberate *Forward Pass → Learning Infrastructure → Training* progression. Modules 01–04 construct forward pass components in the order data flows: tensors (data structure), activations (non-linearity), layers (parameterized transformations), and losses (objective functions). Systems thinking begins immediately: Module 01 introduces `memory_footprint()` before matrix

Table 2. Competency Matrix. The 40 cells (8 knowledge areas × 5 capability levels) characterize ML systems fundamentals. Rows progress from foundational concepts (tensors, operations) through learning mechanics (graphs, optimization) to production concerns (efficiency, deployment). Columns progress from conceptual understanding (Knows) through quantitative reasoning (Measures) and implementation (Implements) to diagnostic skill (Debugs) and performance engineering (Optimizes). This matrix guided TinyTorch's curriculum design. Section 6.3 maps the 20 modules to this framework.

Knowledge Area	Knows	Measures	Implements	Debugs	Optimizes
Tensors & Memory	Layouts, strides, dtypes	Bytes, peak allocation	Tensor class, broadcasting	OOM, memory leaks	In-place ops, pooling
Operations & Compute	Broadcast rules, semantics	FLOPs, arithmetic intensity	MatMul, Conv2d, activations	Shape mismatch, instability	Vectorization, fusion
Graphs & Differentiation	Computation graphs, memory lifecycle	Gradient memory, node count	Autograd, backward()	Vanishing/ exploding grads	Checkpointing
Optimization & Training	Optimizer state, update costs	State size, convergence rate	Optimizers, training loop	Non-convergence, NaN	LR scheduling, clipping
Architectures & Models	Parameter scaling, compute patterns	Parameters, receptive field	Layers, blocks, full models	Accuracy plateau, overfit	Pruning, distillation
Pipelines & Data	Batching, shuffling, prefetch	Throughput, CPU utilization	Dataset, DataLoader	Data bottleneck, stalls	Parallel loading, caching
Efficiency & Compression	Quantization, pruning tradeoffs	Compression ratio, accuracy delta	INT8, magnitude pruning	Accuracy degradation	Calibration, sparsity
Deployment & Serving	Batch vs streaming, caching	Latency p50/p99, throughput	KV cache, model export	Timeout, memory bloat	Dynamic batching

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

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

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

Listing 1. Memory Profiling. Tensor implementation from Module 01 with explicit memory tracking.

multiplication (Listing 1), making memory a first-class concept. Modules 05–07 build learning infrastructure: data loading (Module 05) provides efficient batching, then automatic differentiation (Module 06) enables gradient computation through progressive disclosure (Section 4), and optimizers (Module 07) use those gradients for parameter updates—students discover Adam's 3× memory overhead through direct measurement (Section 5). The training loop (Module 08) integrates all components. This order is the minimal dependency chain: you cannot build optimizers without autograd (no gradients), cannot build autograd without losses (nothing to differentiate), cannot build losses without layers (no predictions). 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.

Tier 2: Architectures (Modules 09–13). Students apply foundation knowledge to modern architectures, with the tier branching into parallel *Vision* and *Language* tracks. This bifurcation reflects domain-specific requirements: vision processes spatial grids (images), while language processes variable-length sequences (text). Both tracks build on the DataLoader patterns from Module 05 and training infrastructure from Module 08. TinyTorch ships with two custom educational datasets: **TinyDigits** (5,000 grayscale handwritten digits) and **TinyTalks** (3,000 synthetically-generated conversational Q&A pairs). These datasets are deliberately small and offline-first: they require no network connectivity during training, consume minimal storage (<50MB combined), and train in minutes on CPU-only hardware. This design

Table 3. Module Progression. Each module teaches both "what" (ML technique) and "how much" (memory/compute costs). Foundation tier (M01–08) establishes core operations with explicit resource tracking. Architecture tier (M09–13) 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 (01–08)				
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	DataLoader	Dataset abstraction, batching, shuffling	Iterator protocol, batch collation overhead
06	Fnd	Autograd	Computational graphs, chain rule, backprop	Gradient memory ($2\times$ momentum, $3\times$ Adam)
07	Fnd	Optimizers	SGD, Momentum, Adam, AdamW	Optimizer state memory, in-place updates
08	Fnd	Training	Cosine scheduling, gradient clipping	Peak memory lifecycle, checkpoint tradeoffs
Architecture Tier (09–13)				
09	Arch	Convolutions	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↔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	Acceleration	Vectorization, memory access patterns	Cache locality, SIMD utilization
18	Opt	Memoization	KV-cache for autoregressive generation	$O(n^2)\rightarrow O(n)$ caching, memory-compute tradeoff
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

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 branches into two paths, each following the principle of building components in the order they compose. **Vision** implements Conv2d with seven explicit nested loops making $O(C_{out} \times H \times W \times C_{in} \times K^2)$ complexity visible before optimization. Students discover weight sharing's dramatic efficiency through direct comparison: Conv2d(3→32, kernel=3) requires 896 parameters while an equivalent dense layer needs 98,336 parameters (3072 input features × 32 outputs + 32 bias terms), a 109× reduction demonstrating how inductive biases enable CNNs to learn spatial patterns without brute-force parameterization. Students validate their implementations by training CNNs targeting 65–75% CIFAR-10 accuracy (Krizhevsky and Hinton, 2009; Lecun et al.).

Language progresses through tokenization, embeddings, attention, and complete transformers (Vaswani et al., 2017)—each step transforming the data representation in sequence (text → token IDs → embeddings → contextualized representations). This order is non-negotiable: you cannot compute attention without embeddings (attention needs continuous vectors), cannot create embeddings without tokenization (embedding lookup needs integer indices), cannot tokenize without text (input must exist). Module 10 (Tokenization) teaches a fundamental NLP systems trade-off: vocabulary size controls model parameters (embedding matrix rows × 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, validating their transformer implementations through text generation on TinyTalks.

Tier 3: Optimization (Modules 14–19). Students transition from "models that train" to "systems that deploy." The optimization tier follows a deliberate pedagogical structure: *Measure → Model-Level → Runtime → Validate*. Profiling (14) teaches measuring time, memory, and FLOPs (floating-point operations), introducing Amdahl's Law: optimizing 70% of runtime by 2× yields only 1.53× overall speedup because the remaining 30% becomes the new bottleneck. This teaches that optimization is iterative and measurement-driven.

The tier then divides into two optimization categories. **Model-level optimizations** (Modules 15–16) change the model itself: Quantization (15) achieves 4× compression (FP32→INT8) with 1–2% accuracy cost, while Compression (16) applies pruning and distillation for 10× shrinkage. These techniques permanently modify model weights and architecture.

Runtime optimizations (Modules 17–18) change how execution happens without modifying model weights. Acceleration (17) teaches general-purpose optimization: vectorization exploits SIMD instructions for 10–100× convolution speedups, memory access pattern optimization improves cache locality, and kernel fusion eliminates intermediate memory traffic. These techniques apply universally to any numerical computation. Memoization (18) then applies domain-specific optimization to transformers through KV caching: students discover that naive autoregressive generation recomputes attention keys and values at every step—generating 100 tokens requires 5,050 redundant computations (1+2+...+100). By caching these values, students transform $O(n^2)$ generation into $O(n)$, achieving 10–100× speedup and understanding why this optimization is essential in systems like ChatGPT and Claude for economically viable inference.

Benchmarking (19) teaches statistical rigor in performance measurement: students learn that single measurements are meaningless (performance varies 10–30% across runs due to thermal throttling, OS noise, and cache state), implement confidence intervals and warmup protocols, and discover when a 5% speedup is statistically significant versus noise.

Capstone (Module 20). The capstone builds submission infrastructure enabling ML competitions and reproducible benchmarking. Inspired by MLPerf (Mattson et al., 2020; Reddi et al., a), students implement standardized submission formats (JSON schemas with system metadata, normalized metrics, and improvement calculations), benchmark reporting classes, and complete optimization workflows. This infrastructure serves dual purposes: students demonstrate mastery by applying the complete optimization pipeline (Profile → Optimize → Benchmark → Submit) to their prior implementations, and the submission system enables future community challenges (classroom competitions, optimization contests, reproducible research comparisons). Students learn that production ML engineering requires not just optimization techniques but shareable, reproducible infrastructure for validating and comparing results.

3.3 Module Structure

Each module follows a consistent **Build → Use → Reflect** pedagogical cycle that integrates implementation, application, and systems reasoning. This structure addresses

multiple learning objectives: students construct working components (Build), validate integration with prior modules (Use), and develop systems thinking (Meadows, 2008) through analysis (Reflect).

Build: Implementation with Explicit Dependencies. Students implement components in Jupyter notebooks (`*.py`) with scaffolded guidance. Each module begins with a *connection map*: a visual diagram showing which prior modules students must have completed (prerequisites), what the current module teaches (focus), and what capabilities become available after completion (unlocks). For example, Module 09 (Convolutions) shows prerequisites of Modules 01–08, focus on spatial operations, and unlocks CNN architectures for image classification. These maps make dependency relationships explicit, helping students understand where each module fits in the larger framework architecture. To maintain consistency across all 20 modules, Module 05 (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 06 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 06: 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 06's autograd (gradient flow through kernels), Module 07's optimizers (parameter updates), and Module 08'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 (Perkins and Salomon, 1992): can students apply learned concepts to novel scenarios not seen in exercises?

3.4 Milestone Arcs

Milestones are validation checkpoints where students recreate historical ML breakthroughs using exclusively their own TinyTorch implementations. Rather than toy exercises, milestones are real tasks: training Rosenblatt's 1958 Perceptron, solving Minsky's XOR problem, classifying CIFAR-10 images with CNNs, generating text with transformers. Each milestone requires importing only from `tinytorch.*`—every layer, optimizer, and training loop is code the student wrote. Six milestones span ML history from 1958 to present, with each requiring progressively more modules from the growing framework.

Why Milestones Matter. Milestones serve dual pedagogical and validation purposes that differentiate TinyTorch from traditional programming assignments. First, pedagogical motivation through historical framing: Rather than "implement this function," students "recreate the breakthrough that proved Minsky wrong about neural networks," connecting implementation work to historically significant results. This instantiates Bruner's spiral curriculum (Bruner, 1960): students train neural networks 6 times with increasing sophistication, each iteration deepening understanding through historical progression from 1958 (Perceptron) to present (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 01–08 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 to present, 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 08): 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 08): Handwritten digit recognition demonstrating backpropagation's power. Requires Modules 01–08 working together (tensor operations, activations, layers, losses, dataloader, 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–Present: MLPerf Benchmarks** (after Module 20): Building submission infrastructure for reproducible benchmarking and future competitions. MLPerf (Mattson et al., 2020; Reddi et al., a) established industry-standard benchmarking in 2018, and systems engineering has only grown more critical since. Students apply the complete optimization pipeline (Profile → Optimize → Benchmark → Submit) to prior implementations, generating standardized results. This validates that students understand production ML engineering: not just optimization techniques, but shareable infrastructure for validating and comparing results—the systems skills that now determine who can actually deploy at scale.

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)**: Achieves meaningful CIFAR-10 classification accuracy, showing convolutional feature extraction and spatial hierarchies work correctly.
- **Milestone 5 (2017 Transformer)**: Generates coherent multi-token text continuations on TinyTalks dataset, demonstrating functional attention mechanisms and autoregressive generation.

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

  def backward(self, gradient=None):
    """No-op until Module 06"""
    pass

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

Listing 2. Dormant Gradient Infrastructure. Module 01 Tensor includes `.backward()`, `.grad`, and `.requires_grad`—visible but inactive until Module 06 activation.

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

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

# Module 06 usage
enable_autograd()
x = Tensor([3.0], requires_grad=True)
y = x * x  # y = 9.0
y.backward()
print(x.grad)  # [6.0] - dy/dx = 2x
```

Listing 3. Autograd Activation. Module 06 monkey-patches Tensor to enable gradient computation.

- **Milestone 6 (2018–Present: MLPerf Benchmarks)**: Generates valid benchmark submissions with reproducible metrics following the optimization pipeline, demonstrating mastery of systems engineering practices essential for production deployment.

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 06 activates them through monkey-patching (Listings 2 and 3). Figure 3 visualizes this activation timeline across the curriculum.

4.2 Pedagogical Justification

Progressive disclosure is grounded in cognitive load theory (Sweller) and threshold concept pedagogy (Meyer and Land, 2003). The cognitive load hypothesis (early API familiarity reduces future load when features activate) competes with potential split-attention effects from visible but dormant features. Autograd represents a threshold concept—transformative and troublesome—made visible early (dormant) but activatable when students are cognitively ready. Empirical measurement planned for Fall 2025 (Section 8) will quantify the net cognitive load impact.

Implementation Choice: Monkey-Patching vs. Inheritance. Alternative designs include inheritance (`TensorV1/TensorV2`) or composition. We chose monkey-patching because it mirrors PyTorch 0.4's Variable-Tensor merger via runtime consolidation. The software engineering trade-off (global state modification) is explicitly discussed in Module 06's reflection questions.

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()`: immedi-

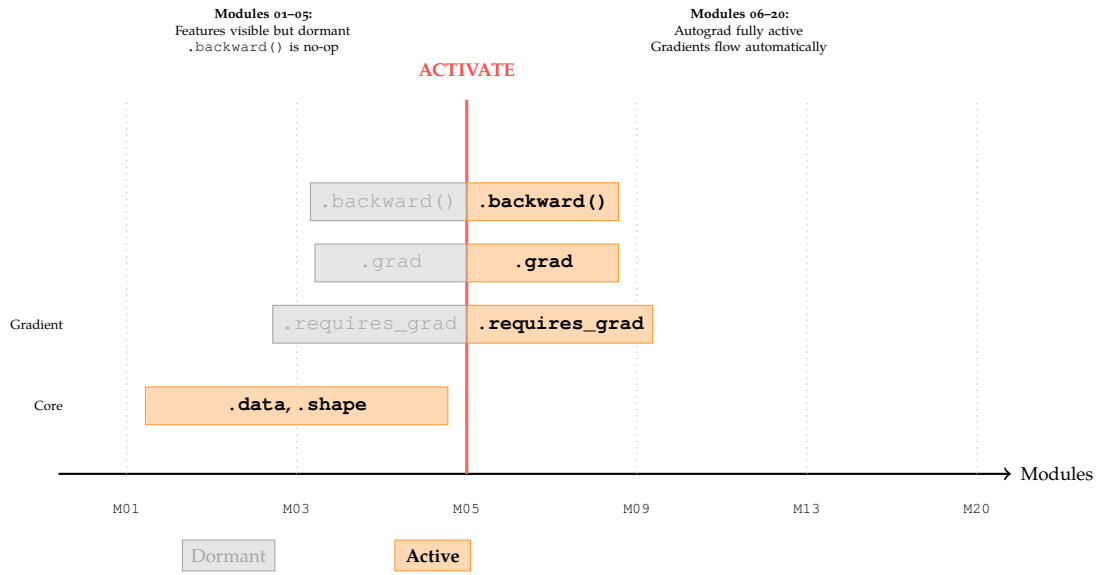


Figure 3. Progressive Disclosure. Runtime feature activation manages cognitive load. From Module 01, students see the complete Tensor API including gradient methods (`.backward()`, `.grad`, `.requires_grad`), but these features remain dormant (gray, dashed). In Module 06, runtime enhancement activates full autograd functionality (orange, solid) without breaking earlier code. Three learning benefits: (1) students learn the complete API early, avoiding interface surprise later; (2) Module 01 code continues working unchanged when autograd activates (forward compatibility); (3) visible but inactive features create curiosity-driven questions motivating curriculum progression.

ate 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.

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 ($1.2\text{M} \times 224 \times 224 \times 3 \times 4$ 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–$100\times$ larger than parameters). This decomposition enables accurate capacity planning for training runs.

5.2 Phase 2: Analyzing Complexity Through Transparent Implementations

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

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

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

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

Listing 4. Explicit Convolution. Seven nested loops reveal $O(C_{out} \times H \times W \times C_{in} \times K^2)$ complexity.

manual weight-input multiplication loops before Module 08 introduces NumPy's `@` operator; Module 09 teaches convolution through seven nested loops before Module 17 vectorizes with NumPy operations. This progression ensures students understand *what* operations do (and their complexity) before learning *how* to optimize them. Pure Python transparency enables this pedagogical sequencing: students can inspect every operation without navigating compiled C extensions or CUDA kernels.

Students calculate: CIFAR-10 batch (128, 3, 32, 32) through 32-filter 5×5 convolution: $128 \times 32 \times 28 \times 28 \times 3 \times 5 \times 5 = 241$M multiply-accumulate operations. This concrete measurement motivates Module 17's vectorization (10–100× speedup) and explains why CNNs require hardware acceleration.

Experiencing Performance Reality. Table 4 shows TinyTorch's deliberate slowness compared to PyTorch (100–10,000× 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×?"). Where foundation modules taught

Table 4. Performance Comparison. TinyTorch vs PyTorch (CPU). Pure Python implementations with explicit loops are 100–10,000× 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×
`conv2d` (CIFAR batch)	97 s	10 ms	10,017×
`softmax` (10K elem)	6 ms	0.05 ms	134×

calculating footprints and counting FLOPs, optimization modules teach systematic improvement through profiling-driven iteration.

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

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

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

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

The Optimization Tier completes the systems-first integration arc: students who calculate memory in Module 01, count FLOPs in Module 09, and optimize deployment in Modules 14–19 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.

Textbook Integration. 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 06: 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).

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 (01–08) 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 06 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 01–08). Students build core framework internals from scratch: tensors, activations, layers, losses, dataloader, 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 scal-

ing ($O(N^2)$ attention), and recognize architectural trade-offs (109× 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, acceleration, memoization, 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 Benchmarks capstone) demonstrating reproducible benchmarking and optimization workflows. 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 06 autograd to understand `loss.backward()`, or Module 09 convolution to demystify `torch.nn.Conv2d`).

6.3 ML Systems Competency Coverage

The ML Systems Competency Matrix (Table 2) established 40 cells as the design framework. Table 5 maps TinyTorch's 20 modules to this framework, demonstrating systematic coverage across knowledge areas and capability levels.

Coverage Analysis. TinyTorch provides full coverage of 36 cells (90%), partial coverage of 3 cells (7.5%), and an intentional gap in 1 cell (2.5%). Full coverage means students implement working code, measure relevant metrics, and receive automated assessment. Partial coverage indicates conceptual instruction without complete implementation.

Intentional Gap. One cell remains uncovered by design:

- **Pipelines-Optimizes** (parallel data loading, GPU prefetch): Requires multi-threaded I/O and GPU memory management beyond CPU-only scope. Students understand concepts through M05 but don't implement production-grade parallel loaders.

Partial Coverage. Three cells receive conceptual treatment: **Tensors-Optimizes** (memory pooling taught conceptually, students implement in-place operations but not allocator design), **Graphs-Optimizes** (activation checkpointing explained as memory-compute tradeoff but not implemented), and **Deployment-Debugs** (timeout debugging discussed without production environment practice). These represent deliberate scope boundaries. Students completing TinyTorch are prepared to address these areas through subsequent coursework or industry experience, building on strong foundations across the remaining 39 cells.

6.4 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.

6.4.1 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).

6.4.2 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

Table 5. Competency Coverage. Each cell shows which modules address that competency. ✓ = full coverage through implementation and assessment. ○ = partial coverage through conceptual instruction. – = intentional gap where production infrastructure exceeds accessibility-first scope. Of 40 cells, 36 receive full coverage, 3 receive partial coverage, and 1 represents an intentional gap.

Knowledge Area	Knows	Measures	Implements	Debugs	Optimizes
Tensors & Memory	✓ M01	✓ M01, M14	✓ M01	✓ M01	○ M18
Operations & Compute	✓ M01, M02	✓ M14	✓ M01, M02, M09, M12	✓ M02, M09	✓ M18
Graphs & Differentiation	✓ M06	✓ M06, M14	✓ M06	✓ M06	○ conceptual
Optimization & Training	✓ M07, M08	✓ M07, M14	✓ M07, M08	✓ M08	✓ M08
Architectures & Models	✓ M03, M09–M13	✓ M14	✓ M03, M09–M13	✓ milestones	✓ M16
Pipelines & Data	✓ M05	✓ M05, M14	✓ M05	✓ M05	– CPU-only
Efficiency & Compression	✓ M15, M16	✓ M15, M16, M19	✓ M15, M16	✓ M15, M16	✓ M15, M16
Deployment & Serving	✓ M17	✓ M19	✓ M17	○ limited	✓ M17

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

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

Listing 5. NBGrader Structure. Cell metadata defines point allocation and solution delimiters.

`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, 1978) makes educational objectives explicit while enabling automated grading. The `name` field identifies the exercise, `points` assigns weight, and the description provides context before students see code cells.

Handling Autograder Edge Cases: Pure Python convolution (Module 09) may exceed default 30-second timeout on slower hardware; we set 5-minute timeouts and provide vectorized reference solutions for comparison. Critical modules (06 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).

6.5 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), and **point allocations** reflecting pedagogical priorities (Module 06 Autograd: 100 points; Module 01 Tensor: 60 points). Students validate correctness locally before submission, enabling immediate feedback.

This infrastructure enables deployment in MOOCs

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

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

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

Listing 6. Progressive Imports. Framework capabilities grow module-by-module as students complete implementations.

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

6.6 Package Organization

Unlike tutorial-style notebooks creating isolated code, TinyTorch modules export to a package structure inspired by PyTorch's API organization. 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 06, 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`. TinyTorch modules are developed as Python source files using Jupytext percent format (`src/*/*.py`), with Jupyter notebooks generated for student distribution. The build system maintains single source of truth: developers edit `src/*/*.py` 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.

6.7 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 strategies).

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.

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

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 06 (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 06) 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 autograd 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 limitations 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 100–1000× slowdown versus PyTorch (Table 4) is acceptable when pedagogical goal is understanding internals, not training production models.

Incremental complexity management: GPU program-

ming 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.

Experimentation constraints: The performance tradeoff discussed above (Section 7.1) limits practical experimentation. Students complete milestones (65–75% CIFAR-10 accuracy, transformer text generation) but cannot iterate rapidly on architecture search or hyperparameter tuning.

Energy consumption measurement: While TinyTorch covers optimization techniques with significant energy implications (quantization achieving 4× compression, pruning enabling 10× model shrinkage), the curriculum does not explicitly measure or quantify energy consumption. 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 06 (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., 1989)), its pedagogical effectiveness requires empirical validation through controlled classroom studies. We commit to the following validation roadmap:

Phase 1: Pilot Deployment (Fall 2025, $n = 30$–50 students). 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 06 (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, $n = 100$–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 sec-

tions 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 published 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.), astra-sim (Won et al., 2023)) 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 $\rightarrow$ systems profiling $\rightarrow$ 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 broader educational infrastructure while the open-source model (MIT license) enables collaborative refinement across institutions: instructor discussion forums for pedagogical exchange, shared teaching resources, and empirical validation of learning outcomes.

As described earlier, module 20 (Capstone) culminates the curriculum with competitive systems engineering challenges. Inspired by MLPerf benchmarking (Mattson et al., 2020; 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 tradeoffs (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× 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× 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 01–08, 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

We thank Colby Banbury and Zishen Wan for their feedback and thoughts on this work.

References

Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 2nd edition, 1996. ISBN 9780262510875.

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. ISBN 9780521820608,9780511811432. doi: 10.1017/cbo9780511811432. URL https://doi.org/10.1017/cbo9780511811432.

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.

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

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

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. URL https://www.usenix.org/conference/usenix-winter-1993-conference/nachos-instructional-operating-system.

Allan Collins, John Seely Brown, and Susan E. Newman. Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics. In Lauren B. Resnick, editor, *Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser*, pages 453–494. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

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

Jeremy Howard and Sylvain Gugger. Fastai: A layered api for deep learning. *Information*, 11(2):108. ISSN 2078-2489. doi: 10.3390/info11020108. URL https://doi.org/10.3390/info11020108.

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

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

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

10.1080/07370000802212669. URL https://doi.org/10.1080/07370000802212669.

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

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.

Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabber, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. Mlperf training benchmark. In *Proceedings of Machine Learning and Systems* (*MLSys*), 2020.

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.

Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, 1980. ISBN 978-0-465-04627-0.

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, IEEE. doi: 10.1109/ispass.2019.00042. URL https://doi.org/10.1109/ispass.2019.00042.

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.

David N. Perkins and Gavriel Salomon. Transfer of learning. In Torsten Husén and T. Neville Postlethwaite, editors, *International Encyclopedia of Education*, pages 6452–6457. Pergamon Press, Oxford, 2nd edition, 1992.

Ben Pfaff, Anthony Romano, and Godmar Back. The pintos instructional operating system kernel. *ACM SIGCSE Bulletin*, 41(1):453–457. ISSN 0097-8418. doi: 10.1145/1539024.1509023. URL https://doi.org/10.1145/1539024.1509023. 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-isss60120.2024.00015. URL https://doi.org/10.1109/codes-isss60120.2024.00015.

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 Tejusve 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.

Robert Half. Building future-forward tech teams: New research reveals severity of the technology skills gap amid talent shortage, 2024. URL https://press.roberthalf.com/2024-05-08-New-Robert-Half-Research-Reveals-Severity-of-the-Technology-Skills-Gap-Amid-Talent-Shortage. Survey of nearly 700 technology leaders conducted October–November 2023.

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, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008, 2017. URL https://arxiv.org/abs/1706.03762.

Lev S. Vygotsky. *Mind in Society: The Development of Higher Psychological Processes*. Harvard University Press, Cambridge, MA, 1978. ISBN 9780674576292.

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.

William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *Proceedings of the 2023 IEEE International Symposium on Performance Analysis of Systems and Software* (*ISPASS*), 2023. doi: 10.1109/ISPASS57527.2023.00035. URL https://arxiv.org/abs/2303.14006.

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.