OPTIMIZATION TIER

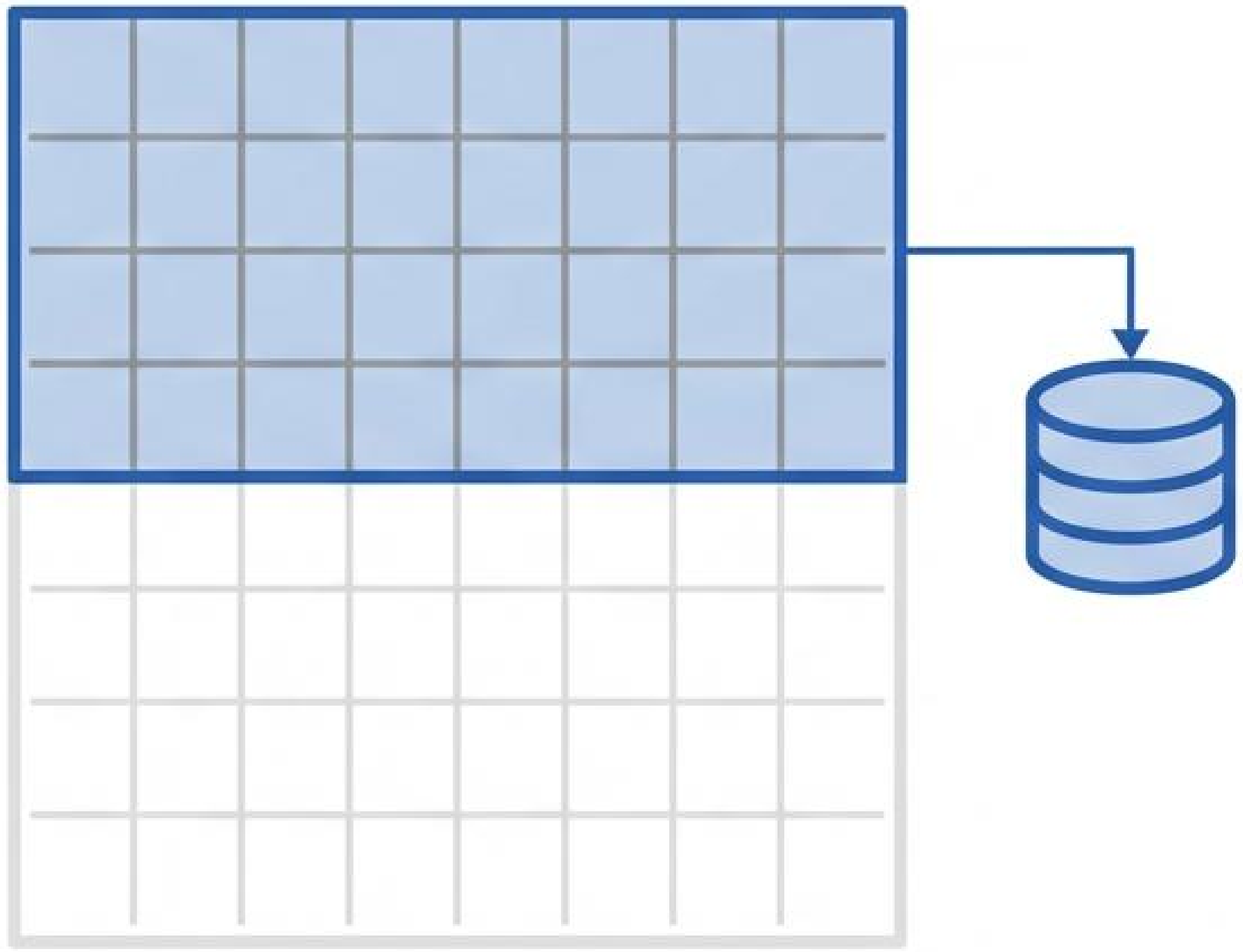
MODULE 18

KV Caching

From quadratic to linear generation complexity

Memoization & KV Caching

Optimization Tier: **Trading Memory for Latency**

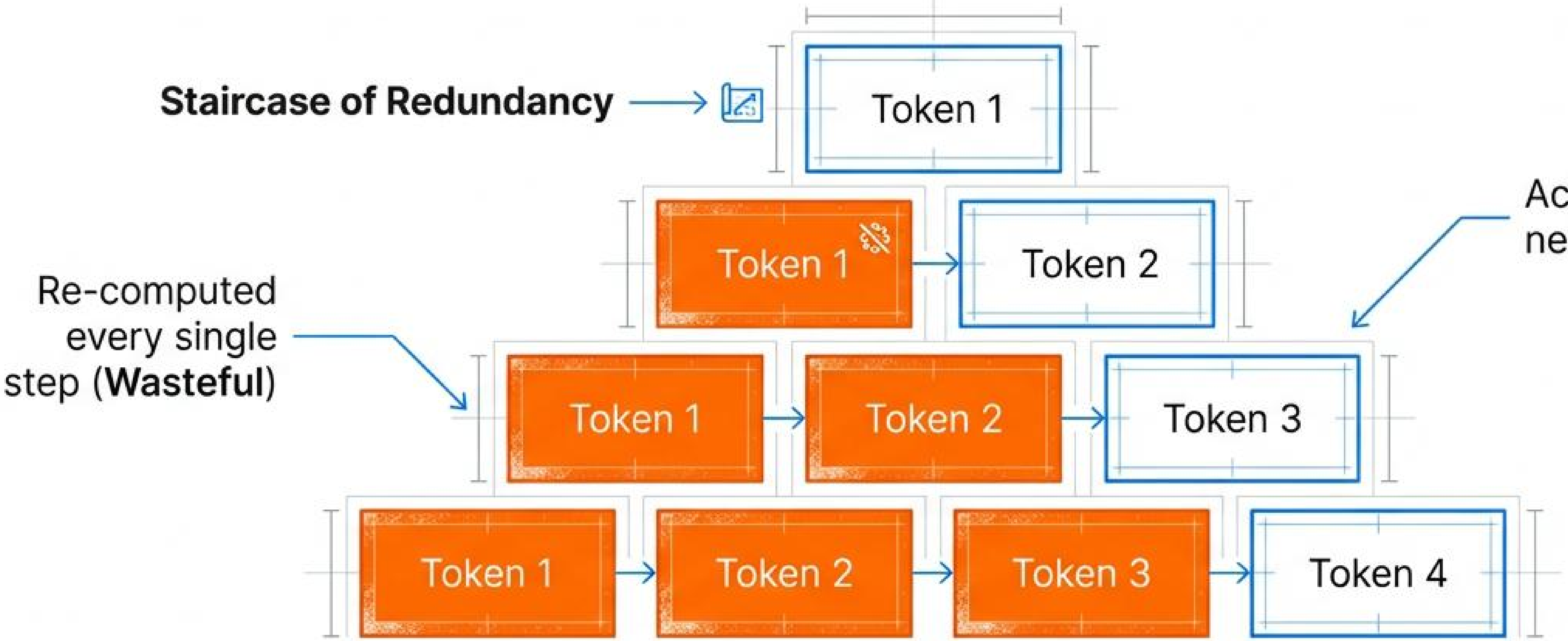


Goal: Transform autoregressive complexity from $O(n^2)$ to $O(n)$.

Implementation: tinytorch/perf/memoization.py

The Autoregressive Bottleneck

Why naive generation scales quadratically



The Cost: **The Cost:** $1 + 2 + 3 + \ldots + N = O(N^2)$

To generate the 100th token, the naive loop re-calculates the previous 99 tokens. ⚠

The Invariant

Identifying what changes vs. what stays the same

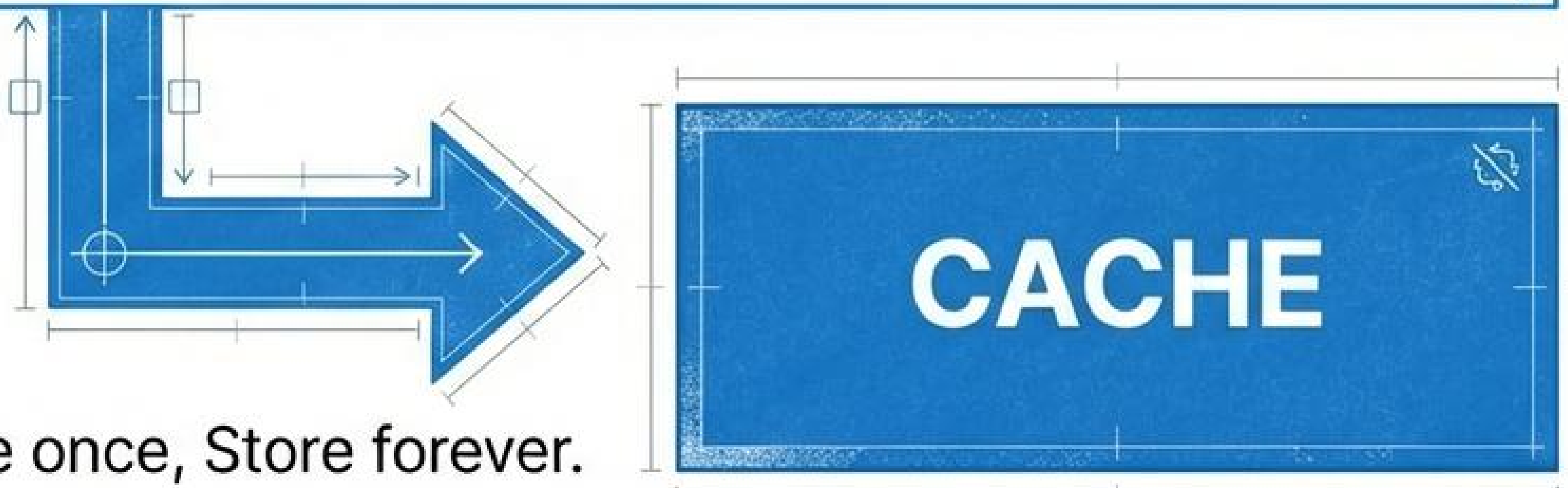
Query (Q)

Variable
Changes every step.

Derived from the NEW token.
Changes every step.

Key (K) & Value (V)

Invariant
Derived from PAST tokens. Once computed, never changes.

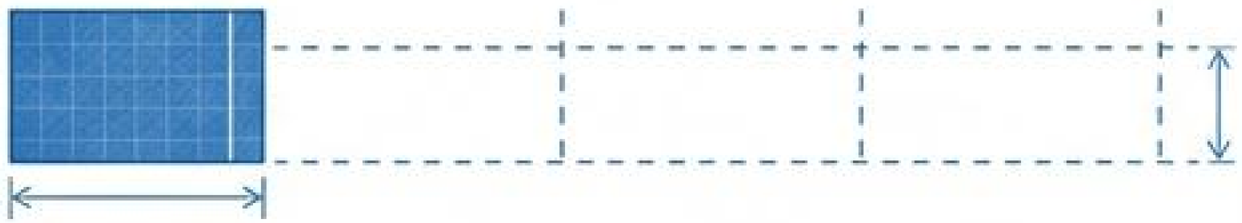


Strategy: Compute once, Store forever.

Systems Constraints

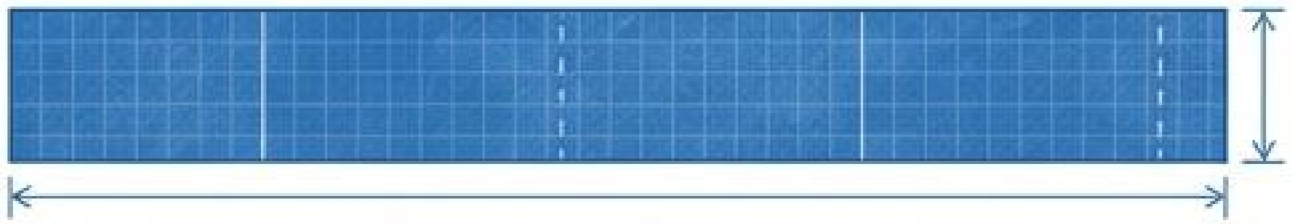
The Engineering Trade-off

The Cost (Memory)



GPT-2 Small Cache: ~75 MB

GPT-3 Cache: ~4 GB



Memory is cheap.

The Payoff (Compute)

Complexity: $O(n^2) \rightarrow O(n)$

Speedup: 10-15x

Latency is **expensive**.

Design Pressure: Cache operations (Read/Write) must be **O(1)**. Overhead kills speed.

TinyTorch Realization: The KVCache Class

A dedicated memory manager for inference

KVCache

tinytorch/perf/memoization.py

State

```
self.caches: List[Tuple[Tensor, Tensor]]
self.seq_pos: int // seq cache tensors
self.max_seq_len: int // max max_seq_len
```

Methods

`update(layer_idx, k, v)` → O(1) Write

`get(layer_idx)` → O(1) Read

`advance()` → Increment Ptr

Responsibility: Pre-allocate tensors and manage read/write pointers.

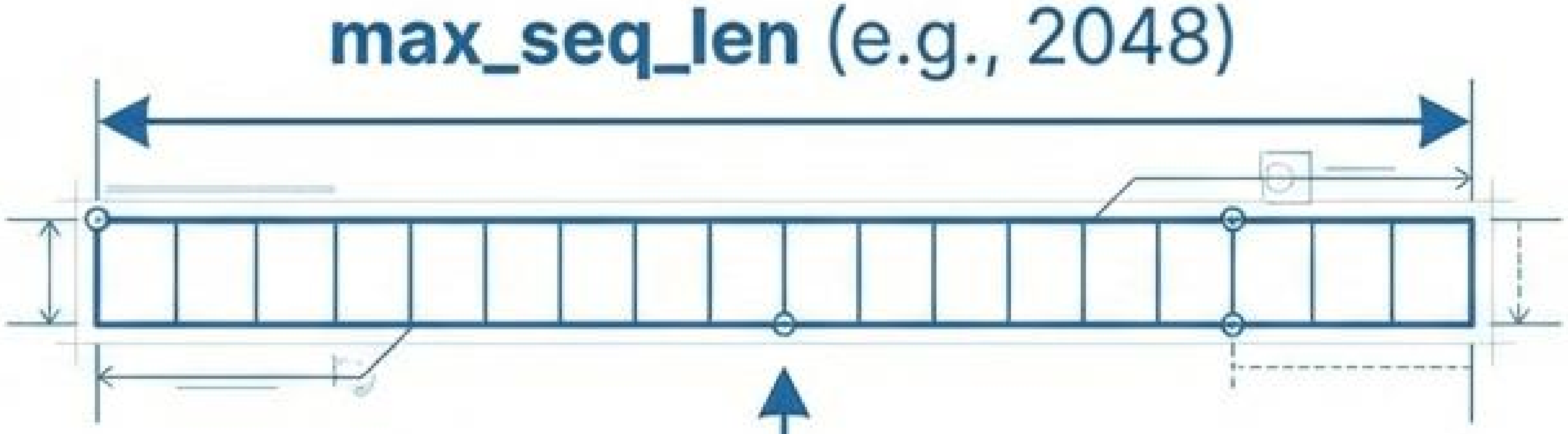
Constraint: Inference Only. No gradient tracking needed.

1. Initialization & Pre-allocation

Avoiding dynamic resizing overhead

```
def __init__(self, batch_size, max_seq_len,
    num_layers, ...):

    # Pre-allocate cache tensors with maximum size
    # Shape: (batch_size, num_heads, max_seq_len,
    key_cache = Tensor(np.zeros((batch_size,
        num_heads, max_seq_len, head_dim)))
    value_cache = Tensor(np.zeros((batch_size,
        num_heads, max_seq_len, head_dim)))
    self.caches.append((key_cache, value_cache))
```



Allocated immediately using **np.zeros**. **No resizing** during loop.

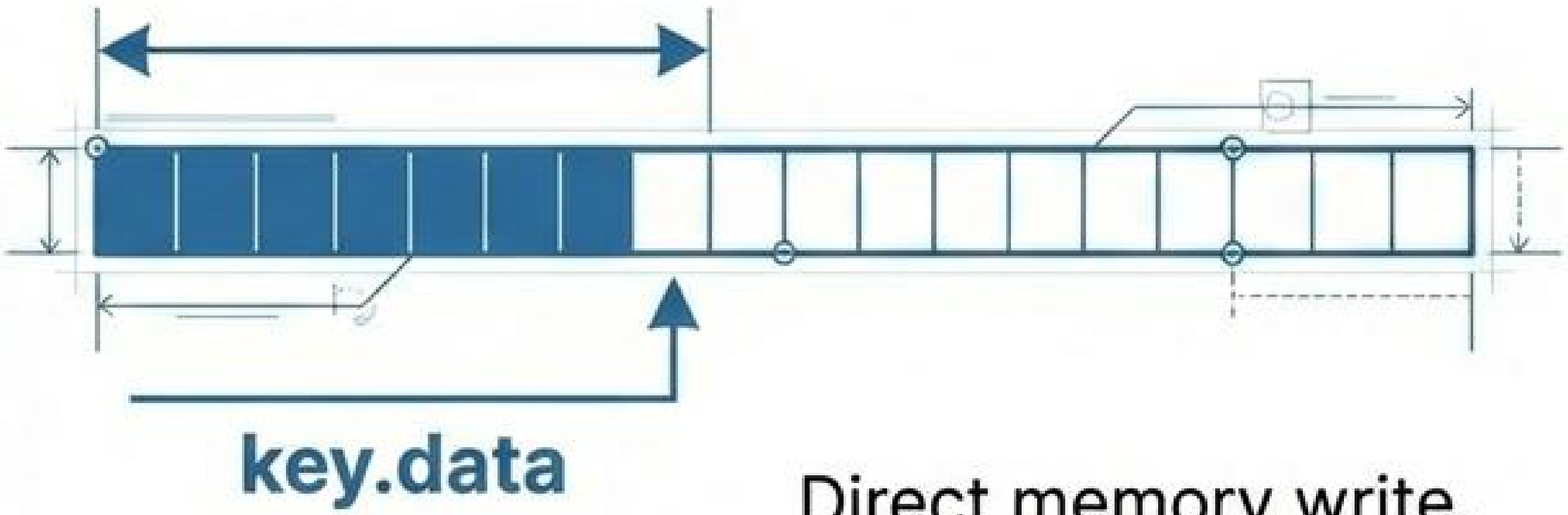
2. Efficient O(1) Updates

Writing directly to memory

```python
def update(self, layer_idx: int, key: Tensor,
    value: Tensor) -> None:

    # Get cache for this layer
    key_cache, value_cache = self.caches[layer_idx]

    # Update cache at current position (efficient O(1) write)
    # We use .data to avcid gradient tracking overhead
    key_cache.data[:, :, self.seq_pos:self.seq_pos+1, :]
        = key.data
    value_cache.data[:, :, self.seq_pos:self.seq_pos+1, :]
        = value.data
```



Direct memory write.
No Autograd.

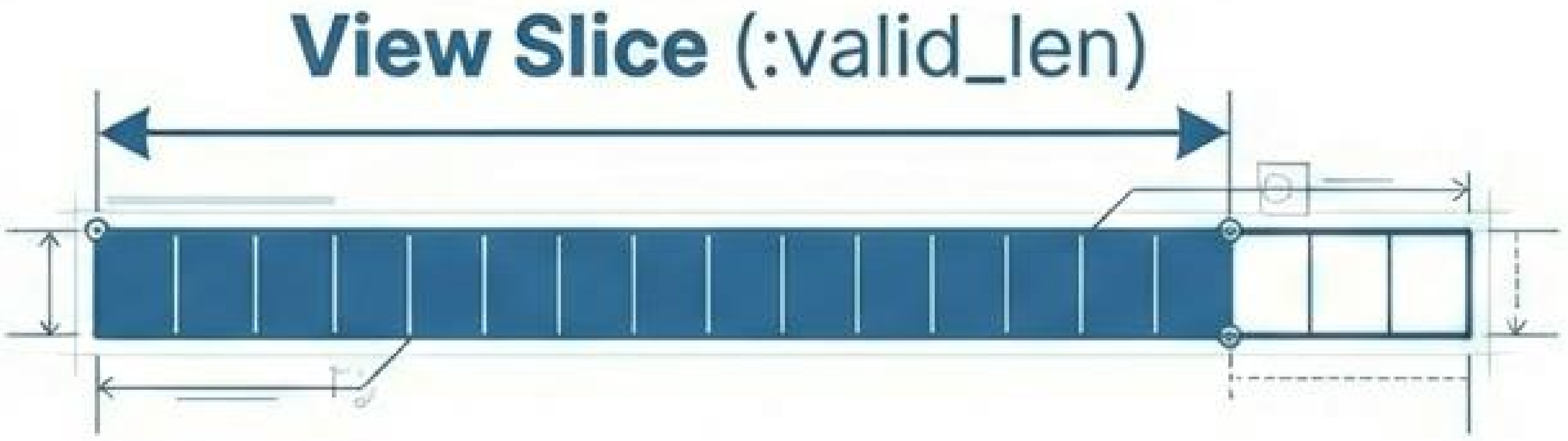
3. Zero-Copy Retrieval

Providing history to the model

```
def get(self, layer_idx: int) -> Tuple[Tensor,
    Tensor]:
    valid_len = self.seq_pos

    # Return only the valid portion
    cached_keys = Tensor(key_cache.data[:, :,
       :valid_len, :])
    cached_values = Tensor(value_cache.data[:, :,
       :valid_len, :])

    return cached_keys, cached_values
```



Returns a view, not a copy.
Instant retrieval.

Integration: Monkey Patching

Enhancing the model without rewriting it

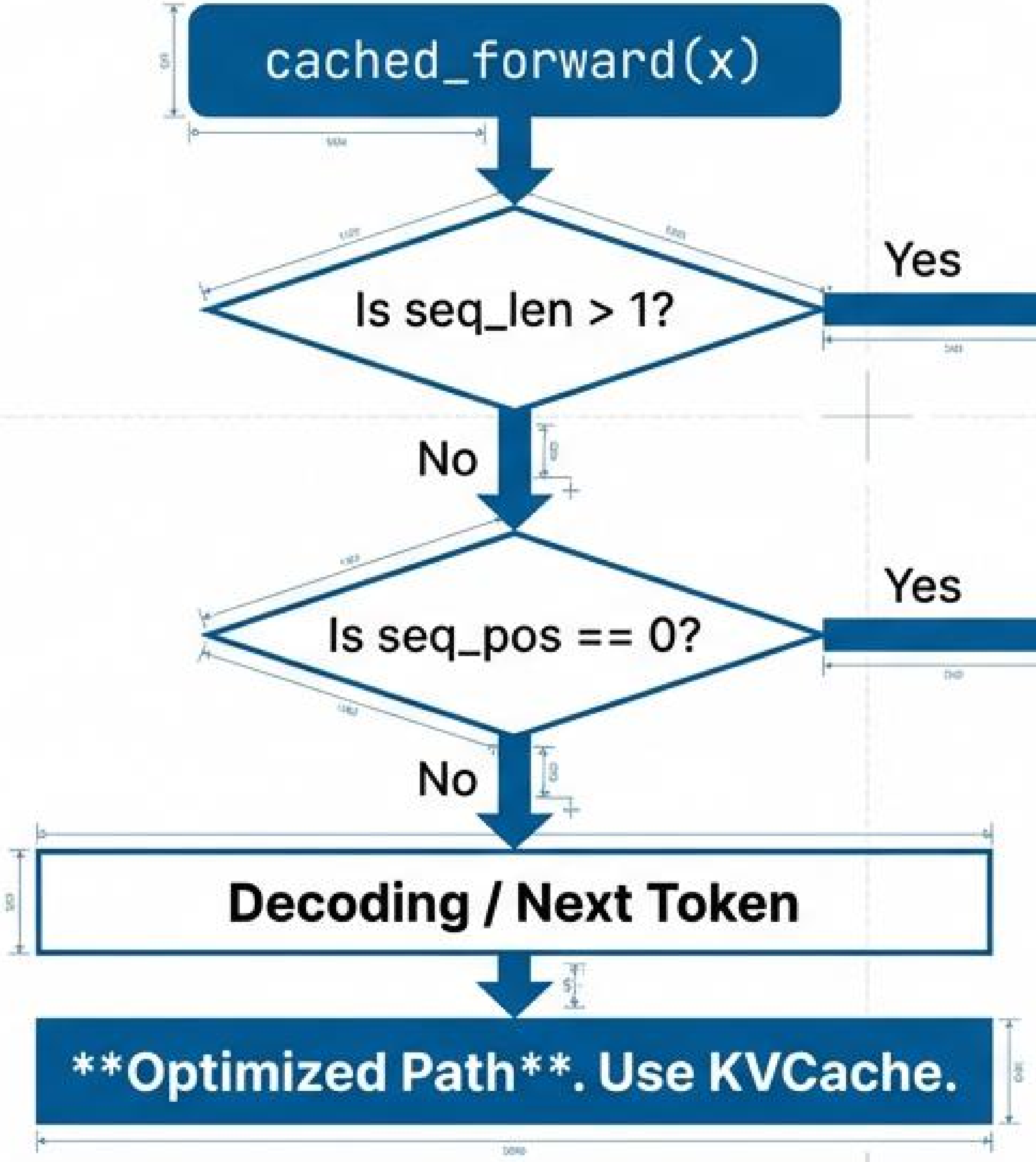
Wraps attention.forward

```python
def enable_kv_cache(model):
    # Create cache for this model
    cache = KVCache(...)
    model._kv_cache = cache

   for layer_idx, block in enumerate(model.blocks):
     # Patch this block's attention
     block.attention.forward = make_cached_forward(...)
```

4. The Dispatch Logic

Three execution paths inside the wrapper



The Optimized Path

Connecting the math to the cache

1. Compute New Only

```
# Projections for NEW token only
Q_new = attention.q_proj.forward(x)
K_new = attention.k_proj.forward(x)
```

2. Update Cache

```
# Store new K, V immediately
cache_obj.update(layer_idx, K_heads, V_heads)
```

3. Retrieve History

```
# Get everything we know so far
K_all, V_all = cache_obj.get(layer_idx)
```

4. Attend

```
# Attend Q_new to K_all
scores = np.matmul(Q_heads.data, K_transposed)
```

System Comparison: Trade-offs

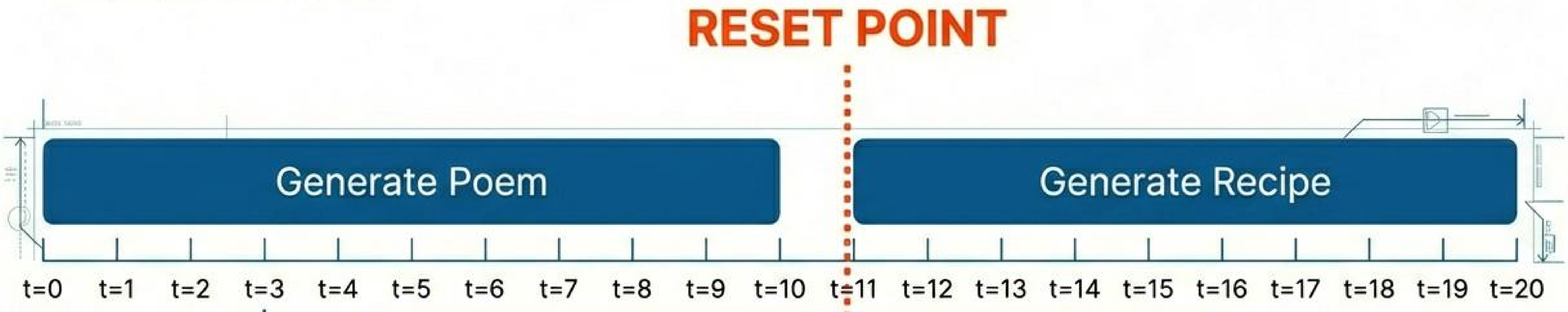
Memoization vs. Checkpointing

KV Caching (Module 18)	Gradient Checkpointing (Training)
• **Inference**	• **Training**
• Spend Memory → Save Compute	• Spend Compute → Save Memory
• **Latency Reduction** $(O(n^2) \rightarrow O(n))$	• **Capacity Increase** (Fit larger batch/model)

Memory and Compute are fungible resources.
The choice depends on the bottleneck.

Cache Lifecycle

Invalidation rules



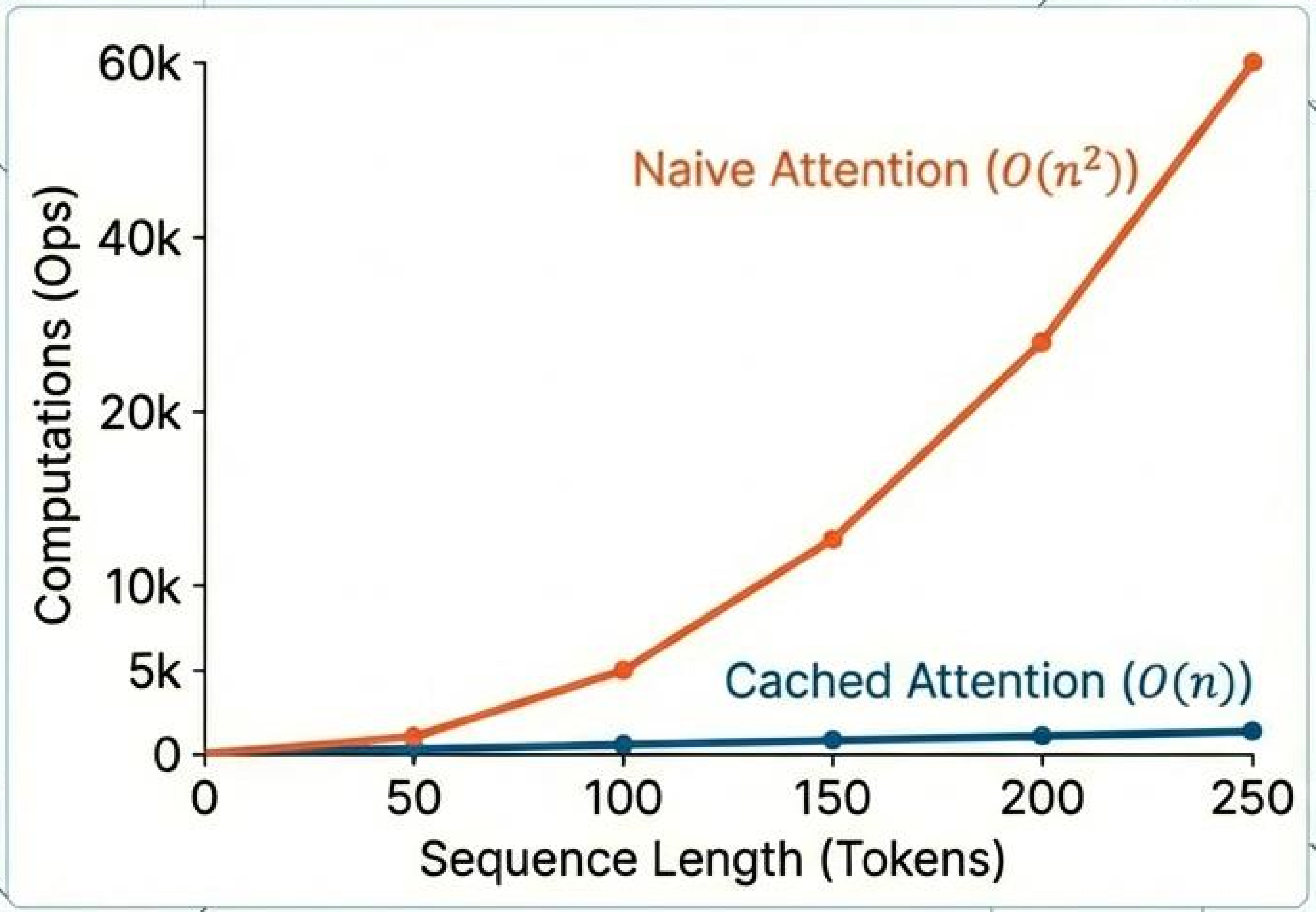
Rule: Cache is valid ONLY for the current sequence.
Must reset between prompts.

Python Code (Reset)

```
def reset(self) -> None:
    self.seq_pos = 0
    # Zero out caches for clean state
    for key_cache, value_cache in self.caches:
        key_cache.data.fill(0.0)
        value_cache.data.fill(0.0)
```

Performance Analysis

Quantifying the win

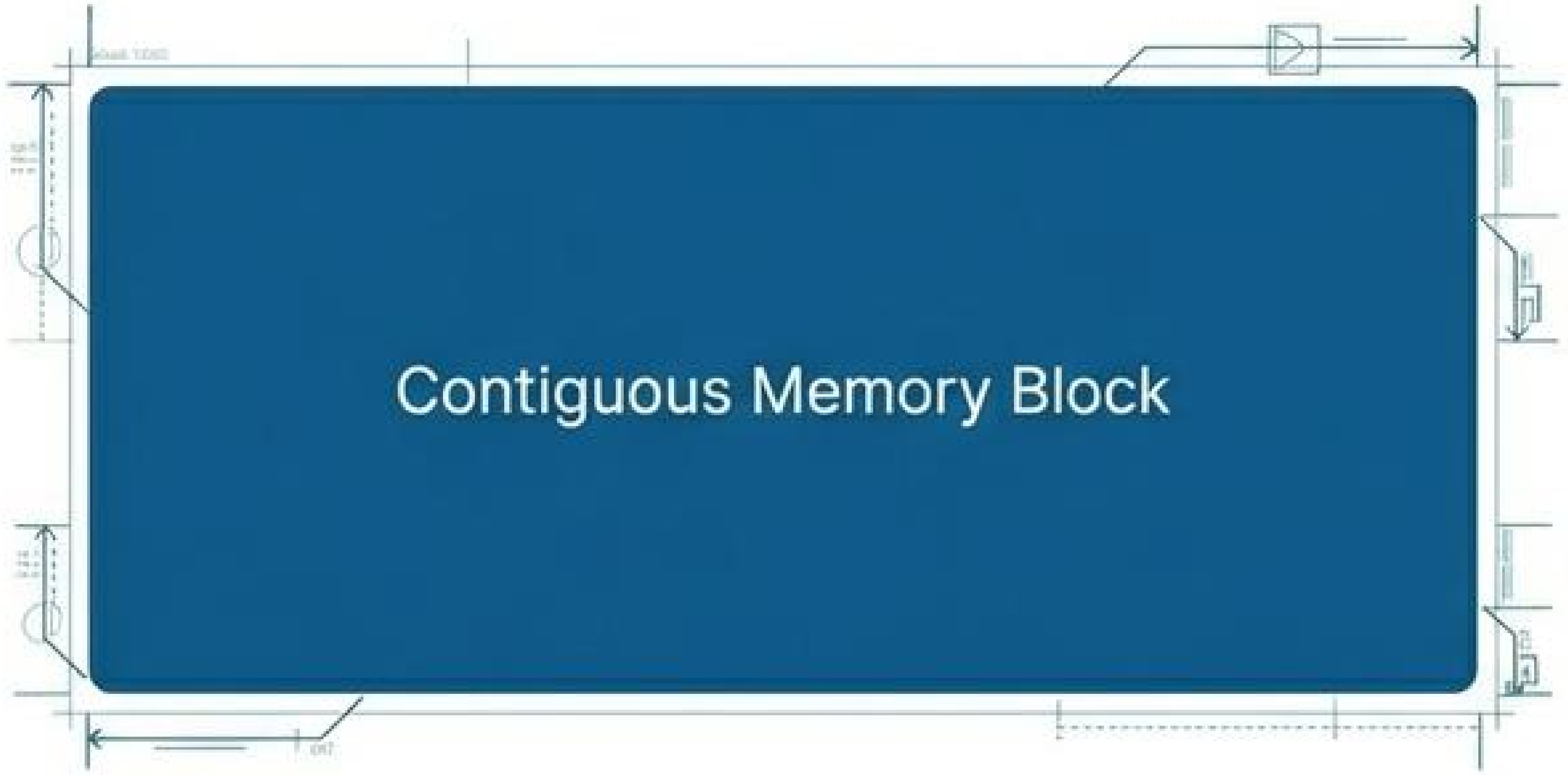


Scenario: 200 Tokens	
Naive Ops:	~20,100
Cached Ops:	200
Reduction:	**100x**

Production Context

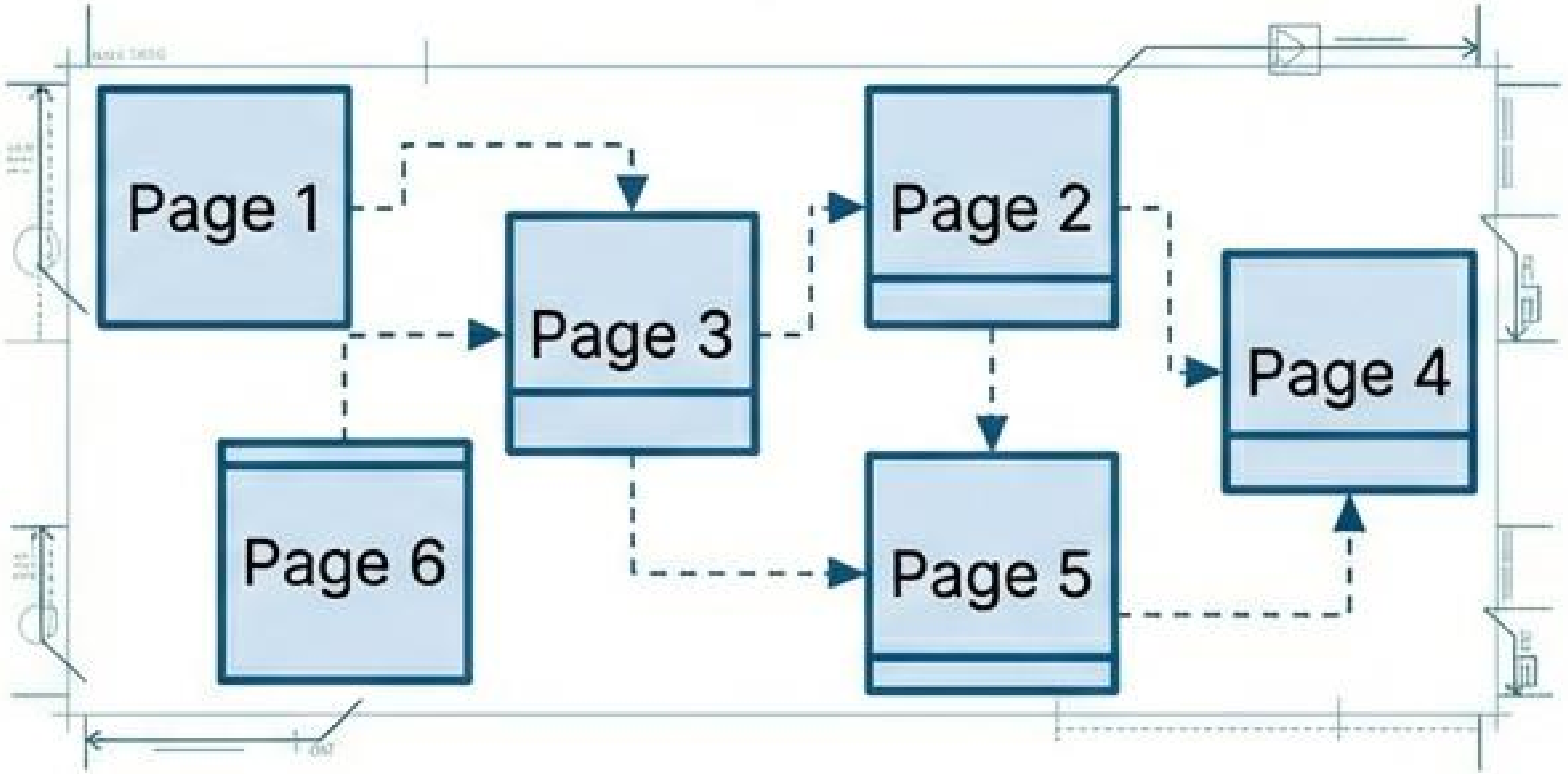
TinyTorch vs. vLLM / PyTorch

TinyTorch Implementation



Static Allocation. Contiguous memory. Simple.

Production (PagedAttention)



Virtual Memory Paging. Non-contiguous.
Dynamic Batching.

The Math is Identical. The Memory Management scales.

Common Implementation Pitfalls

Debugging Memoization

Saturation

Error:
`ValueError: Sequence position >= max_seq_len`

Fix:
Increase `max_seq_len` or implement eviction.

Stagnation

Symptom:
Model loops or repeats tokens.

Fix:
Forgot `cache.advance()` after update.

Shape Mismatch

Error:
Broadcasting failure.

Fix:
Input to update must be `(Batch, Heads, 1, Dim)`.

Synthesis: Module 18

From Theory to Engine

- Built `KVCache` class for memory management.
- Implemented $O(1)$ updates and retrievals.
- Integrated non-invasively via Monkey Patching.
- Achieved $O(n)$ inference complexity.

NEXT UP: MODULE 19

Benchmarking

We claimed a 15x speedup. Now we build the tools to prove it.