OPTIMIZATION TIER

MODULE 16

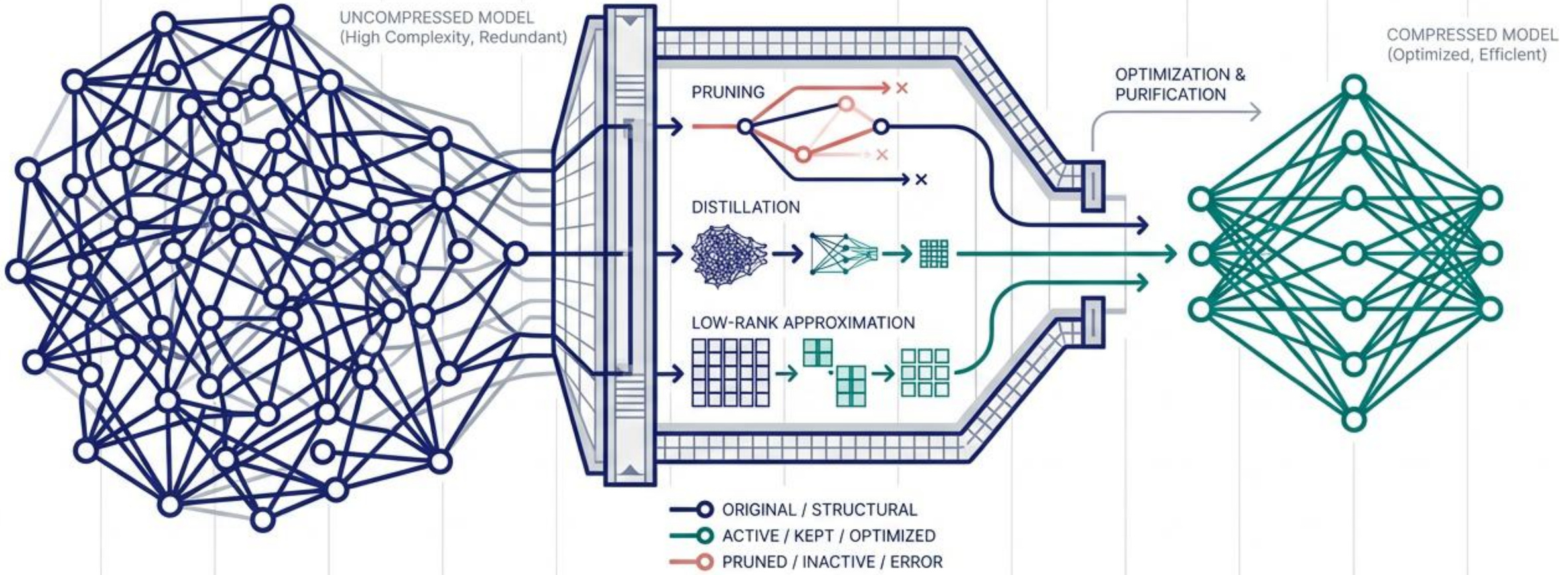
Model Compression

Bridging the deployment gap with efficient models

TinyTorch Module 16

Compression

Pruning, Distillation, and Low-Rank Approximation

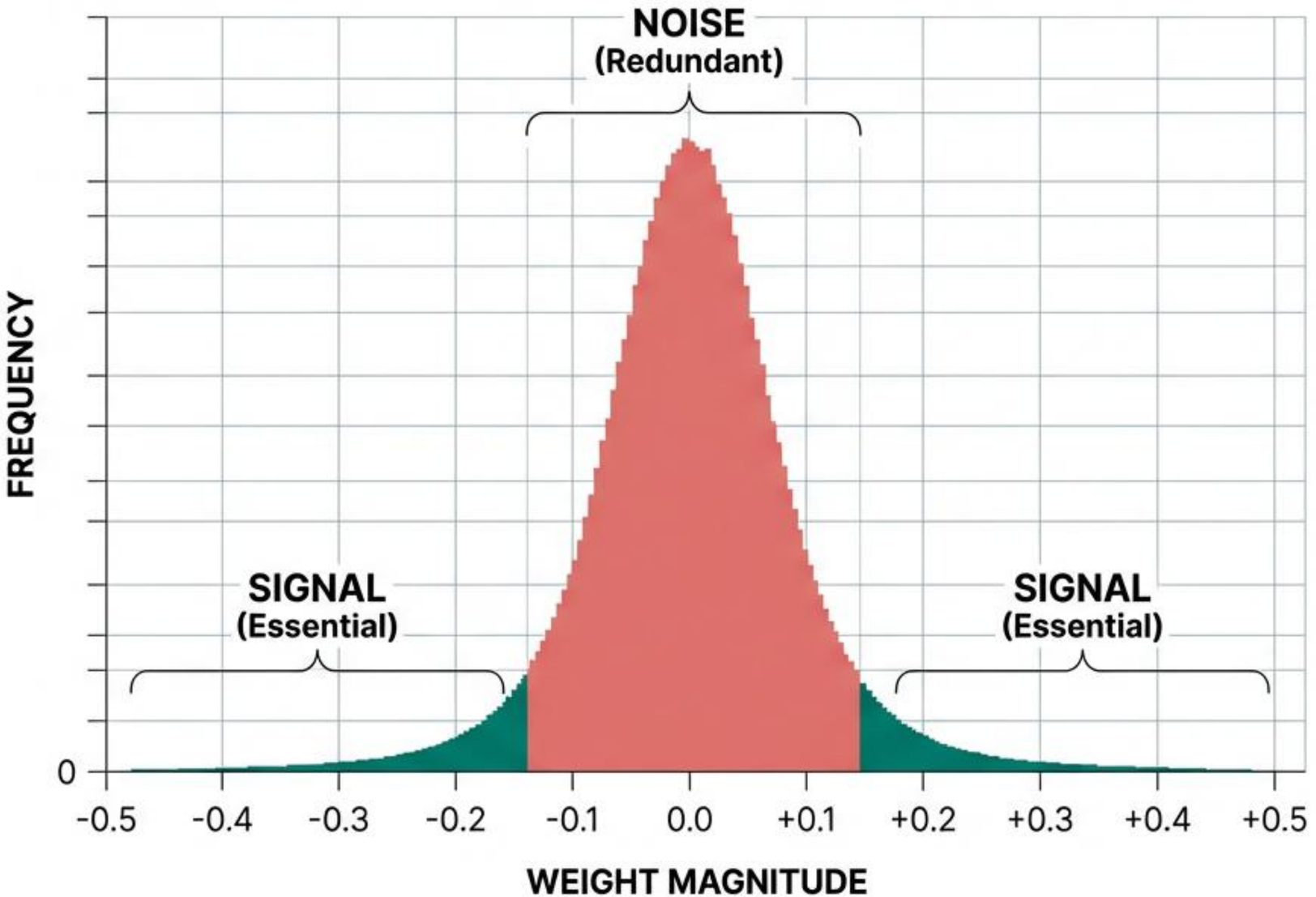


THE ABSTRACTION: SIGNAL VS. NOISE

The Lottery Ticket Hypothesis

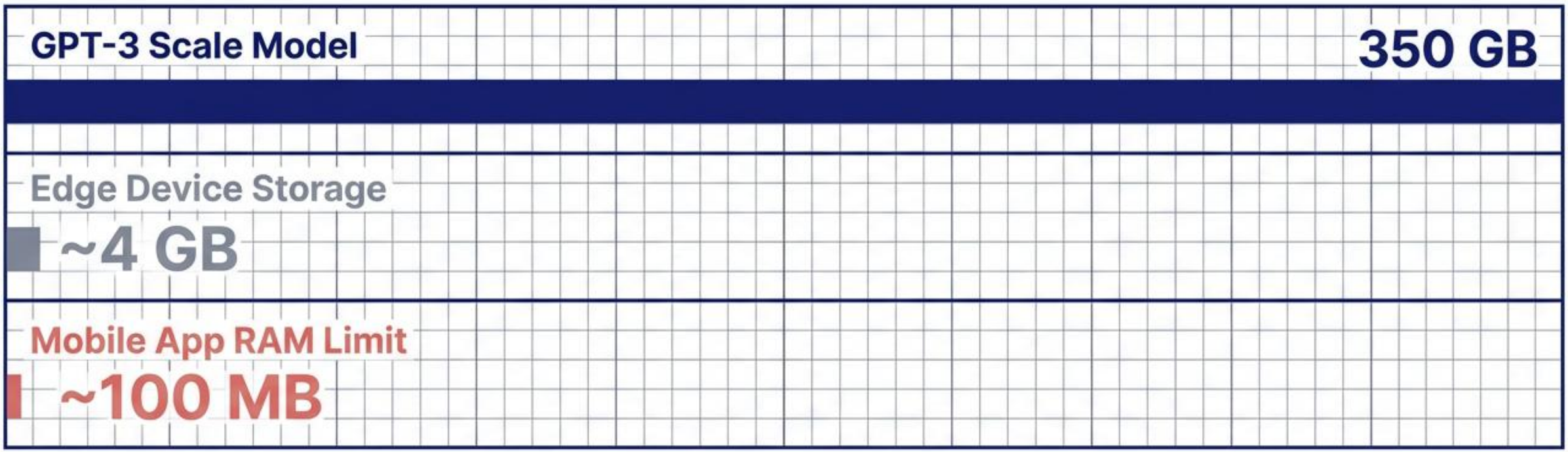
Research indicates that 50-90% of weights in a trained neural network contribute minimally to the final prediction.

These redundant weights cluster around zero, acting as **noise** rather than signal.



THE SYSTEMS CONSTRAINT: THE DEPLOYMENT GAP

You cannot deploy a 100GB model to a 100MB application.



Impact Metrics:

- Latency: Must be <100ms for real-time interaction.
- Energy: Direct impact on battery life.
- Bandwidth: App store download limits.

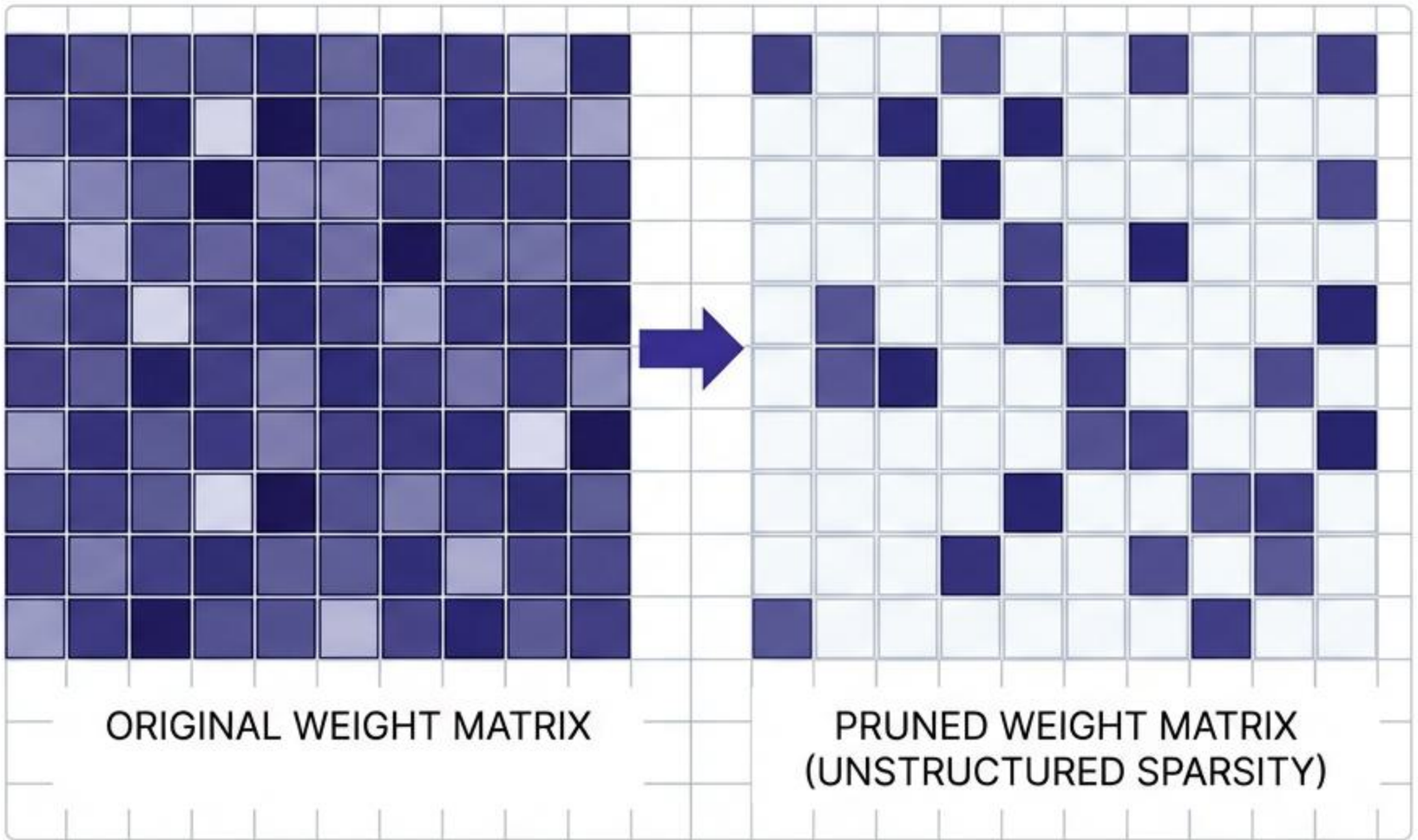
TECHNIQUE 1: MAGNITUDE PRUNING (UNSTRUCTURED)

Definition

Removing weights closer to zero than a calculated threshold.

The Invariant

```
if |w| < threshold:
    w = 0
```



Code: Measuring Sparsity

```python
def measure_sparsity(model) -> float:
    total_params = 0
    zero_params = 0

    for param in model.parameters():
        # Skip 1D biases (negligible size)
        if len(param.shape) > 1:
            total_params += param.size
            # Count exact zeros
            zero_params += np.sum(param.data == 0)

    return (zero_params / total_params) * 100.0
```

Key Logic

1. Iterate over all model parameters.
2. Identify weights that are exactly zero: `np.sum(param.data == 0)`.
3. Calculate the percentage: `zero_params / total_params`.

Note: We typically exclude bias vectors from pruning as they are small and crucial for stability.

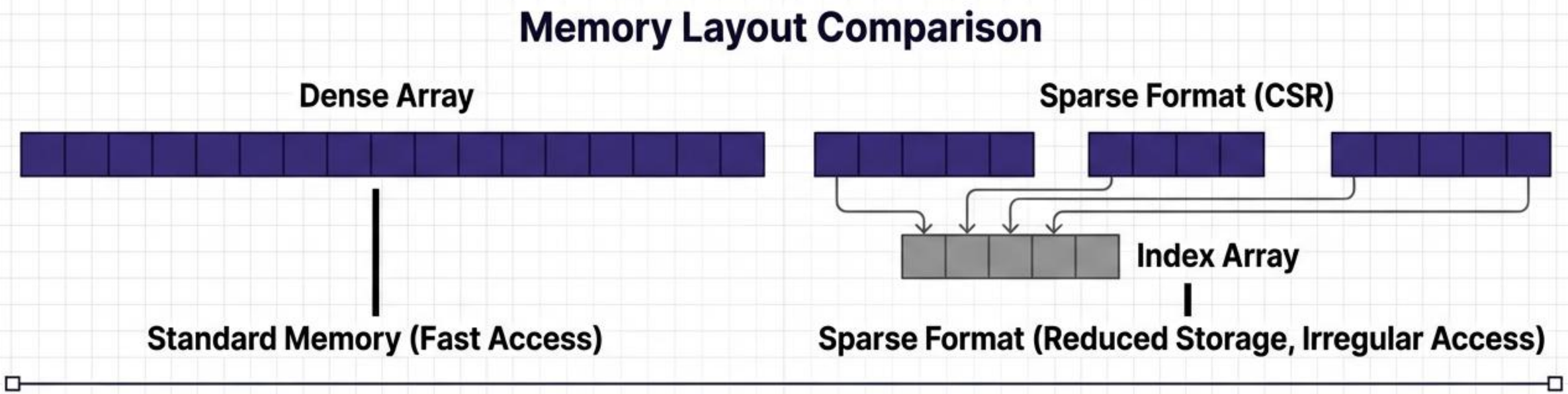
Code: Implementing Magnitude Pruning

```python
def magnitude_prune(model, sparsity=0.9):
    # 1. Collect all weights
    all_weights = []
    for param in model.parameters():
        if len(param.shape) > 1:
            all_weights.extend(param.data.flatten())

    # 2. Find global threshold
    magnitudes = np.abs(all_weights)
    # Find value larger than 90% of weights
    threshold = np.percentile(magnitudes, sparsity * 100)

    # 3. Apply mask in-place
    for param in weight_params:
        mask = np.abs(param.data) >= threshold
        param.data = param.data * mask
```

Global Pruning Strategy

1. **Pool Weights**: Flatten every weight in the network into one massive list.
2. **Determine Threshold**: Use `np.percentile` to find the cutoff value for the target sparsity (e.g., 90%).
3. **Apply Mask**: Create a boolean mask where `weight >= threshold`. Multiply original data by this mask to zero out small values.

Systems Insight: The Memory/Speed Paradox



The Reality:

- **Storage**: Reduced significantly using sparse formats (like CSR/CSC).
- **Compute**: **No speedup** on standard CPUs/GPUs.

Why? Processors rely on predictable memory patterns (cache lines) and vector instructions (SIMD). Randomly accessing non-zero elements breaks these optimizations.

Technique 2: Structured Pruning

Definition

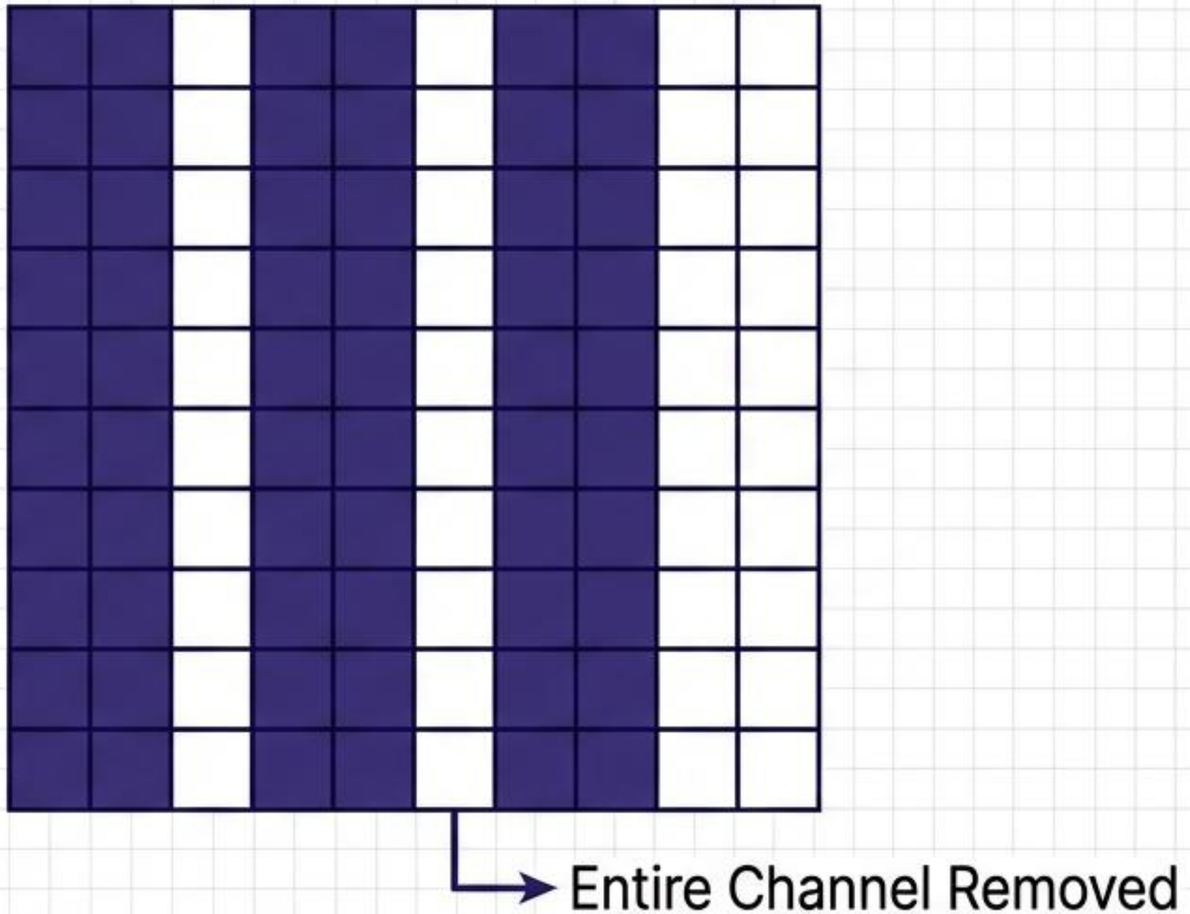
Removing entire structural elements (Channels, Rows, Columns) rather than individual weights.

Metric

L2 Norm of the channel: ||W[:,i]||

Strategy

Rank channels by ‘energy’ and remove the weakest ones completely.



Structured Sparsity (Channel Pruning)

Code: Implementing Structured Pruning

```
def structured_prune(model, prune_ratio=0.5):
    for layer in model.layers:
        if isinstance(layer, Linear):
            weight = layer.weight.data

            # Calculate L2 norm for each output channel (column)
            channel_norms = np.linalg.norm(weight, axis=0)

            # Find indices of smallest norms
            num_to_prune = int(weight.shape[1] * prune_ratio)
            prune_indices = np.argpartition(channel_norms, num_to_prune)[:num_to_prune]

            # Zero out entire columns
            weight[:, prune_indices] = 0

    return model
```

Calculate Importance (Energy)

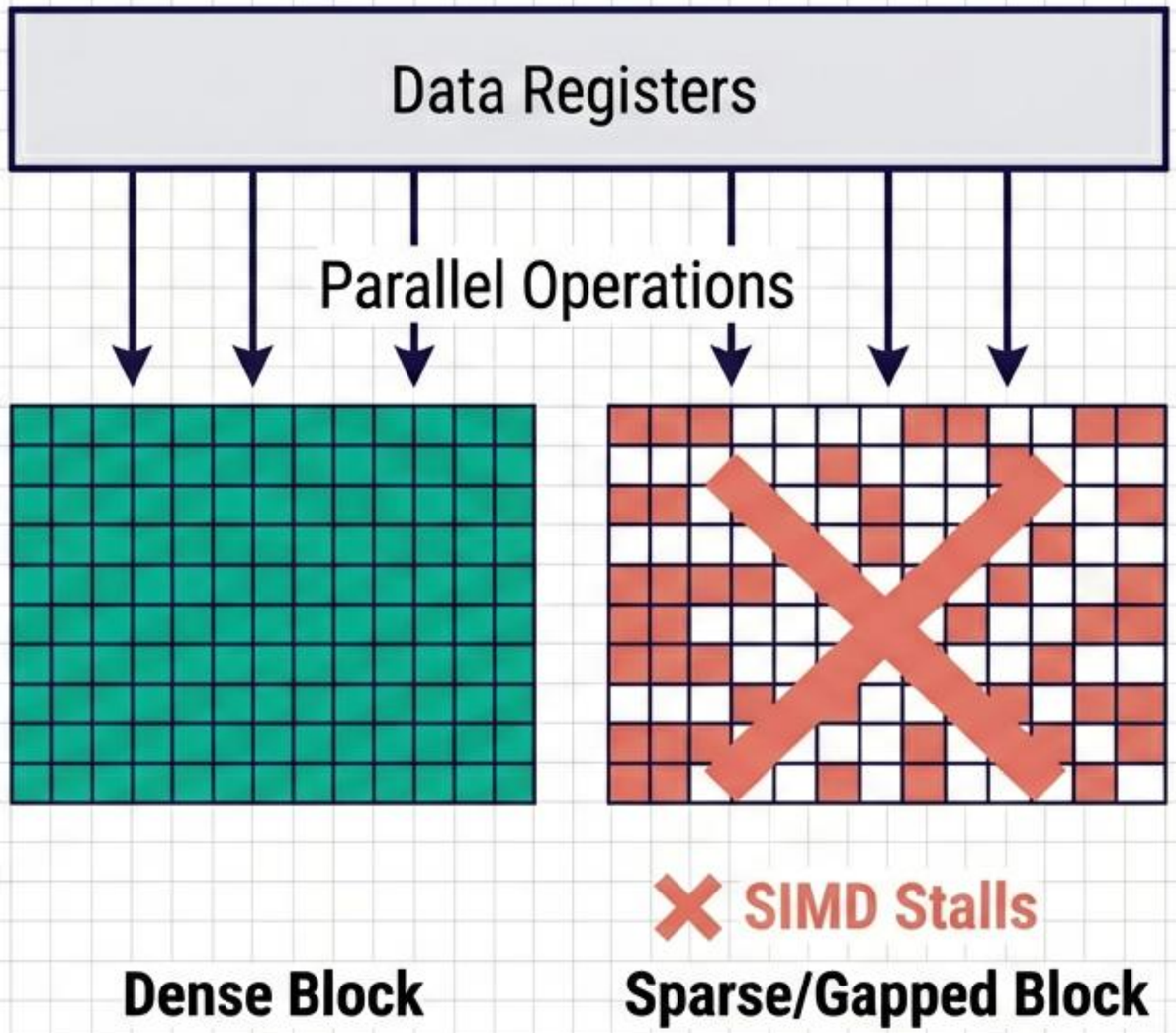
Efficiently find weakest channels

Kill entire channel

Systems Insight: Hardware Acceleration

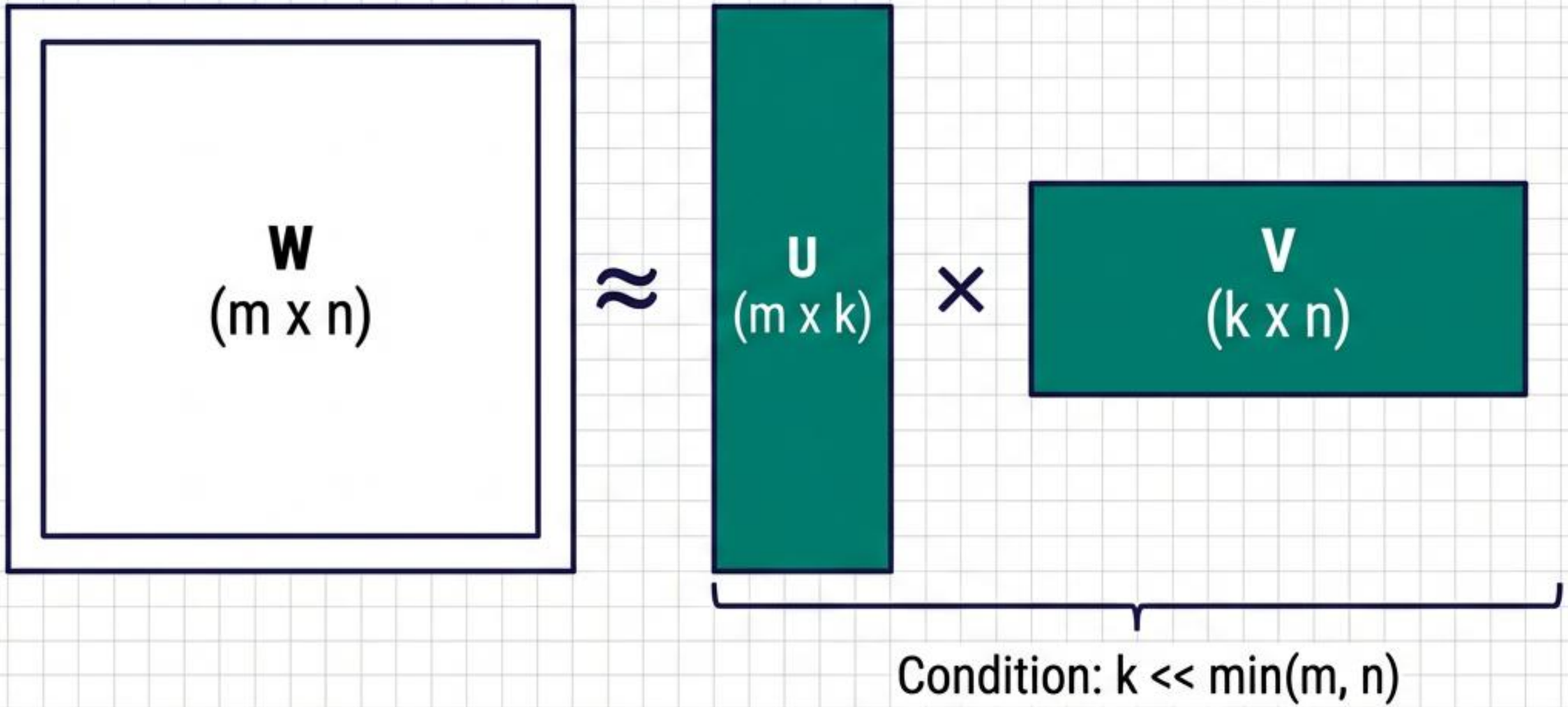
Why Structured Pruning Wins

- **Memory Coalescing:** Hardware reads large, contiguous chunks of memory.
- **Vectorization (SIMD):** Standard kernels can operate on the remaining dense blocks without branching or indexing overhead.
- **Trade-off:** Lower accuracy than magnitude pruning for the same sparsity level.



Technique 3: Low-Rank Approximation

Factoring a large matrix into two smaller matrices to compress information flow.

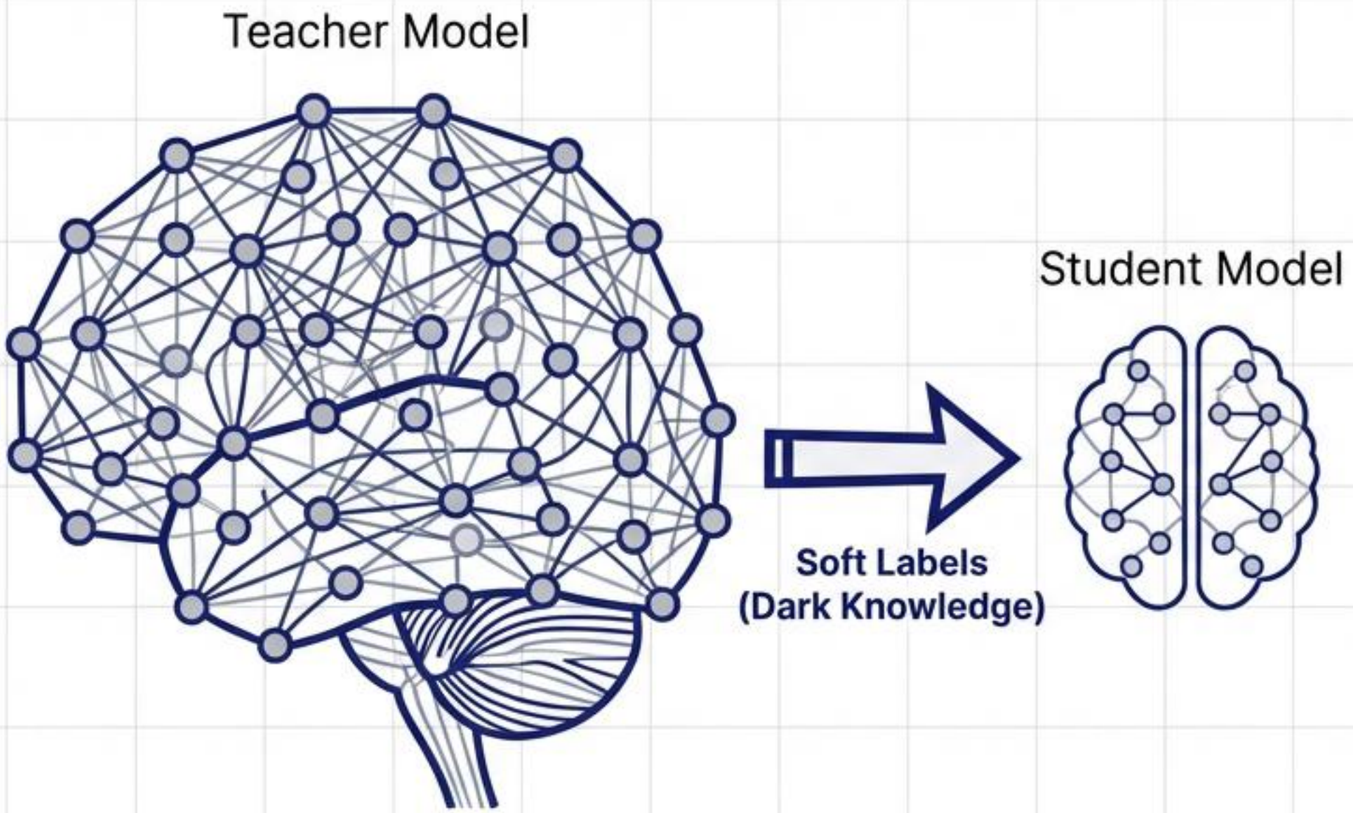


Code: SVD Implementation

```
def low_rank_approximate(weight_matrix, rank_ratio=0.5):
    # 1. Decompose via Singular Value Decomposition
    U, S, V = np.linalg.svd(weight_matrix, full_matrices=False)

    # 2. Determine truncation point (keep top k signals)
    target_rank = int(rank_ratio * min(weight_matrix.shape))

    # 3. Return truncated factors
    # We discard the noise (small singular values)
    return U[:, :target_rank], S[:target_rank], V[:target_rank, :]
```

Mechanism: We use NumPy's SVD to identify the principal components (singular values). We keep the top `rank_ratio` percent of these components—the strongest signals—and discard the rest.

Technique 4: Knowledge Distillation



Concept
Instead of shrinking the weights, we train a new, smaller architecture to mimic a larger, pre-trained one.

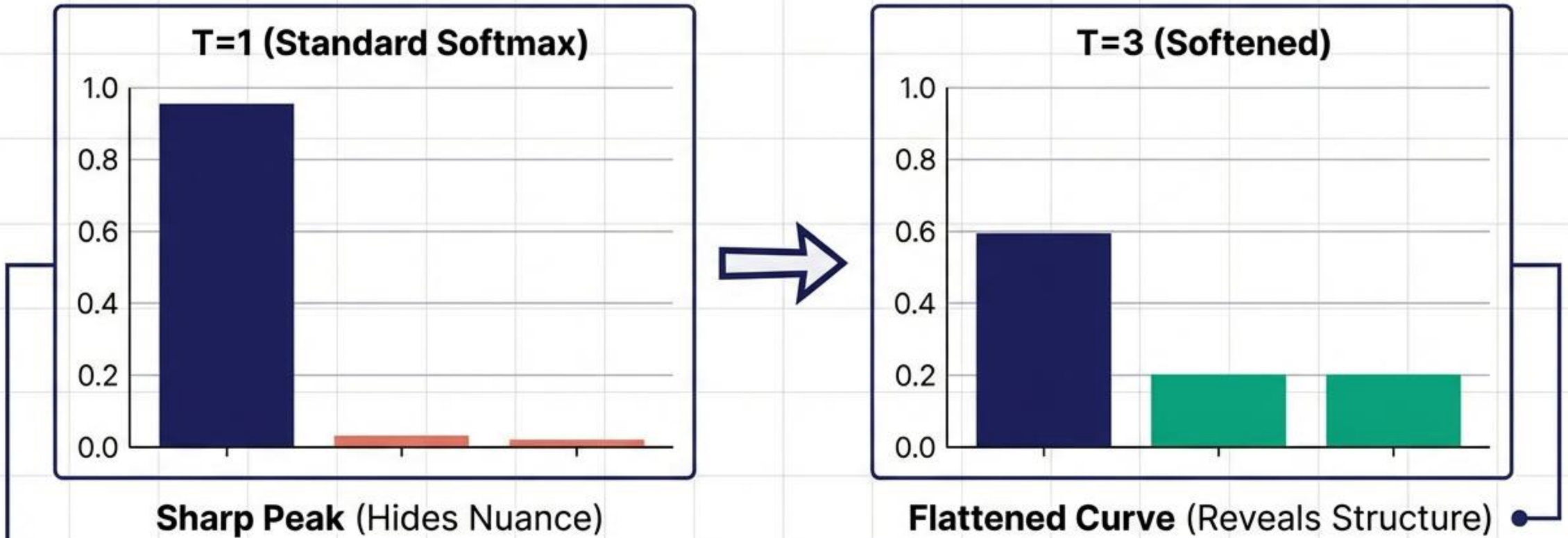
Key Insight
Soft labels (probabilities) contain more information than hard labels (0 or 1).

Example:

Hard Label: `[0, 1, 0]` (It is a Dog)
Soft Label: `[0.05, 0.90, 0.05]` (It is mostly Dog, but looks 5% like a Cat)

Concept: Temperature Scaling

Formula: Softmax(z_i / T)



Sharp Peak (Hides Nuance)

Flattened Curve (Reveals Structure)

T > 1 exposes the “dark knowledge”—the relationships between incorrect classes.

Code: The Distillation Class

```
class KnowledgeDistillation:
    def __init__(self, teacher_model, student_model, temperature=3.0, alpha=0.7):
        self.teacher = teacher_model
        self.student = student_model

        # Hyperparameters
        self.temperature = temperature # Softening factor (3-5 typical)
        self.alpha = alpha             # Balance factor (0.7 = 70% trust in teacher)
```

Configuration

- Teacher: Pre-trained, high accuracy.
- Student: Untrained, compact architecture.
- Alpha: Blending weight. `alpha=0.7` means the loss function cares 70% about matching the teacher and 30% about matching the ground truth.

Code: Distillation Loss

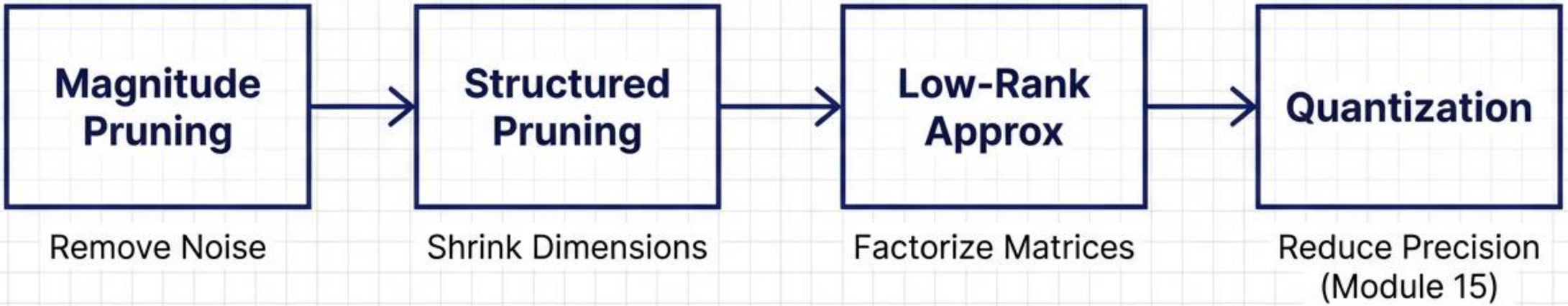
```
def distillation_loss(self, student_logits, teacher_logits, true_labels):
    # 1. Soft Targets (Teacher vs Student)
    # Scale logits by temperature before softmax
    student_soft = softmax(student_logits / self.temperature)
    teacher_soft = softmax(teacher_logits / self.temperature)
    soft_loss = KL_divergence(student_soft, teacher_soft)

    # 2. Hard Targets (Student vs Truth)
    # Standard CrossEntropy on raw logits
    hard_loss = CrossEntropy(softmax(student_logits), true_labels)

    # 3. Combine
    return self.alpha * soft_loss + (1 - self.alpha) * hard_loss
```

Soft Loss: Matches the reasoning.
Inter Regular

Hard Loss: Matches the answer.
Inter Regular

The Compression Pipeline

compress_model function



****Workflow:**

1. Profile the model to find redundancy.
2. Prune and approximate to reduce parameter count.
3. Distill to recover accuracy loss.
4. Quantize for final deployment size.

Systems Trade-offs Summary

Technique	Compression Ratio	Accuracy Impact	Hardware Speedup	Training Needed?
Magnitude Pruning	High (80-90%)	Low	None (Requires Sparse Libs)	No
Structured Pruning	Med (30-50%)	Med	High (SIMD Friendly)	No
Low-Rank Approx	Med (50%)	Med	Med	No
Distillation	High (10x)	Low	High (Smaller Model)	Yes

Map to TinyTorch

Implementation Location: `tinytorch/perf/compression.py`

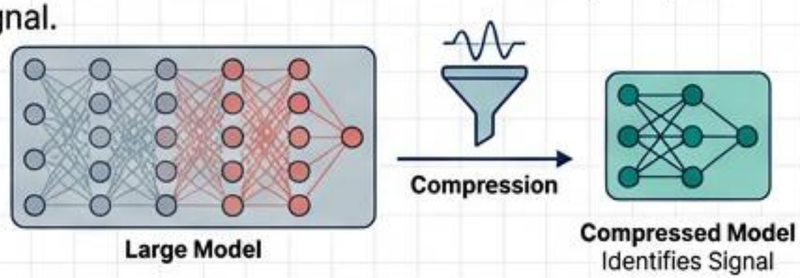
- [ ] `measure_sparsity(model)`
- [ ] `magnitude_prune(model, sparsity)`
- [ ] `structured_prune(model, ratio)`
- [ ] `low_rank_approximate(matrix, rank)`
- [ ] `class KnowledgeDistillation`

Your assignment is to implement these five core components.

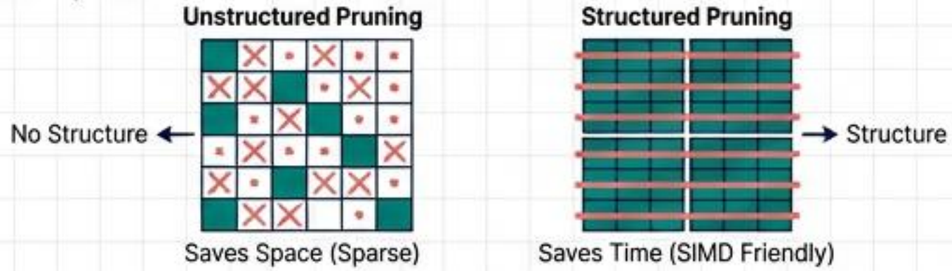
Module 16 Takeaways

Takeaways

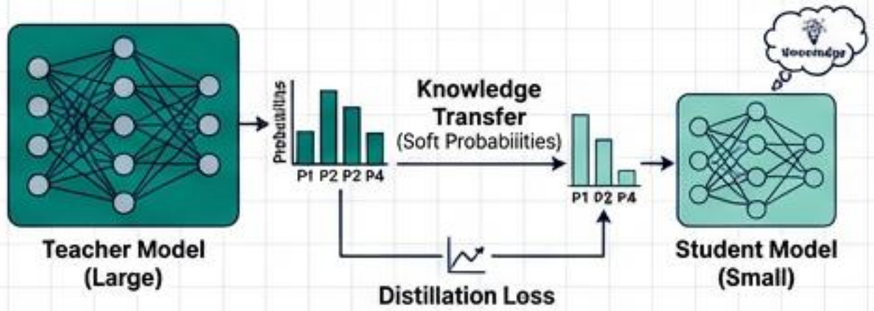
1. **"Signal vs Noise"**: Models are 90% redundant; compression identifies the signal.



2. **"Structure Matters"**: Unstructured pruning saves space; Structured pruning saves time.



3. **"Distillation"**: Transfer intelligence via soft probabilities, not just weights.



Next Steps

Next Up: Module 17 - Acceleration

Now that the model is small, how do we make the math run at light speed?

- Vectorization
- Kernel Fusion
- Memory Caching