OPTIMIZATION TIER

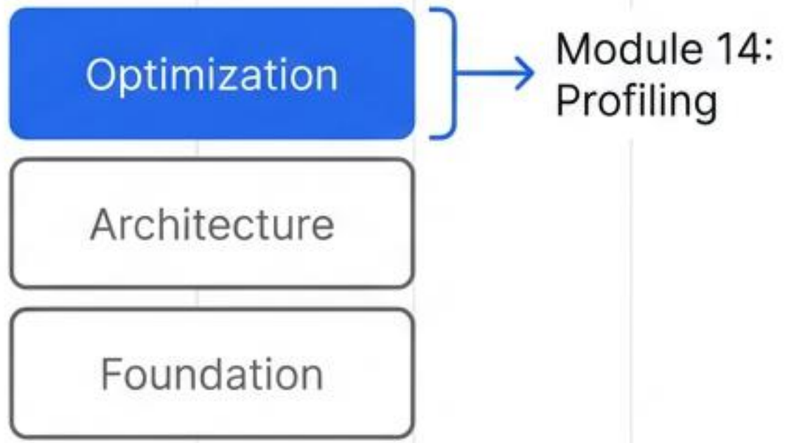
MODULE 14

Profiling

Measurement and optimization for ML systems

Module 14: Profiling

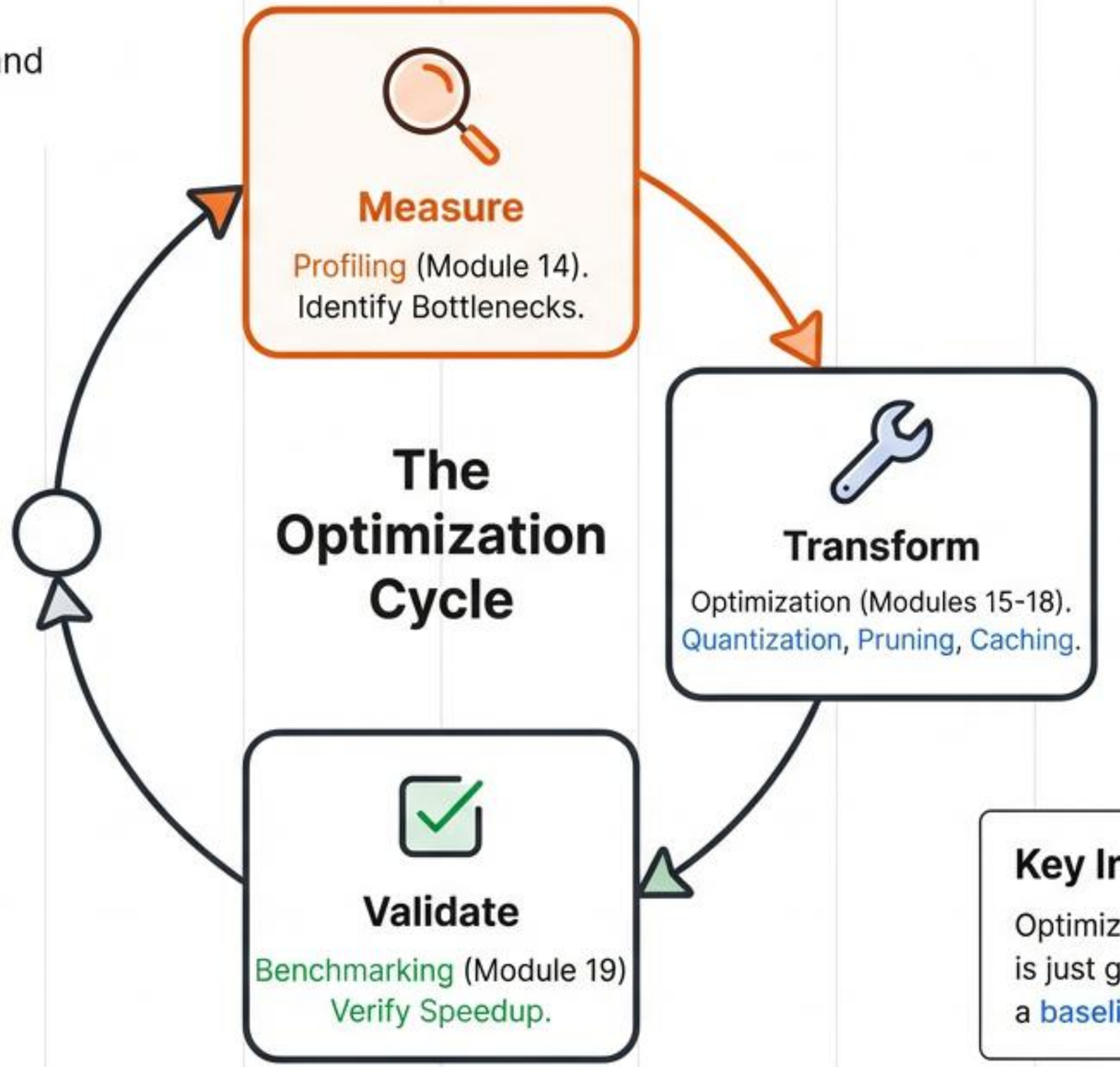
Optimization Begins with with Measurement



You cannot optimize what you cannot measure.

The Optimization Landscape

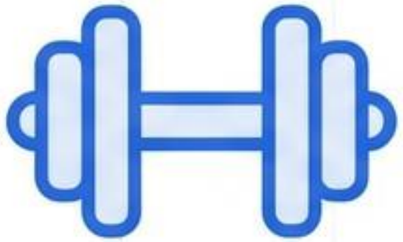
A structured approach to improving performance through iterative analysis and refinement.



Key Insight:

Optimization without measurement is just guessing. We must establish a baseline first.

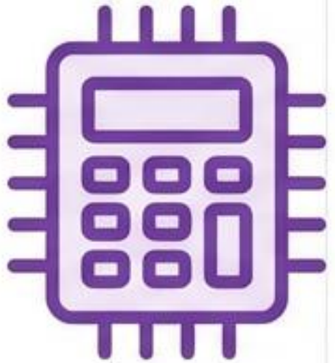
What Are We Measuring?



Model Size (Parameters)

Count (Millions/Billions)

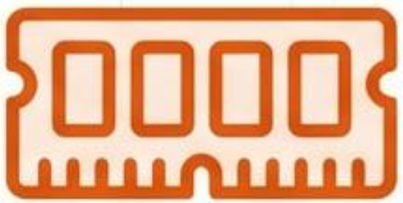
Storage, Fixed Memory Cost.



Computational Cost (FLOPs)

Floating Point Operations

Theoretical complexity, Energy.



Memory Footprint

Peak MB/GB

Batch size limits, OOM errors.

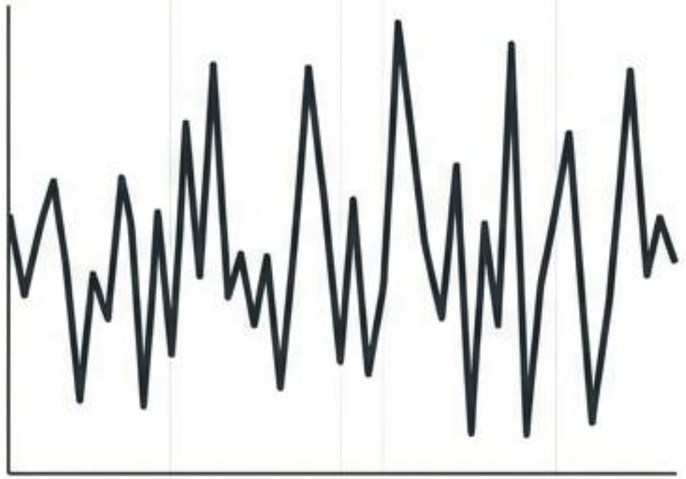
Latency

Milliseconds (ms)

User experience, Throughput.

Why Measurement is Hard

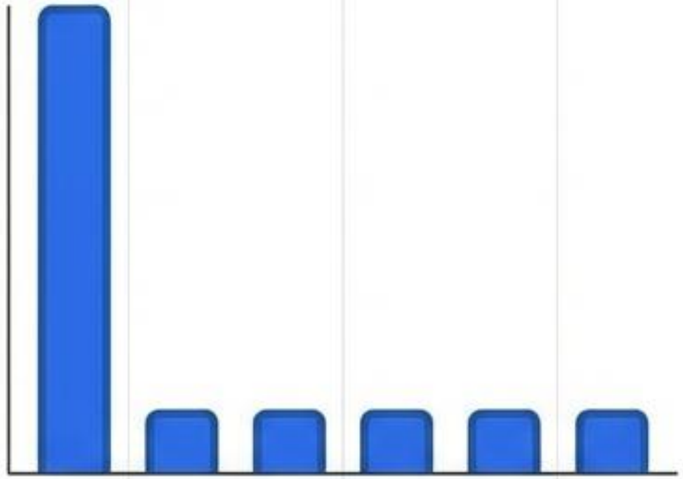
Noise & Variance



OS interrupts and background processes create jitter.

Solution: Statistical aggregation (Median over Mean).

Cold Starts



Caches are empty and Python is interpreting on the first run.

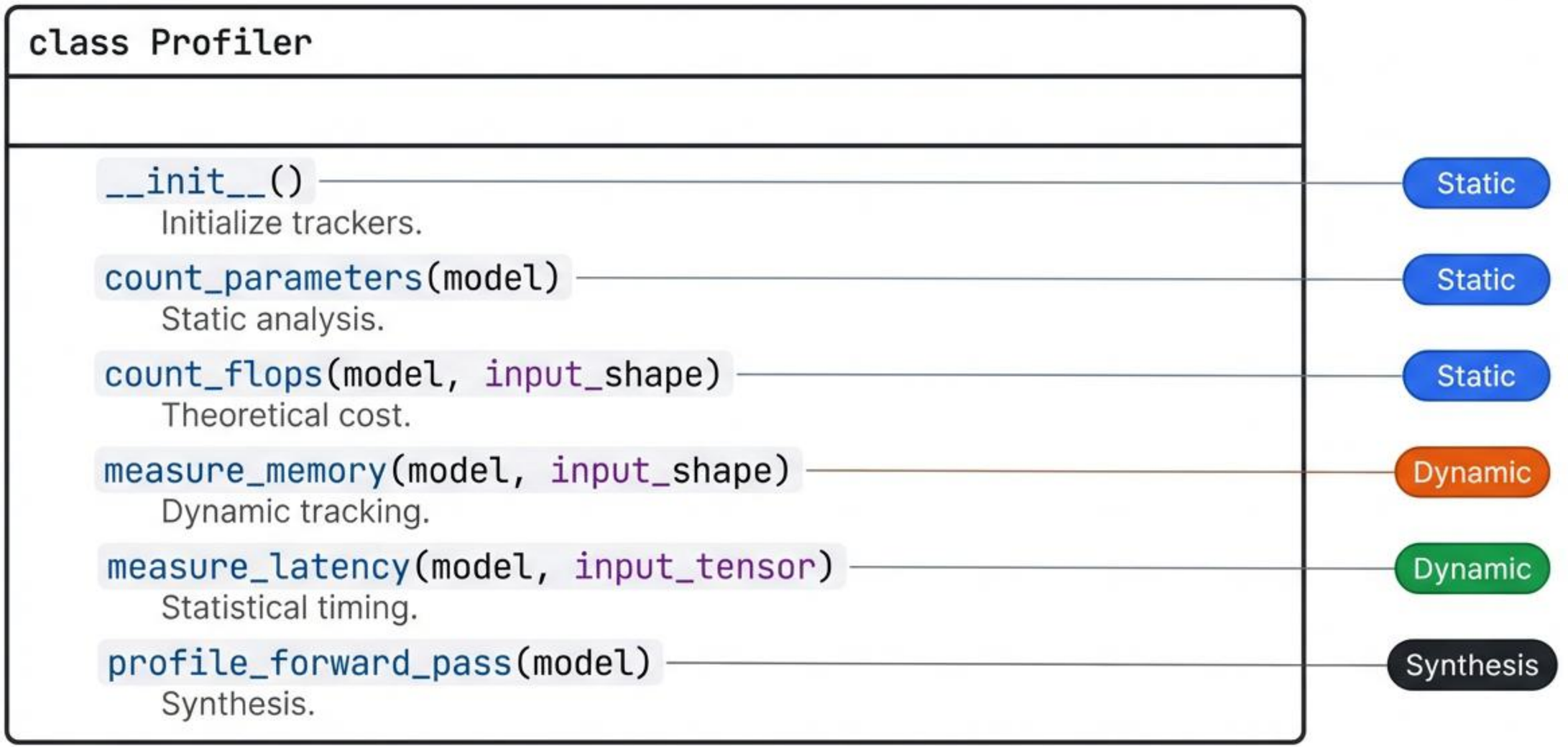
Solution: Warmup iterations.

Memory Complexity

Garbage collection obscures real usage.

Solution: tracemalloc snapshots.

The Profiler Class Architecture



tinytorch/perf/profiling.py

Concept 1: Counting Parameters

The Math

Parameters = Weights + Biases.

```
For Linear(in, out):
  Total = (in × out) + out
```

Every parameter is a float32 (4 bytes).
Count is invariant to batch size.

Example: GPT-2 Layer

Linear(768, 3072)

Weights:	768 × 3072 = 2,359,296
Bias:	3072
Total:	2,362,368 params
Memory:	~9.45 MB

Implementation: count_parameters

```python
def count_parameters(self, model) -> int:
    total_params = 0

    # 1. Handle models with standard PyTorch-style API
    if hasattr(model, 'parameters'):
        for param in model.parameters():
            total_params += param.data.size

    # 2. Handle single layers (Linear, Conv2d) directly
    elif hasattr(model, 'weight'):
        total_params += model.weight.data.size

        # Don't forget the bias!
        if hasattr(model, 'bias') and model.bias is not None:
            total_params += model.bias.data.size

    return total_params
```

We count elements here. Conversion to bytes happens later.

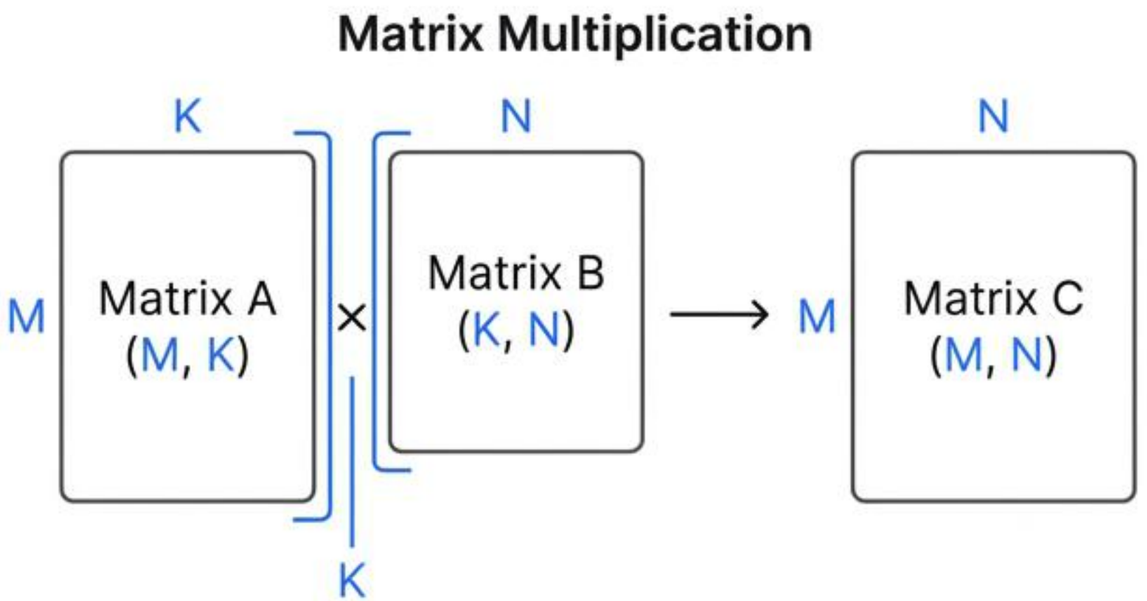
Concept 2: FLOP Counting

Floating Point Operations (Hardware Independent Work)

A FLOP is a single addition, subtraction, multiplication, or division.

Efficiency Formula:

$$\text{Efficiency} = \frac{\textbf{Measured FLOPs}}{\textbf{Hardware Peak FLOPs}}$$



Operations = 2 × M × N × K

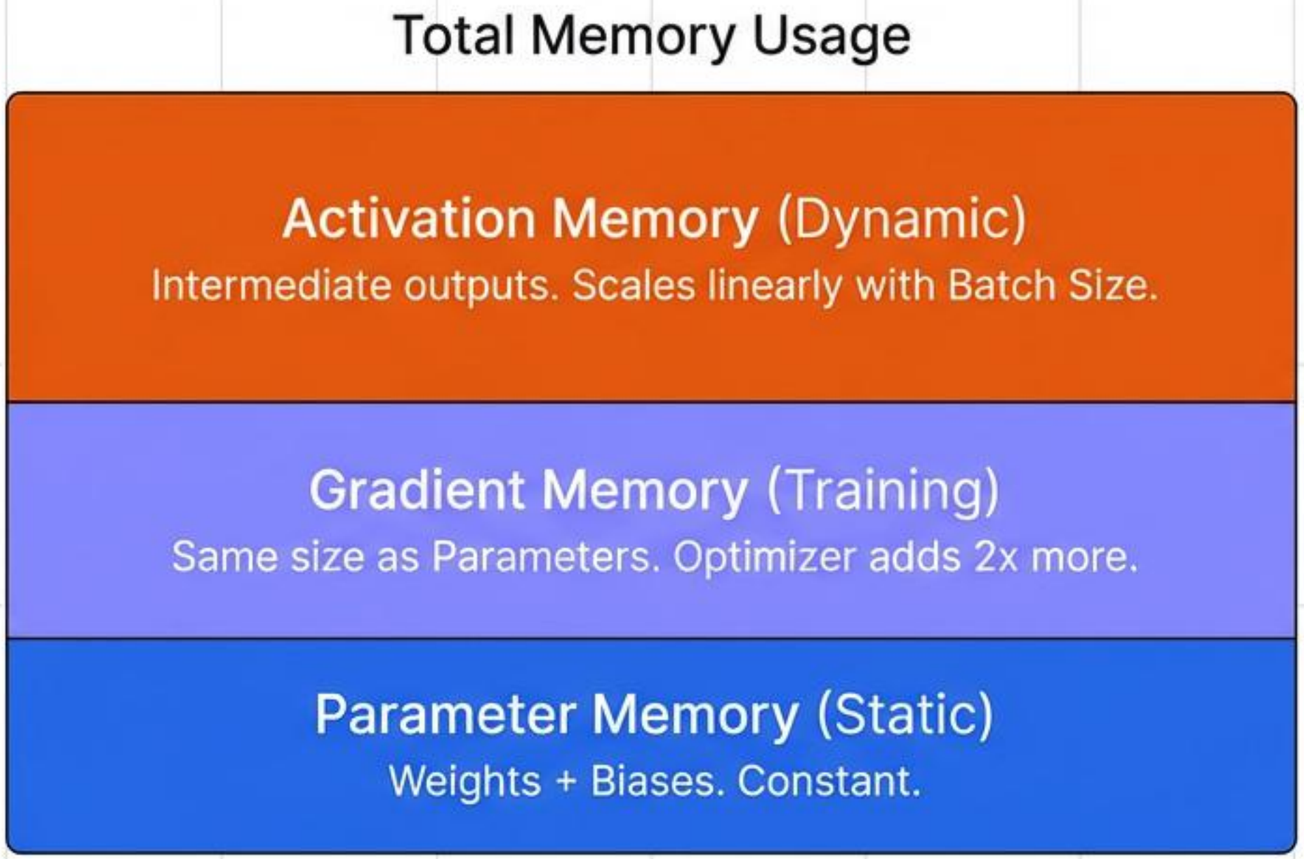
Why 2? Because each dot product requires 1 multiply + 1 add.

Implementation: count_flops

```python
def count_flops(self, model, input_shape: Tuple[int, ...]) -> int:
    total_flops = 0

    if model.__class__.__name__ == 'Linear':
        # Linear: (batch * in) @ (in * out)
        # FLOPs = batch_size * in_features * out_features * 2
        in_features = input_shape[-1]
        out_features = model.weight.shape[1]

        # Batch size is implicit in input_shape[0]
        batch_size = np.prod(input_shape[:-1])

        total_flops = batch_size * in_features * out_features * 2

    return total_flops
```

Crucial difference: FLOPs depend on input size (batch), Parameters do not.

Concept 3: The Three Types of Memory



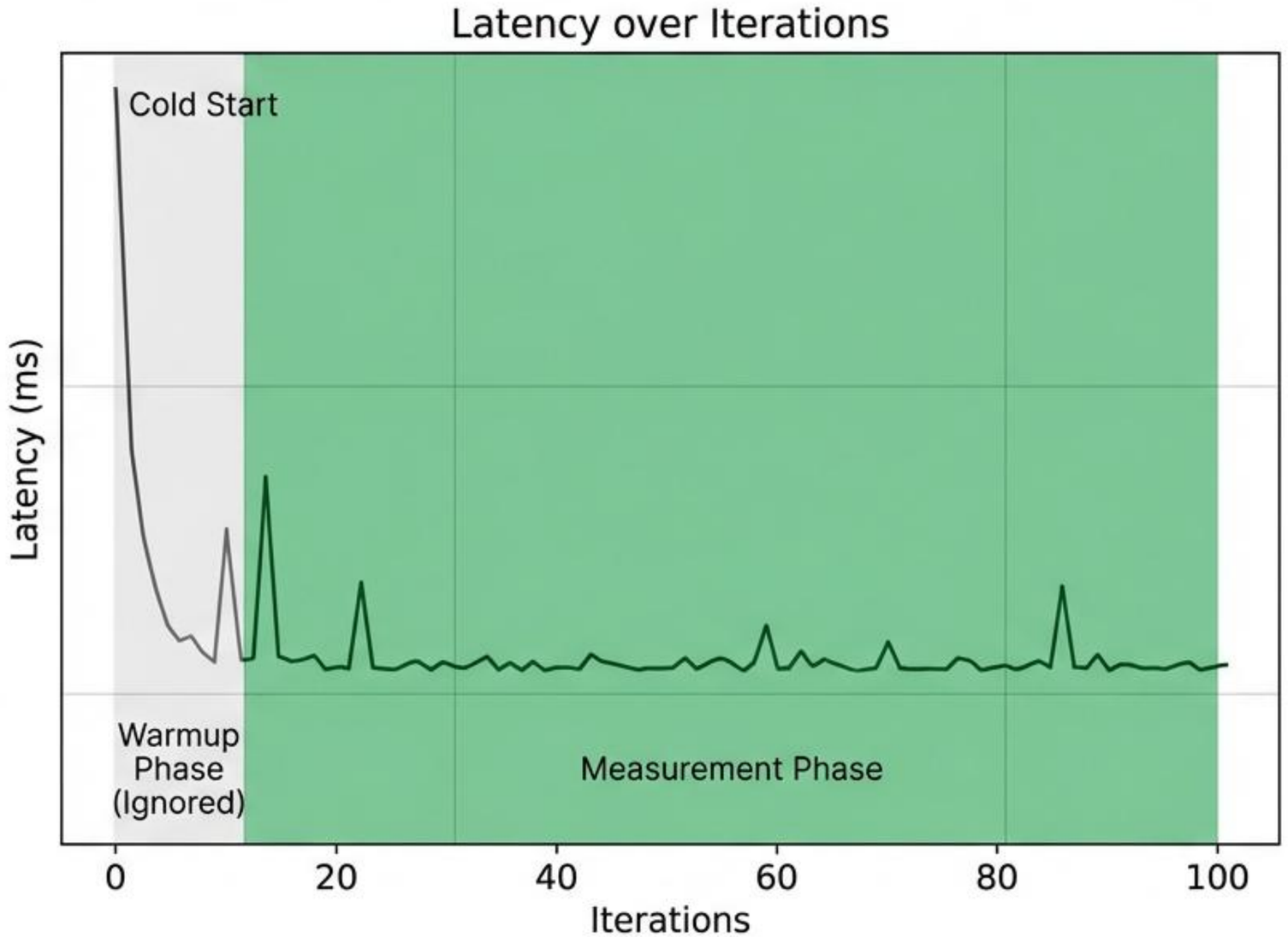
Insight: Most Out-Of-Memory (OOM) errors come from Activations (batch size too high) or Optimizer states, not the model weights themselves.

Implementation: measure_memory

```
import tracemalloc

def measure_memory(self, model, input_shape):
    tracemalloc.start() # 1. Start tracking system allocations

    # 2. Calculate theoretical parameter size
    param_count = self.count_parameters(model)
    param_mb = (param_count * 4) / (1024 * 1024)

    # 3. Create dummy input & Run Forward Pass
    dummy_input = Tensor(np.random.randn(*input_shape))
    _ = model.forward(dummy_input)

    # 4. Capture peak usage
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    peak_mb = peak / (1024 * 1024)
    return {'parameter_mb': param_mb, 'peak_mb': peak_mb}
```

Peak captures overhead + temporary buffers.

Concept 4: Latency & Statistics



Statistical Robustness:

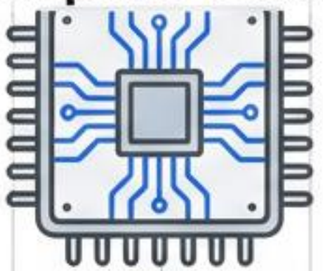
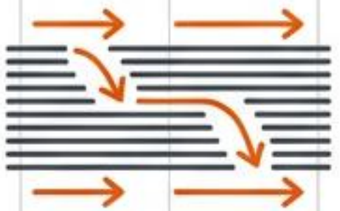
1. **Cold Starts**: Python imports and CPU wake-up skew the first run.
2. **Outliers**: OS interrupts cause spikes.
3. **Rule**: Always use the Median, never the Mean.

Implementation: measure_latency

```
def measure_latency(self, model, input_tensor,
                    warmup=10, iterations=100) -> float:
    # 1. Warmup Phase (Results ignored)
    for _ in range(warmup):
        _ = model.forward(input_tensor)

    # 2. Measurement Phase
    times = []
    for _ in range(iterations):
        start = time.perf_counter() # Use high-res timer
        _ = model.forward(input_tensor)
        end = time.perf_counter()
        times.append((end - start) * 1000) # Convert to ms

    # 3. Statistical Aggregation (Median)
    return float(np.median(times))
```

Note: time.perf_counter() is preferred over time.time() for nanosecond precision.

Bottleneck Identification

Compute-Bound	Memory-Bound
	
Sign: High Arithmetic Intensity (High GFLOP/s). ↑ G GFLOP/s	**Sign:** Low Arithmetic Intensity (Low GFLOP/s). ↓ G GFLOP/s
Constraint: Processor Speed. →	**Constraint:** Memory Bandwidth. ←
Examples: Large Matrix Multiplications, Convolutions	**Examples:** Element-wise ops, Activations, Attention
Fix: Faster hardware, Quantization. bit	**Fix:** Kernel Fusion, Caching, smaller batches.

The Roofline Concept: You are limited by whichever is slower: doing the math, or moving the data.

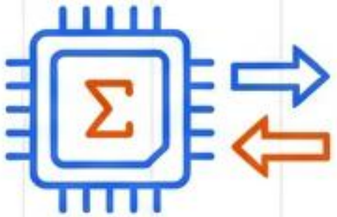
Implementation: profile_forward_pass

```python
def profile_forward_pass(self, model, input_tensor):
    # Gather raw data
    flops = self.count_flops(model, input_tensor.shape)
    latency_ms = self.measure_latency(model, input_tensor)

    # Calculate derived metrics
    latency_sec = latency_ms / 1000.0

    # GFLOP/s = (Total FLOPs / 1e9) / Time in Seconds
    gflops_per_sec = (flops / 1e9) / latency_sec

    return {
        'flops': flops,
        'latency_ms': latency_ms,
        'gflops_per_sec': gflops_per_sec,
        'bottleneck': 'memory' if gflops_per_sec < 10 else 'compute'
    }
```

Note: GFLOP/s calculation (Billion Floating Point Operations per Second) is crucial for determining compute vs. memory bounds. The threshold of 10 is illustrative.

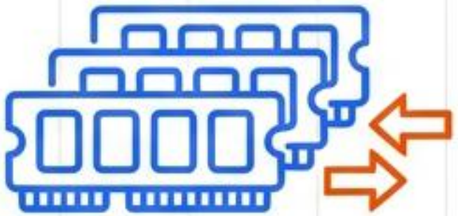
Profiling the Backward Pass (Training)

Why training is expensive



3x Compute

Training takes **~3x** the FLOPs of inference
(1 Forward + 2 Backward passes)



4x Memory

Training requires ~4x the Parameter Memory
(Params + Gradients + Optimizer States)

Adam Optimizer:
1. Parameters (1x)
2. Gradients (1x)
3. Momentum State (1x)
4. Velocity State (1x)

A 1GB model can easily require 4GB+ VRAM to train.

Implementation: profile_backward_pass

```
def profile_backward_pass(self, model, input_tensor):
    fwd = self.profile_forward_pass(model, input_tensor)

    # Heuristic Estimates
    backward_flops = fwd['flops'] * 2
    total_flops = fwd['flops'] + backward_flops

    # Optimizer Memory (Adam Estimate)
    grad_mem = fwd['parameter_memory_mb']
    optim_mem = grad_mem * 2

    return {
        'total_flops': total_flops,
        'total_memory_mb': fwd['peak_memory_mb'] + optim_mem,
        'training_latency_est': fwd['latency_ms'] * 3
    }
```

User Utilities: quick_profile

What you will use in assignments

```
>>> quick_profile(model, input_data)
```

```
🔬 Quick Profile Results:
   Parameters: 12,500
   FLOPs: 4,000,000
   Latency: 0.15 ms
   Memory: 2.10 MB
   Bottleneck: Memory Bound
   Efficiency: 5.0%
```

Production Context: TinyTorch vs. PyTorch

Feature	TinyTorch API	PyTorch API
Parameter Counting	`count_parameters()`	`sum(p.numel() for p in model.parameters())`
Latency	`time.perf_counter()` (CPU)	`torch.profiler` / `CUDA Events` (GPU)
Memory	`tracemalloc`	`torch.cuda.memory_allocated()`

The concepts (Warmup, FLOPs, Memory types) are identical; only the API changes.

Module Summary

Parameters: The static “weight” of the model.

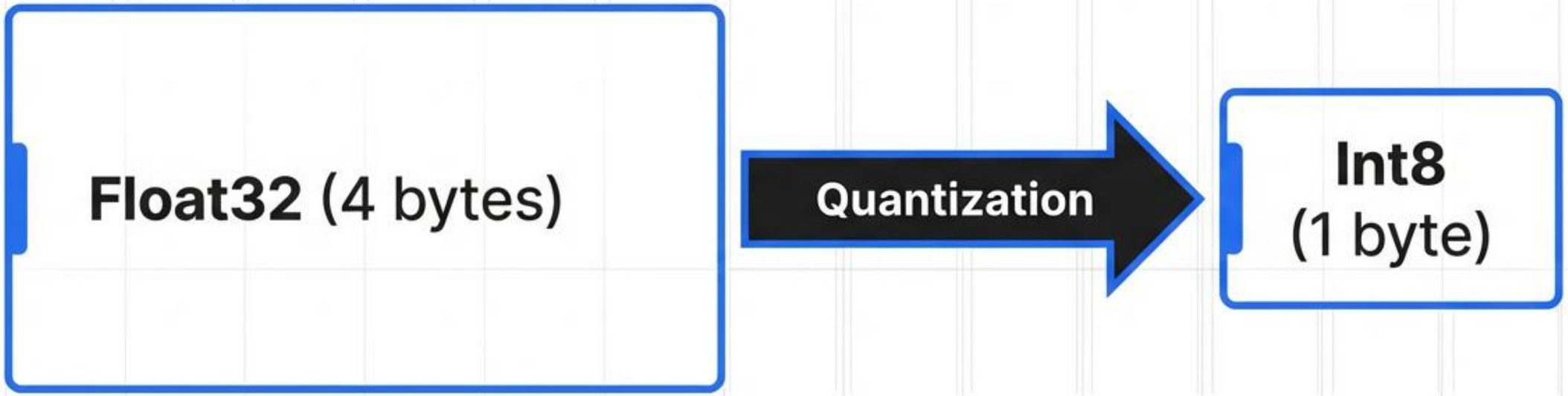
FLOPs: The hardware-independent “work” required.

Memory: Includes dynamic Activations (batch dependent) and static Parameters.

Latency: The real-world time, measured statistically (Median) after Warmup.

Bottleneck: The limiting factor—Compute speed vs. Memory bandwidth.

What's Next: Quantization (Module 15)



1. **Measure**: We now know our parameters take up X MB.
2. **Transform**: We will compress weights to int8.
3. **Validate**: We will use this Profiler to prove the 4x reduction.

See you in Module 15.