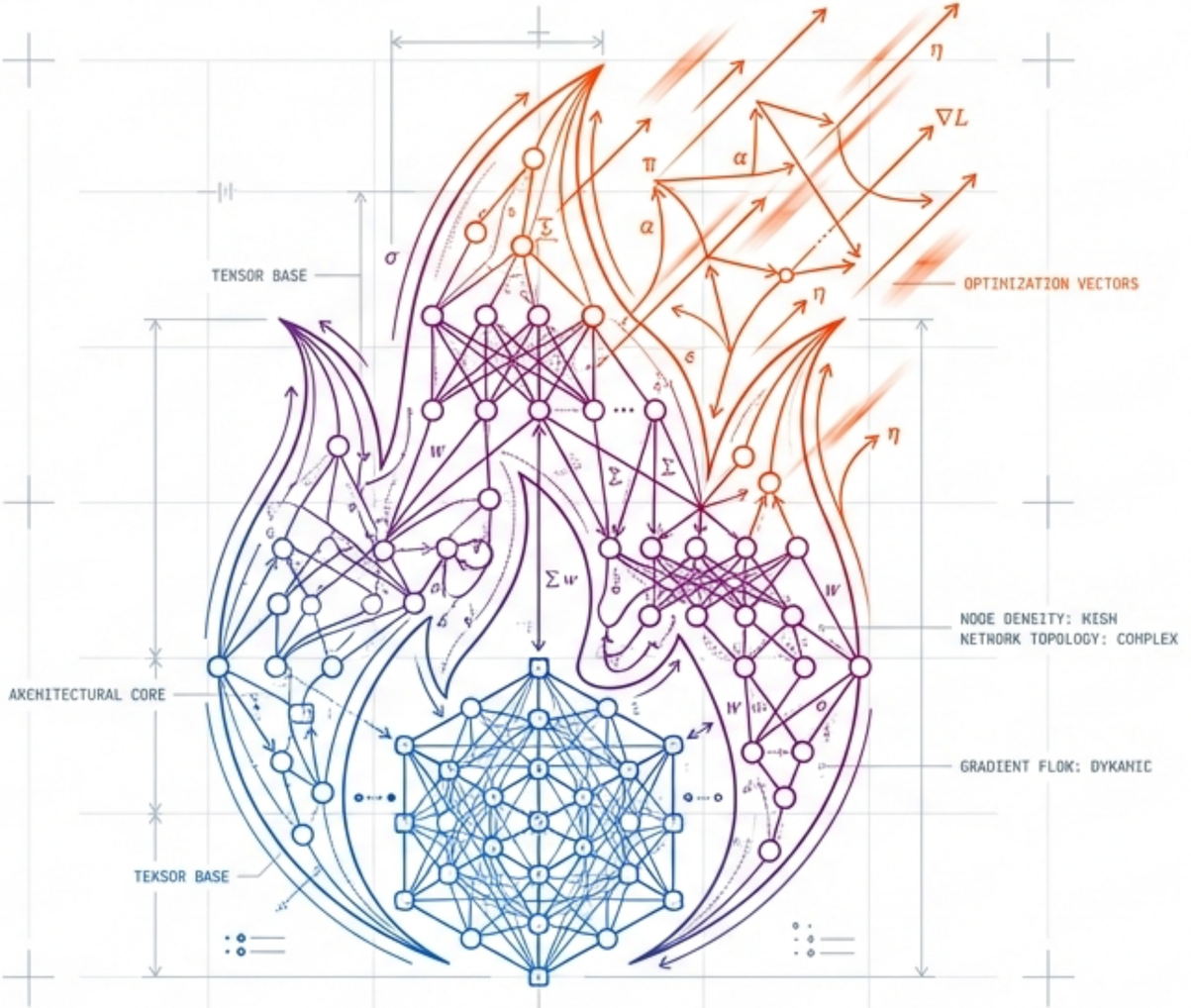


TinyTorch

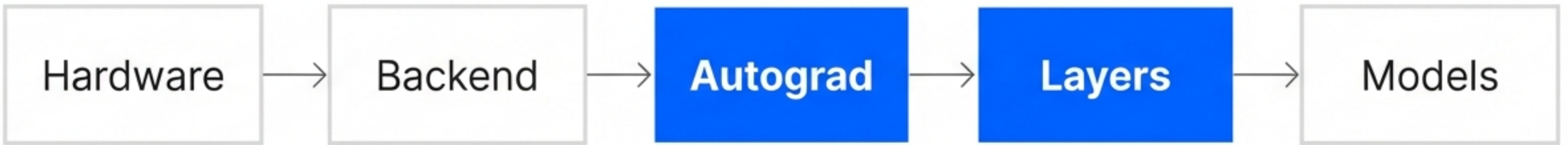
Building a Deep Learning Framework from Scratch

[How Tensors Become Systems]

System Architecture & Learning Path Overview

TinyTorch

From Learning Models to Engineering Systems



A curriculum for building a deep learning framework from scratch to understand the machine inside the black box.

Not a product pitch. An engineering blueprint.

The Gap Between Theory and Systems

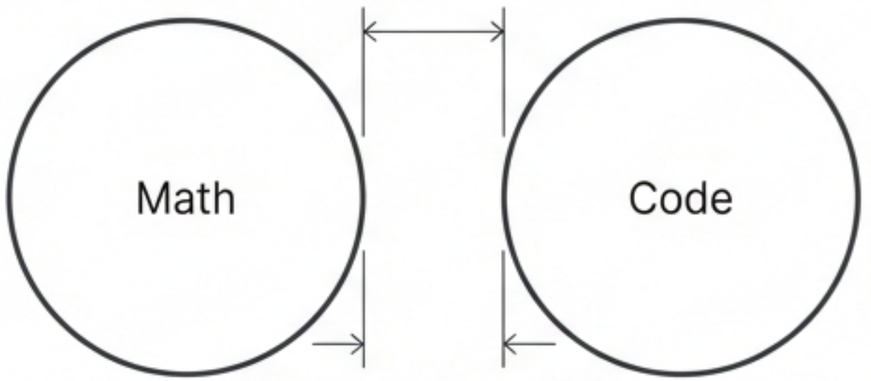
The Status Quo

Theory:

- Focuses on calculus, convergence proofs, loss landscapes.

Application:

- Focuses on `'import torch'`, `'model.fit()'`, hyperparameter tuning.



The Missing Link

The Reality

The Black Box Problem:

- What happens when `'loss.backward()'` returns `NaN`?
- Why is the model memory-bound?
- Why is training slow?

The Consequence:

> *"You cannot debug a system you believe is magic."*

TinyTorch fills the void between the math and the metal.

Why Build TinyTorch?

Core Philosophy: A learning system designed to teach frameworks by building one.

Pure Python/NumPy

Pure Python/NumPy

No C++ or CUDA kernels to hide the logic. We stay in the language you know.

Readable > Fast

Readable > Fast

Optimized for understanding, not execution speed. Simplicity is a feature.

Mirror API

Mirror API

Identical syntax to PyTorch. Skills transfer immediately.

We strip away the optimization to reveal the logic.

The Mirror Effect

TinyTorch

```
from tinytorch import Tensor, Linear

x = Tensor([[1., 2.]], requires_grad=True)
layer = Linear(2, 1)

y = layer(x)
y.backward()
```

Identical API

Same Shape Semantics

Same Broadcasting Rules

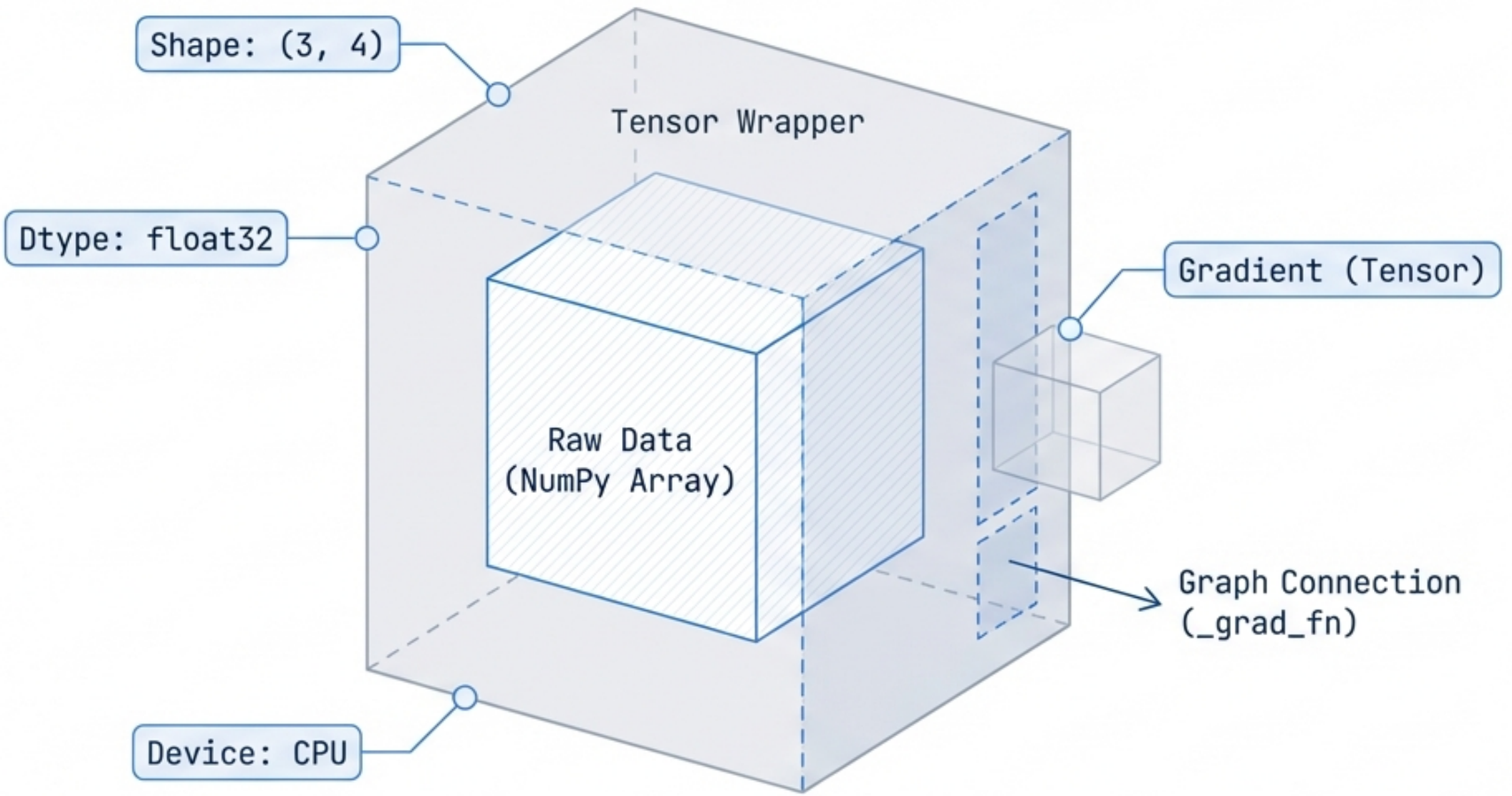
PyTorch

```
import torch
import torch.nn as nn

x = torch.tensor([[1., 2.]], requires_grad=True)
layer = nn.Linear(2, 1)

y = layer(x)
y.backward()
```

The interface is the same. The backend is the lesson.

Module 01: The Atom of ML (Tensors)

A wrapper around NumPy that adds 'Machine Learning Consciousness.'



What We Build:

- [ ] Shape & Device Management
- [ ] Broadcasting Logic (e.g. (3,1) + (3,))
- [ ] Matrix Multiplication (matmul)

Under the Hood of backward()

```
# What happens inside x + y
out = Tensor(x.data + y.data)
out._grad_fn = AddBackward(x, y) # Records history
```

The Chain Rule

$$\frac{dz}{dx} = \left(\frac{dz}{dy}\right) * \left(\frac{dy}{dx}\right)$$

Gradient Accumulation

Why grad += ...? Essential for shared weights (e.g., embeddings used multiple times).

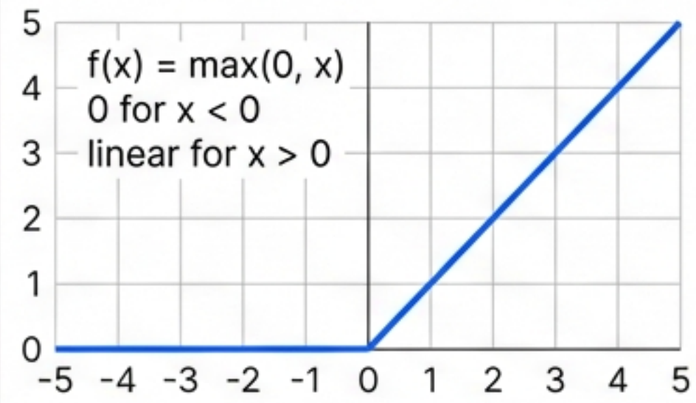
Recursive Traversal

loss.backward() isn't magic; it is simply a graph walk.

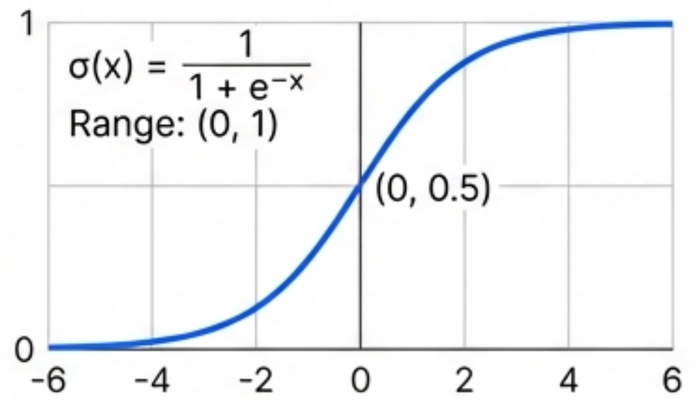
Module 02: Adding Intelligence (Activations)

Without non-linearity, deep nets are just linear regression.

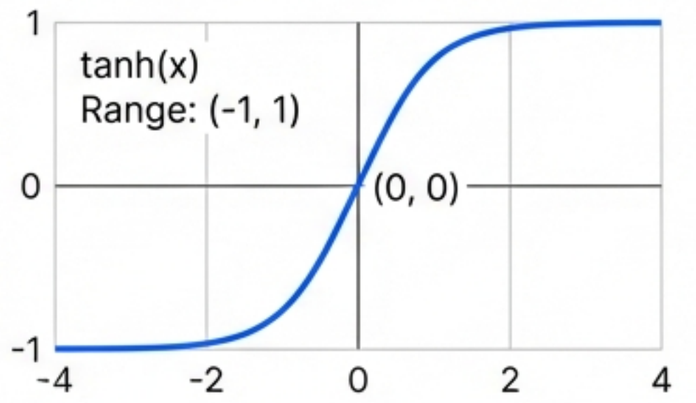
ReLU (Rectified Linear Unit)



Sigmoid



Tanh (Hyperbolic Tangent)



Systems Focus: Numerical Stability

Problem: `exp(1000)` → **Overflow (NaN)**

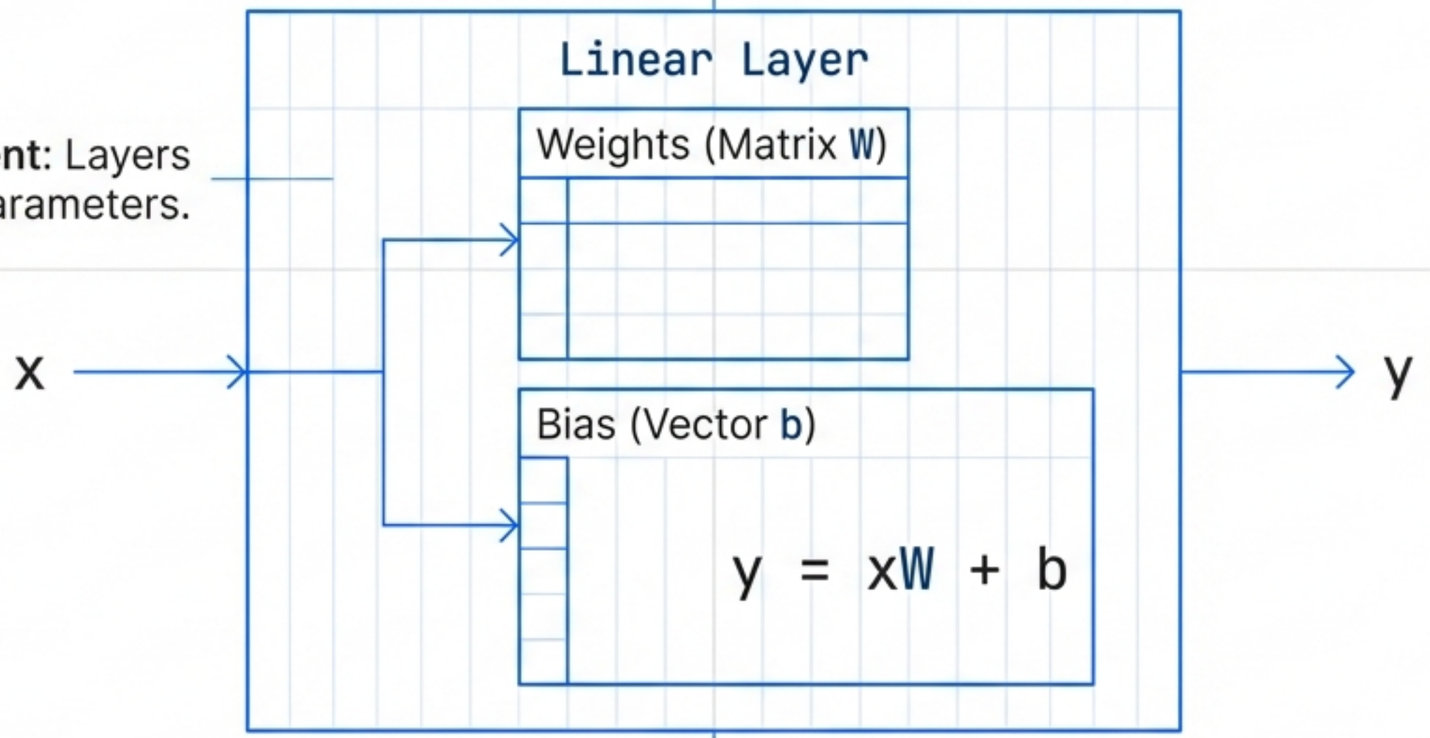
Solution: The **Log-Sum-Exp trick** in Softmax.

We build robust math, not just textbook formulas.

Module 03: The Building Blocks (Layers)

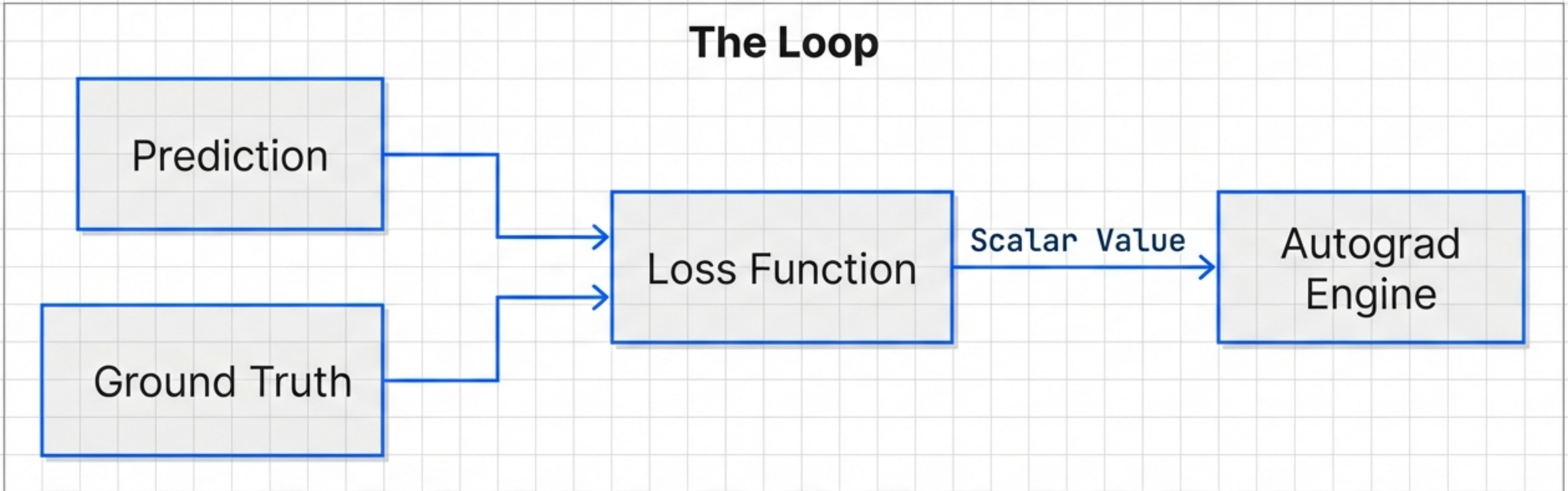
Initialization: Xavier/Glorot logic breaks symmetry.

State Management: Layers are containers for Parameters.



Memory Context: `Linear(784, 256)`
`Linear(784, 256)` = 200,960 params (~800KB).

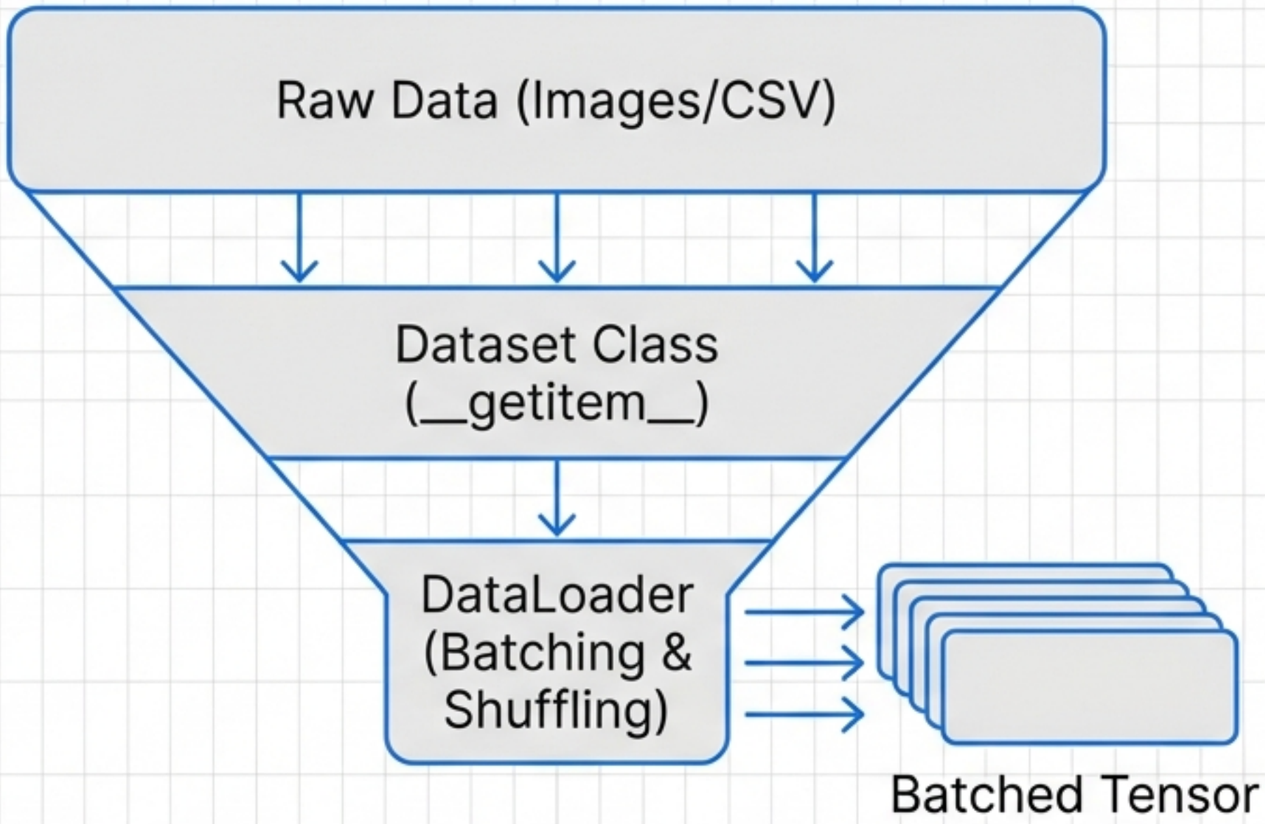
Module 04: The Feedback Signal (Losses)



Implementations:
`MSELoss` **(Regression)**
`CrossEntropyLoss` **(Classification)**

Critical Detail: We use raw logits instead of probabilities for CrossEntropy to ensure numerical stability.

Module 05: The Data Pipeline (DataLoader)



The Bottleneck:
A fast GPU is useless if it is waiting for data.

We build:
- Batching logic (stacking samples)
- Shuffling (statistical independence)

Context:
Processing ImageNet (600GB) requires streaming, not loading.

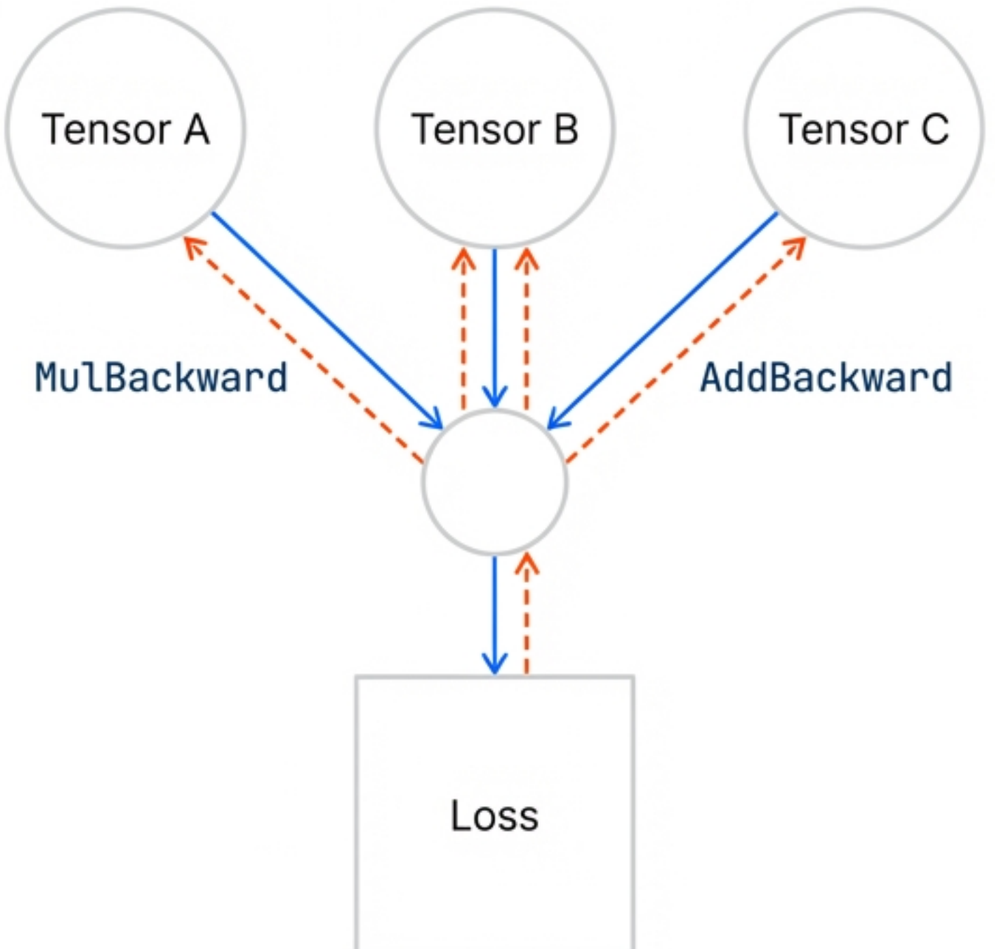
Module 06: The Engine (Autograd)

The Magic Revealed:

Neural networks don't learn without gradients. Computing them by hand is impossible for deep nets.

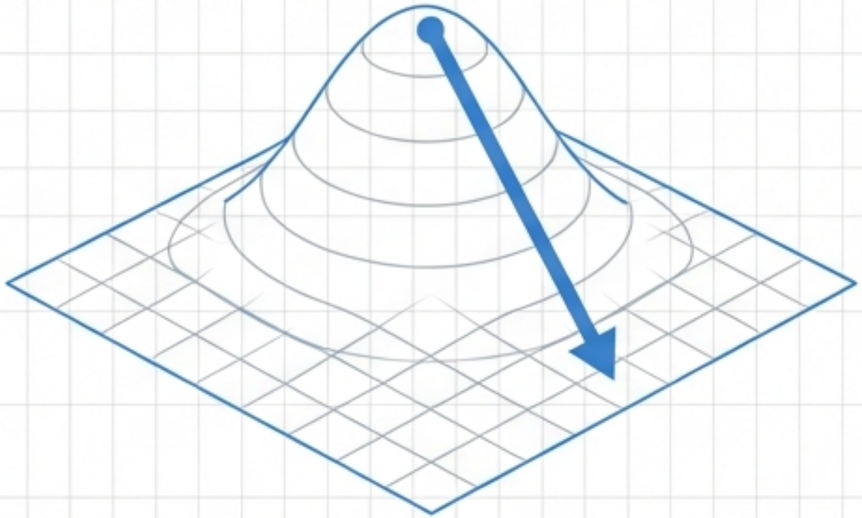
Mechanism:

1. Forward Pass: Build a dynamic Directed Acyclic Graph (DAG).
2. Backward Pass: Traverse the graph in reverse topology.



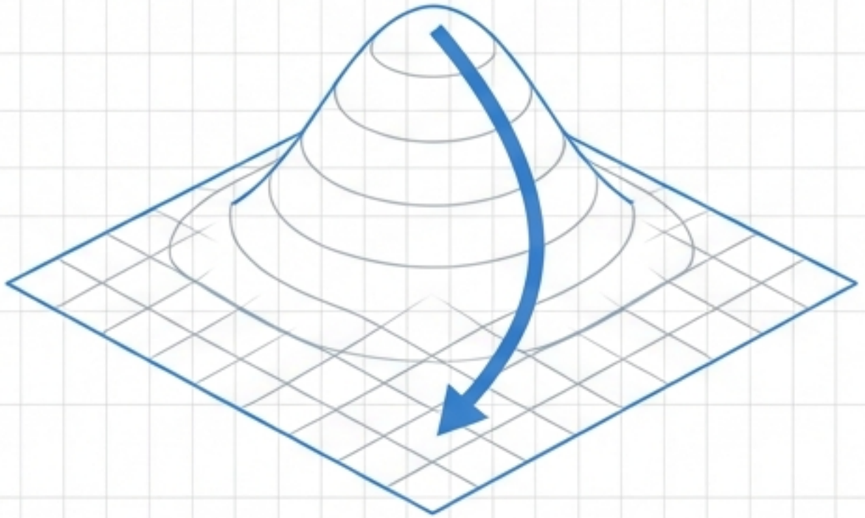
Module 07: Walking Downhill (Optimizers)

SGD



```
w = w - lr * grad
```

Adam



Adaptive learning rates per parameter.

Systems Trade-off Note

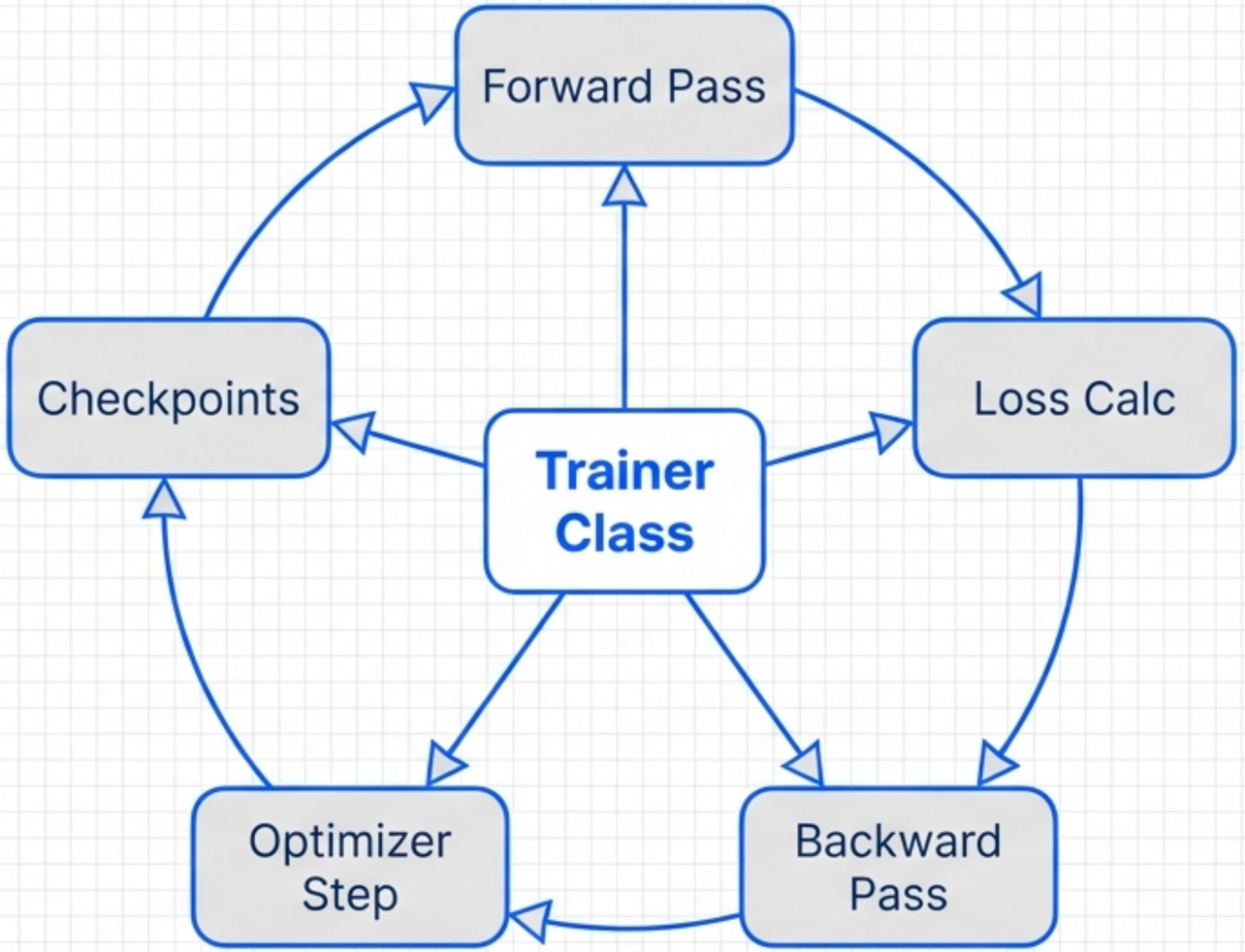
SGD: Low memory.

Adam: Converges faster but uses 3x memory (requires momentum and variance buffers).

Module 08: Orchestration (The Trainer)

Production Features:

- Gradient Clipping **(Safety)**
- Checkpointing **(Persistence)**
- Learning Rate Scheduling **(Cosine Annealing)**

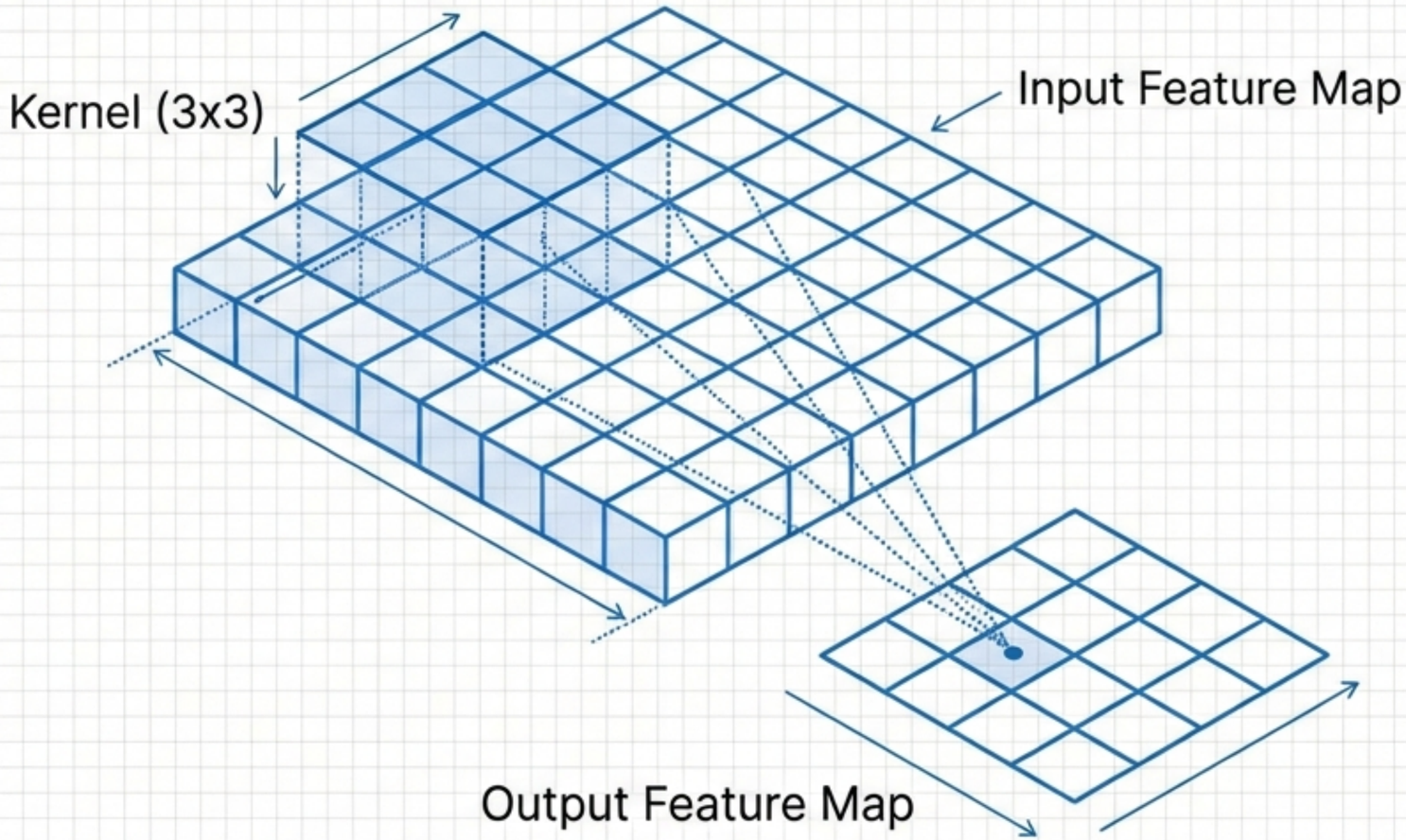


Module 09: Spatial Reasoning (Convolutions)

The Computational Cost:

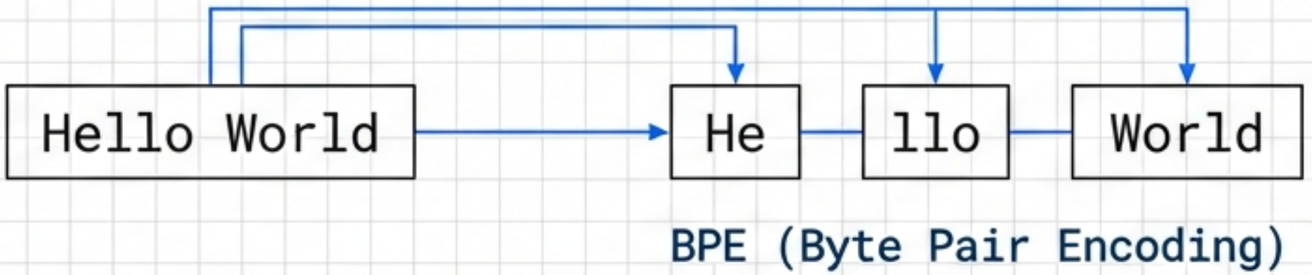
- Naive implementation involves 7-nested loops.
- We implement the naive logic to understand the math, then discuss production “im2col” optimization.
- Application: Building CNNs for CIFAR-10.

Conv2d Operation

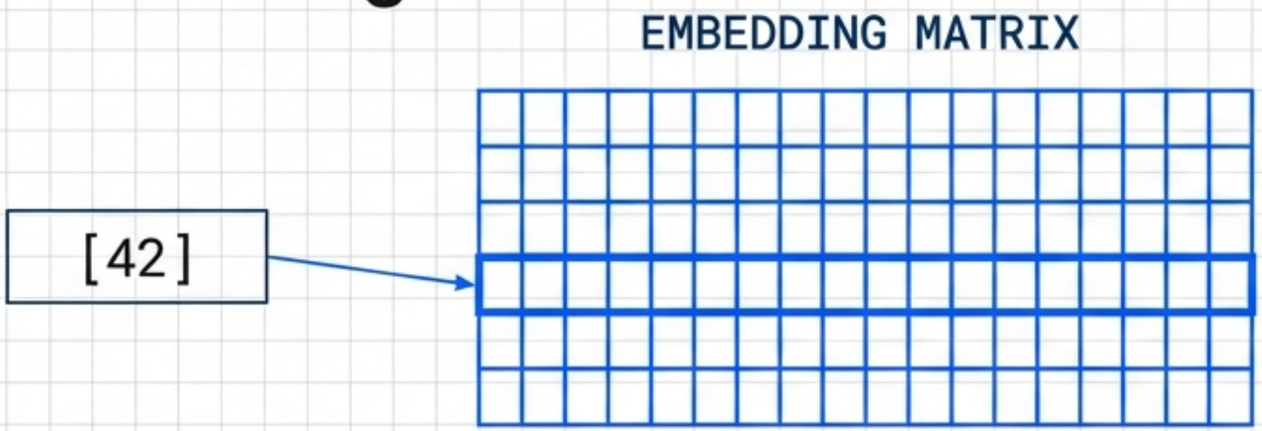


Module 10 & 11: Text to Tensors

Tokenization



Embeddings



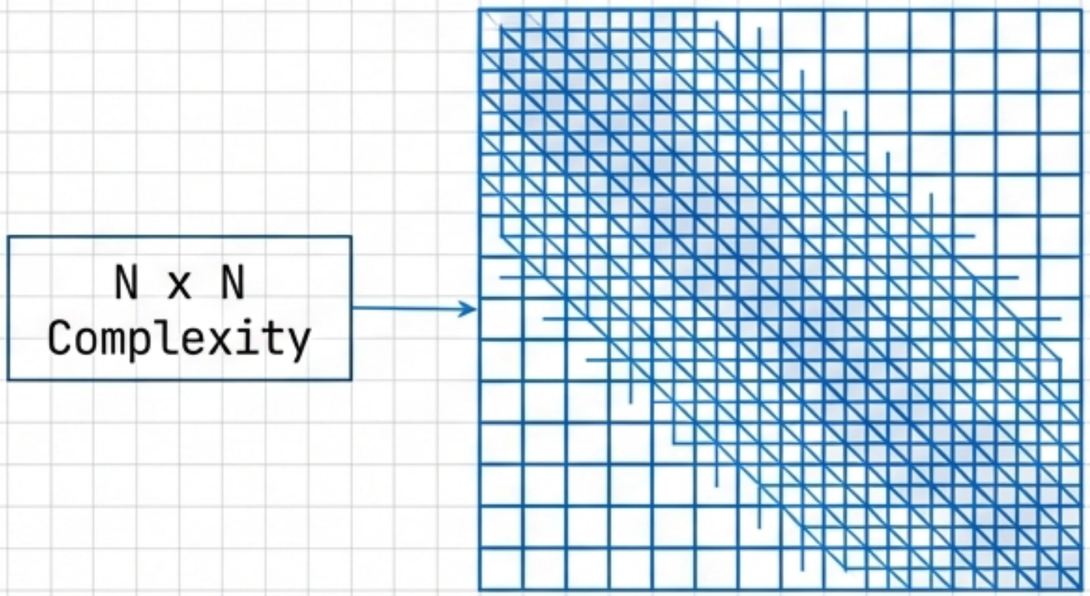
Embedding is just a lookup table.

The Memory Math: GPT-3 Vocab (50k) * Dim (12k) = 600 Million parameters just for the dictionary.

Module 12: The Bottleneck (Attention)

$$\text{softmax}(QK^T/\text{sqr}\sqrt{d})V$$

The Scaling Crisis



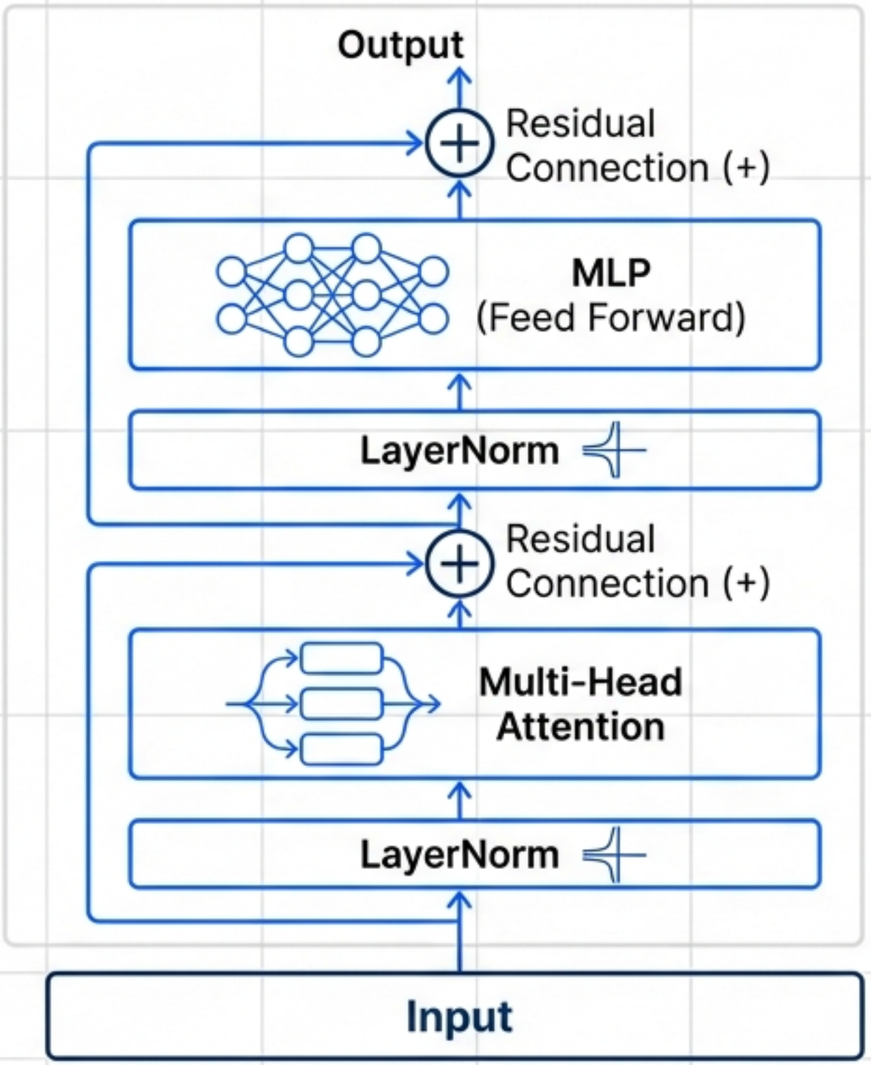
The attention matrix grows quadratically (N^2) with sequence length.

2048 tokens = **Manageable**.
100k tokens = **Explodes**.

Building this explicitly reveals WHY FlashAttention is necessary.

Module 13: The Transformer Architecture

Transformer Block



The Pre-Norm Residual Block.

Why Pre-Norm? Gradient stability in deep networks.

Result: A fully functional GPT model.

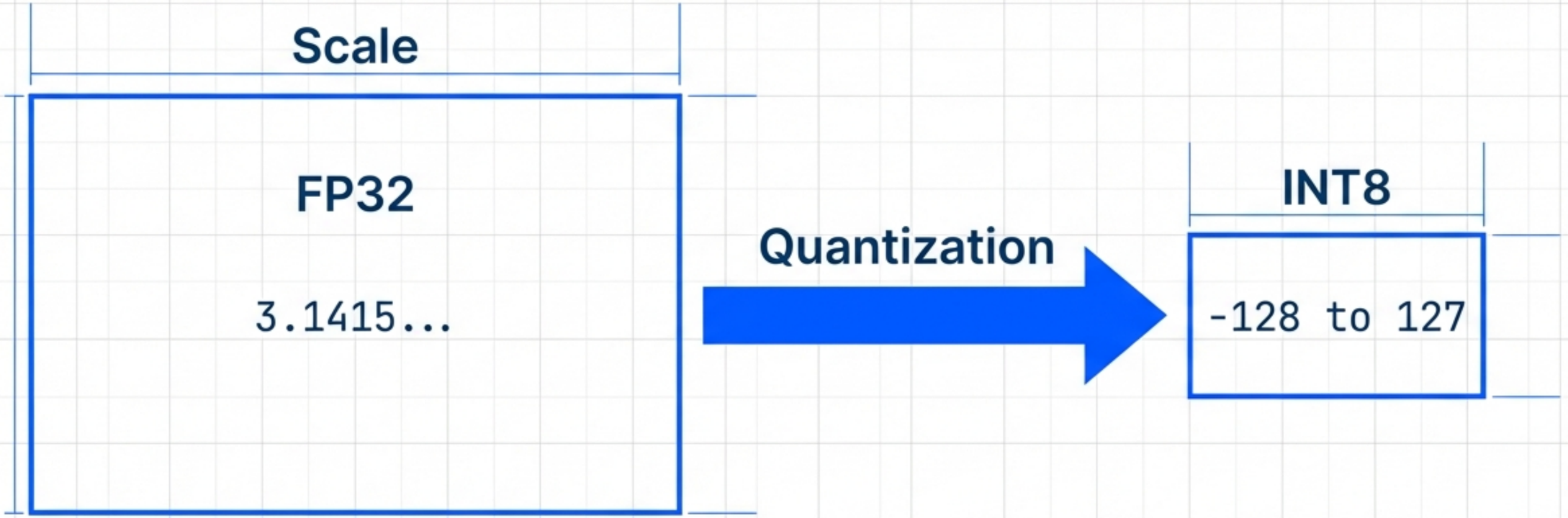
Module 14: The Detective (Profiling)

Don’t guess. Measure.

Parameters	FLOPs	Latency	Memory
Static memory footprint	Computational intensity	Inference time (statistical)	Peak allocation via tracemalloc

Identify if we are Compute-Bound or Memory-Bound.

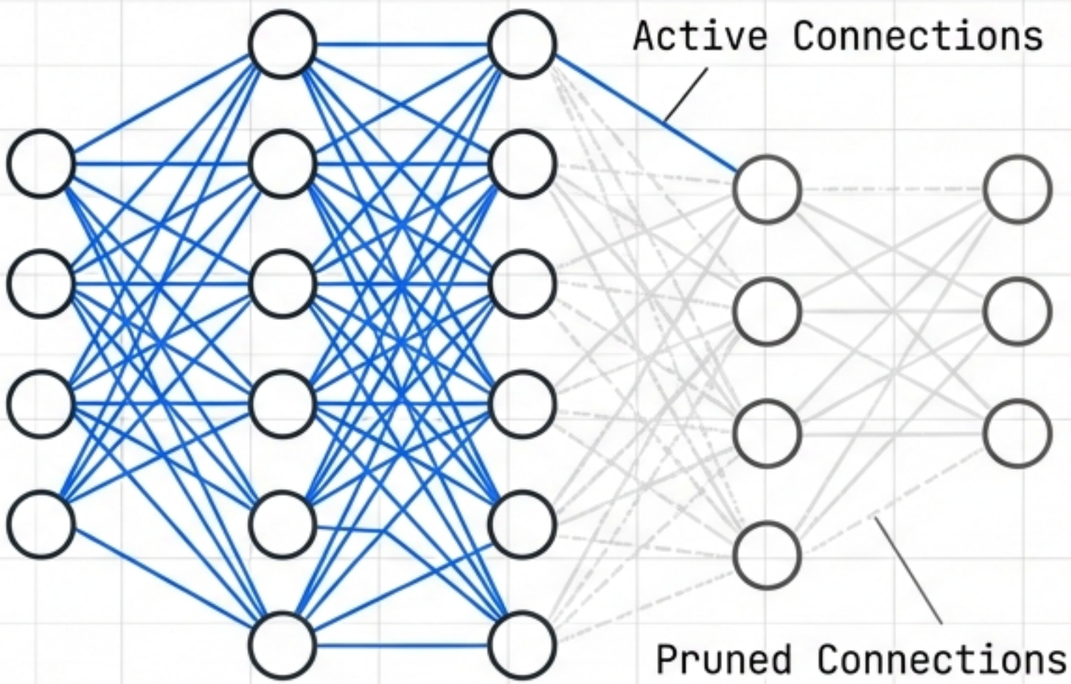
Module 15: Breaking the Memory Wall (Quantization)



Technique: Symmetric Quantization.
Gain: 4x reduction in model size.
Cost: Slight accuracy degradation (minimized via calibration).

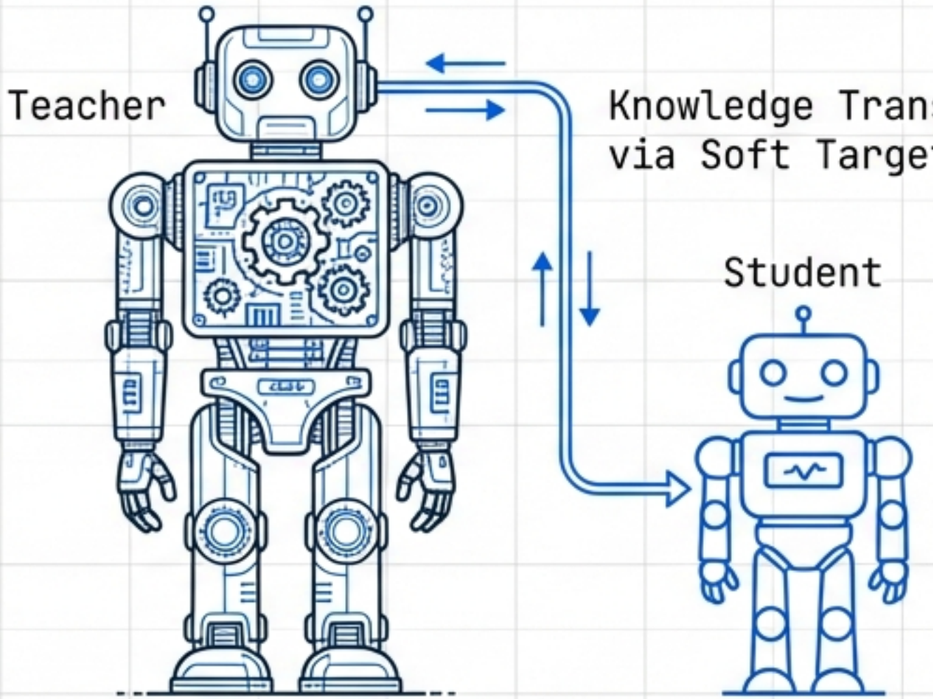
Module 16: Model Compression

Pruning



- Magnitude Pruning: Zeroing out small weights.
- Structured Pruning: Removing entire channels.

Knowledge Distillation

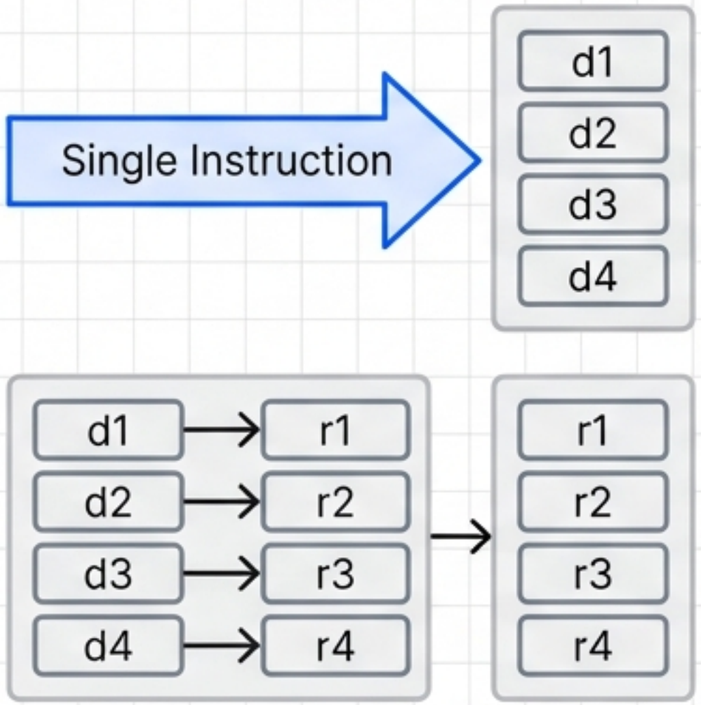


- Teacher (Big) → Student (Small).
- Using Soft Targets (logits) to transfer reasoning.

Module 17: Acceleration & Hardware

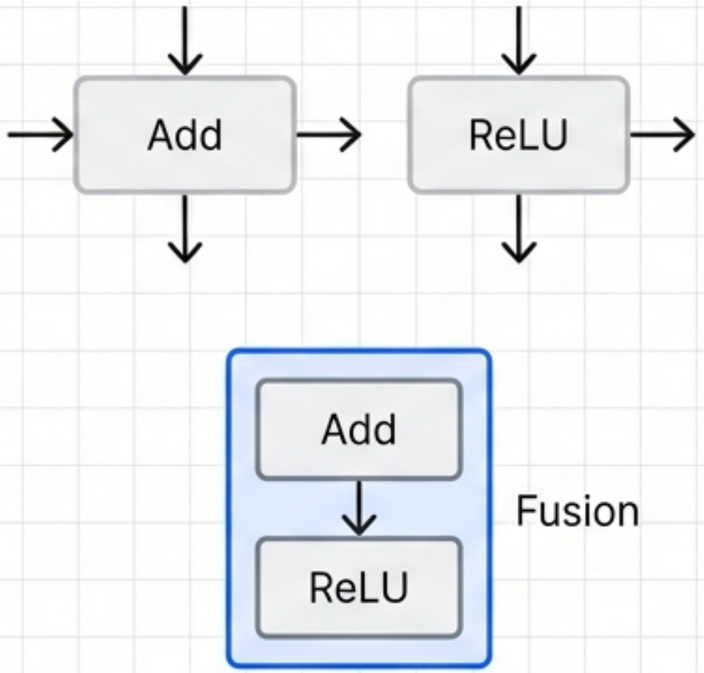
Making the math faster.

Vectorization



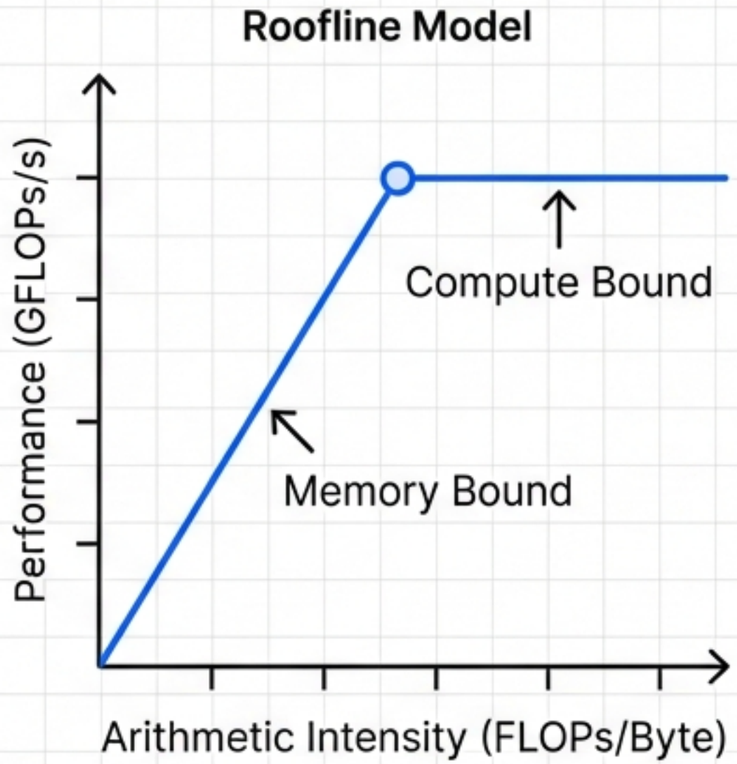
One instruction, multiple data items processed in parallel.

Kernel Fusion



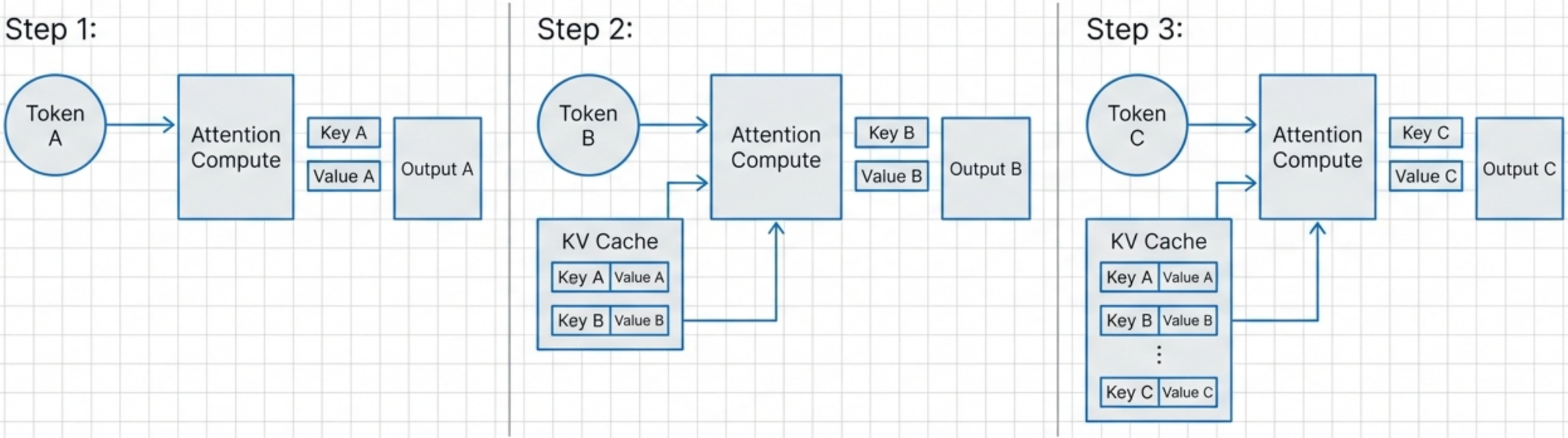
Reduces memory access.

The Roofline Model



Module 18: Optimization for Generation (KV Cache)

In generation, re-computing Attention for past tokens is wasteful O(n^2).



KV Cache: Store the Key/Value matrices.

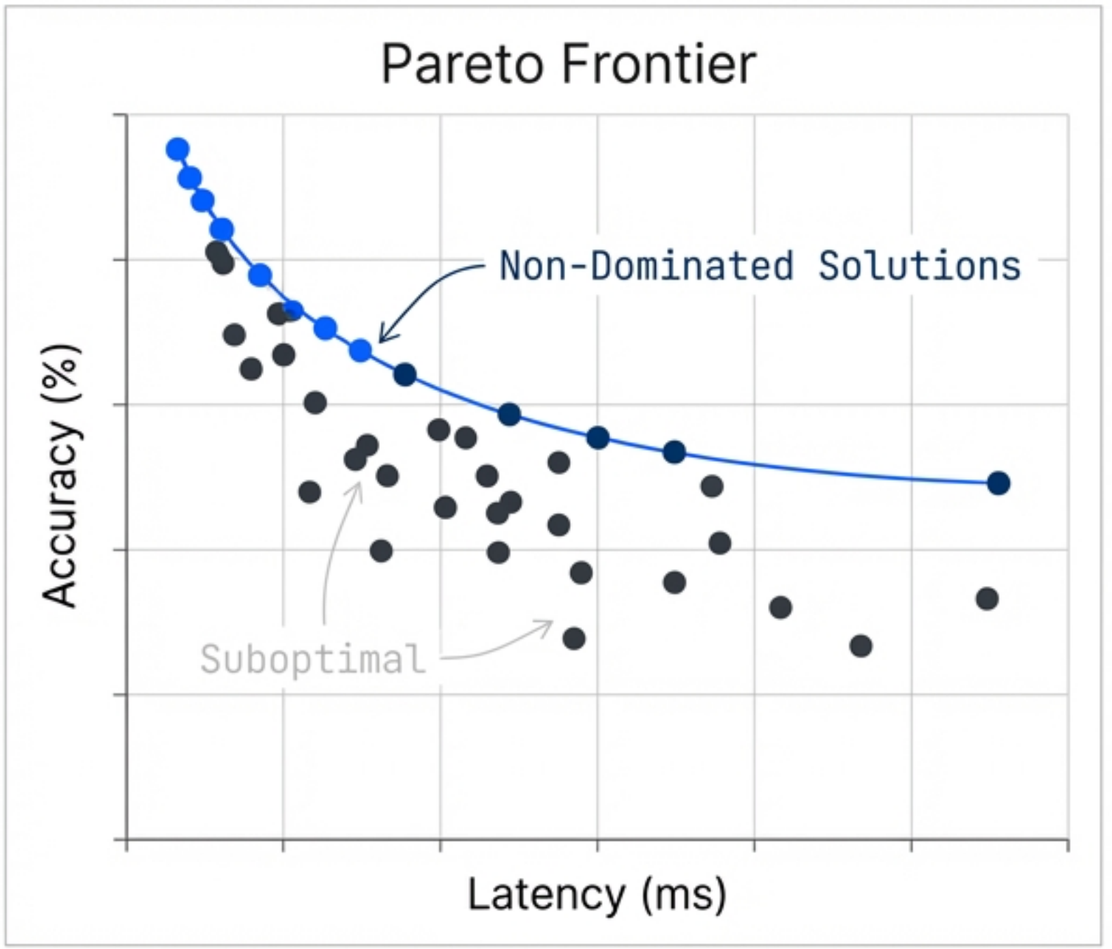
Updates become O(1) per token.

Module 19: Rigorous Benchmarking

Science of Measurement:

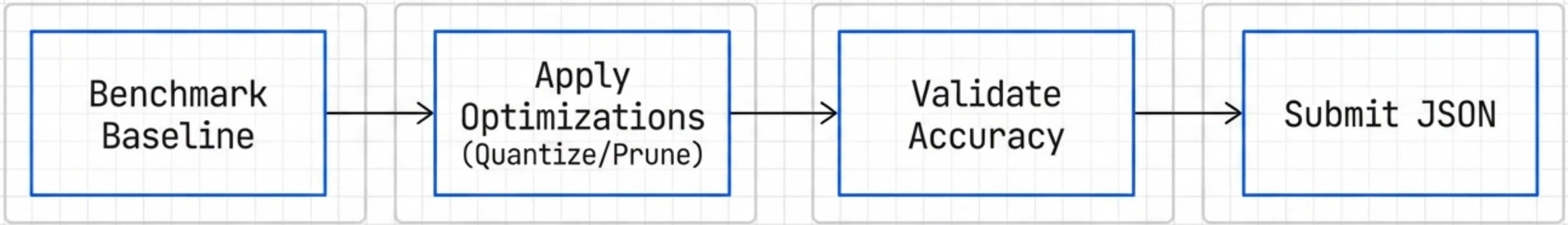
- Warmup runs (avoid cold-start)
- Variance reporting (Mean ± Std Std Dev)

Goal: Move from “feels faster” to “3.4x faster with 95% confidence”.



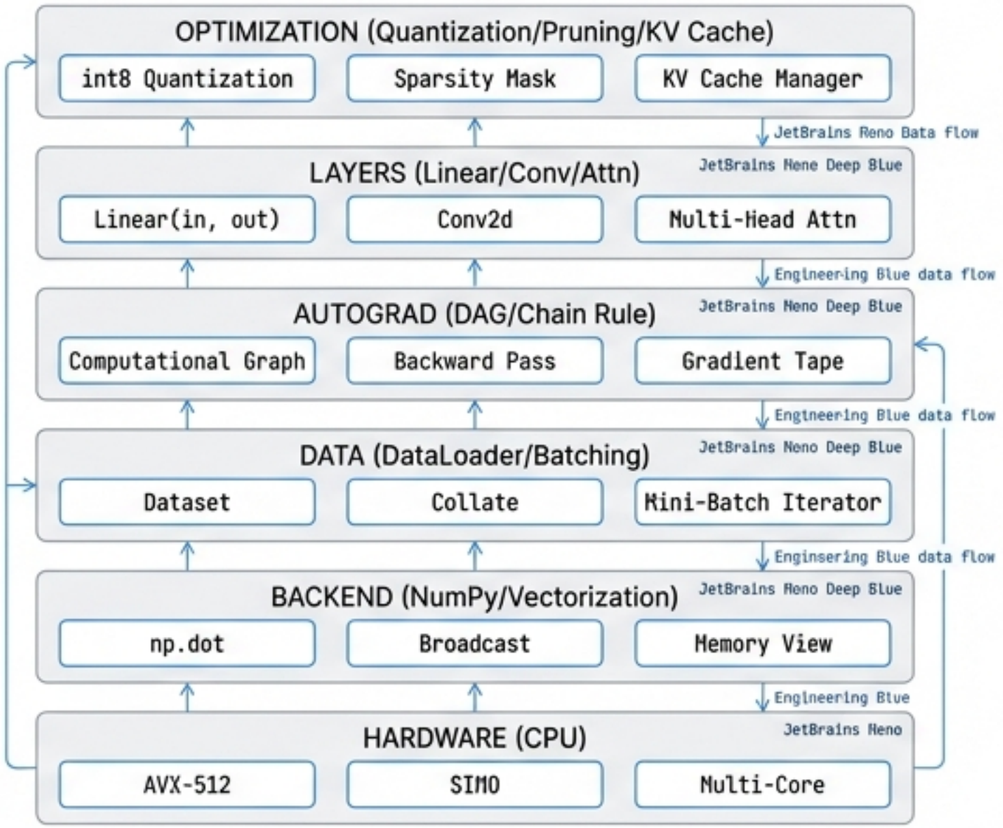
Module 20: The Capstone (TorchPerf Olympics)

The Challenge



- A portfolio-ready artifact proving systems engineering capability.
- Optimizing for Latency, Memory, or Accuracy.

The Engineer’s Perspective



You are no longer just a User.
You are a Builder.
TinyTorch: The starting point for your engineering journey.