ARCHITECTURE TIER

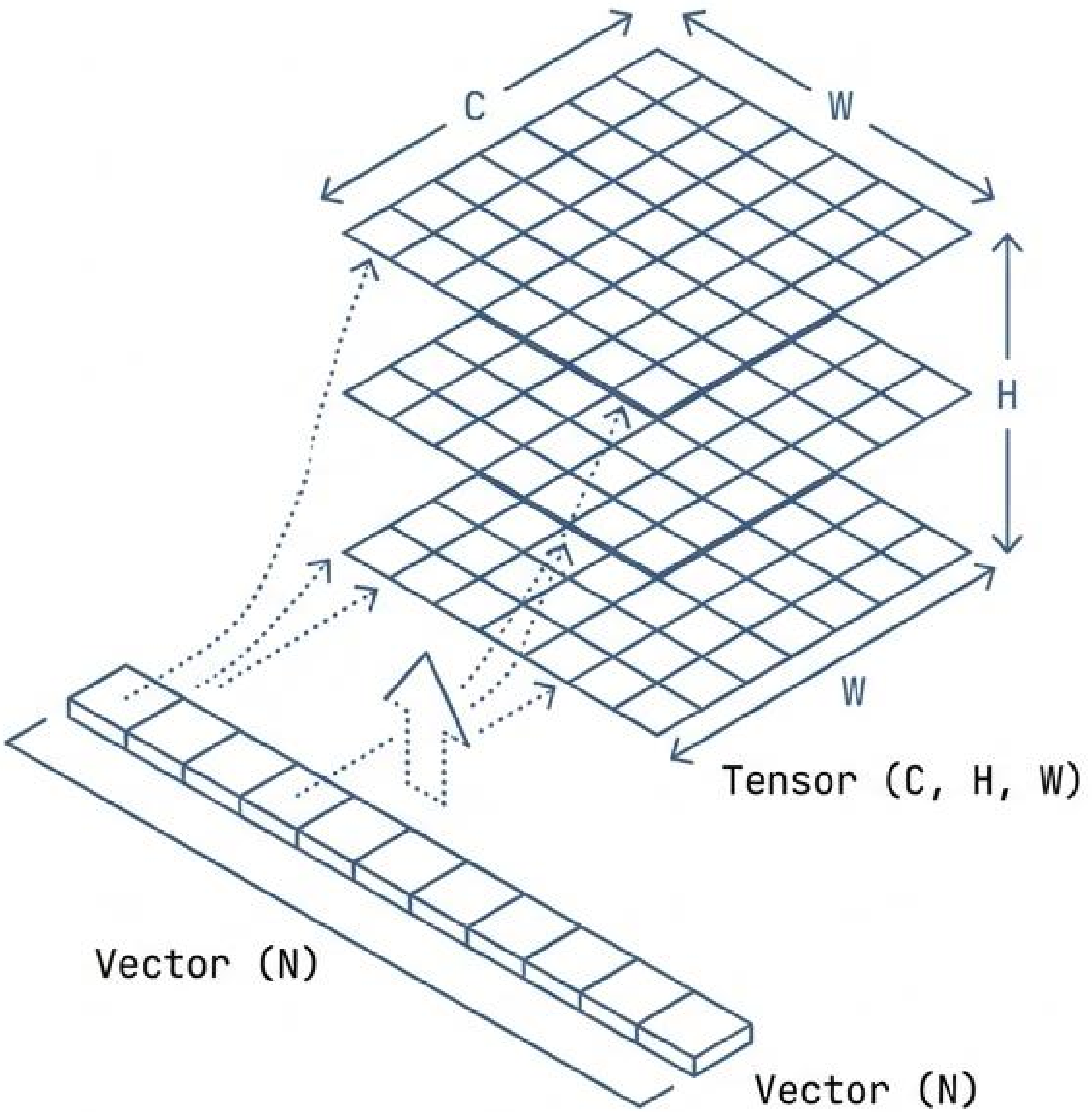
MODULE 09

The CNN Engine

Convolutional neural networks for visual understanding

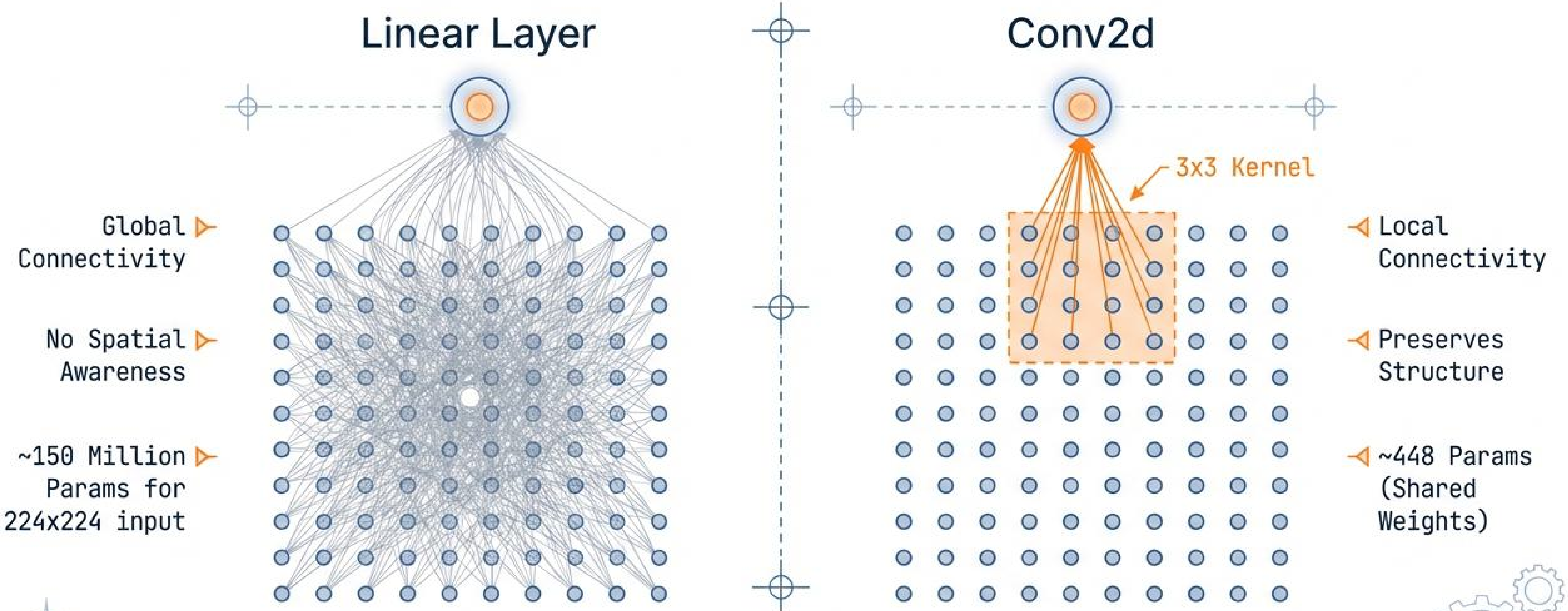
Module 09
Spatial Operations

TINYTORCH ARCHITECTURE TIER



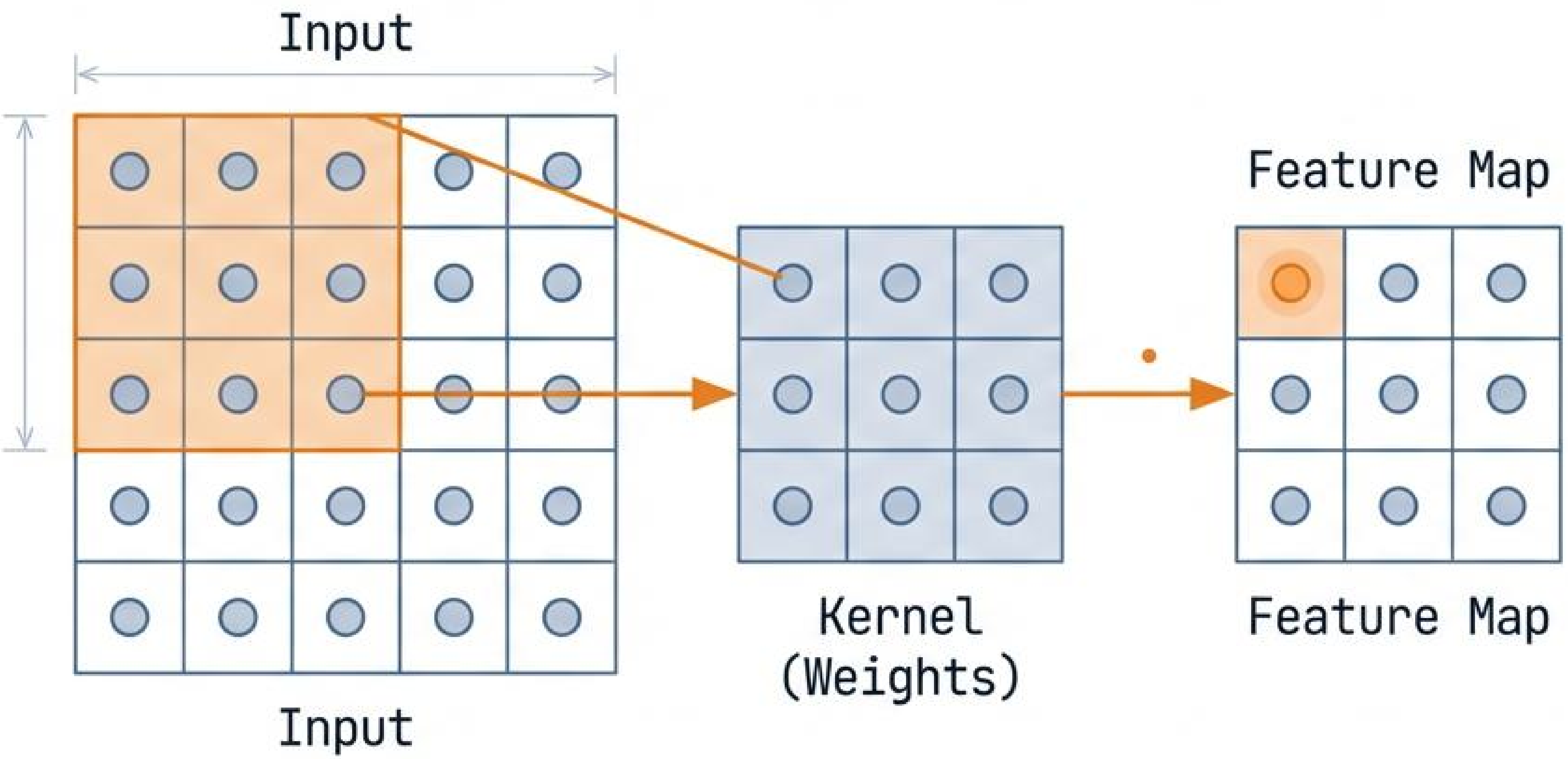
- **GOAL:** Implement Conv2d, Pooling, and BatchNorm from scratch.
- **INPUT:** 4D Tensors (Batch, Channel, Height, Width), Width).
- **OUTPUT:** Hierarchical feature extraction.

Why Linear Layers Fail at Vision



SOLUTION: Enforce Locality (neighbors matter) and Translation Equivariance (features can appear anywhere).

The Abstraction: Convolution



- OPERATION: Sliding window dot-product.
- KERNEL: A small, learnable filter (weights) shared across the image.
- INVARIANT: Parameter Sharing. The same edge detector looks at the top-left and bottom-right.
- OUTPUT: A “Feature Map” encoding activation strength.

The Computational Reality

$$\text{Complexity} = O(B \times C_{out} \times H \times W \times C_{in} \times K_h \times K_w)$$

NAIVE IMPLEMENTATION: 7 Nested Loops

OPERATIONS: ~2.8 Billion ops per forward pass (Standard Layer)

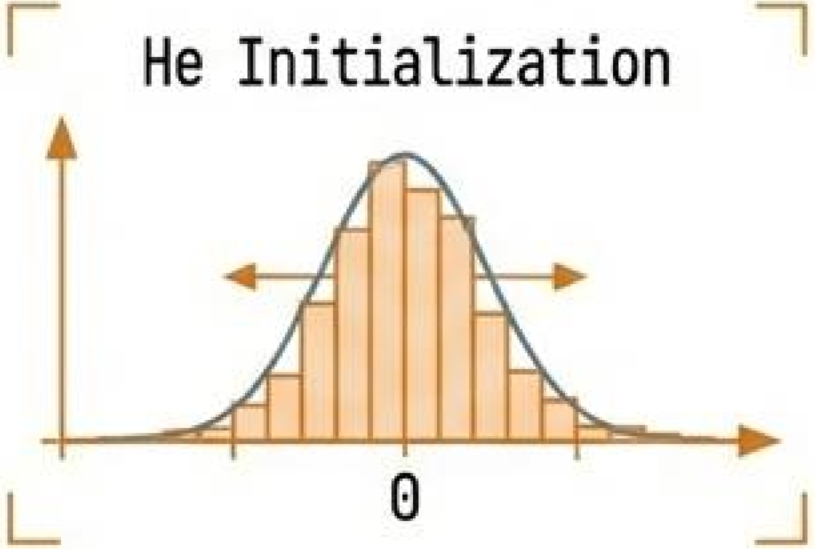
TINYTORCH APPROACH

We will implement these loops explicitly in Python.

Goal: Correctness and understanding memory access patterns (not raw speed).

Conv2d Initialization and State

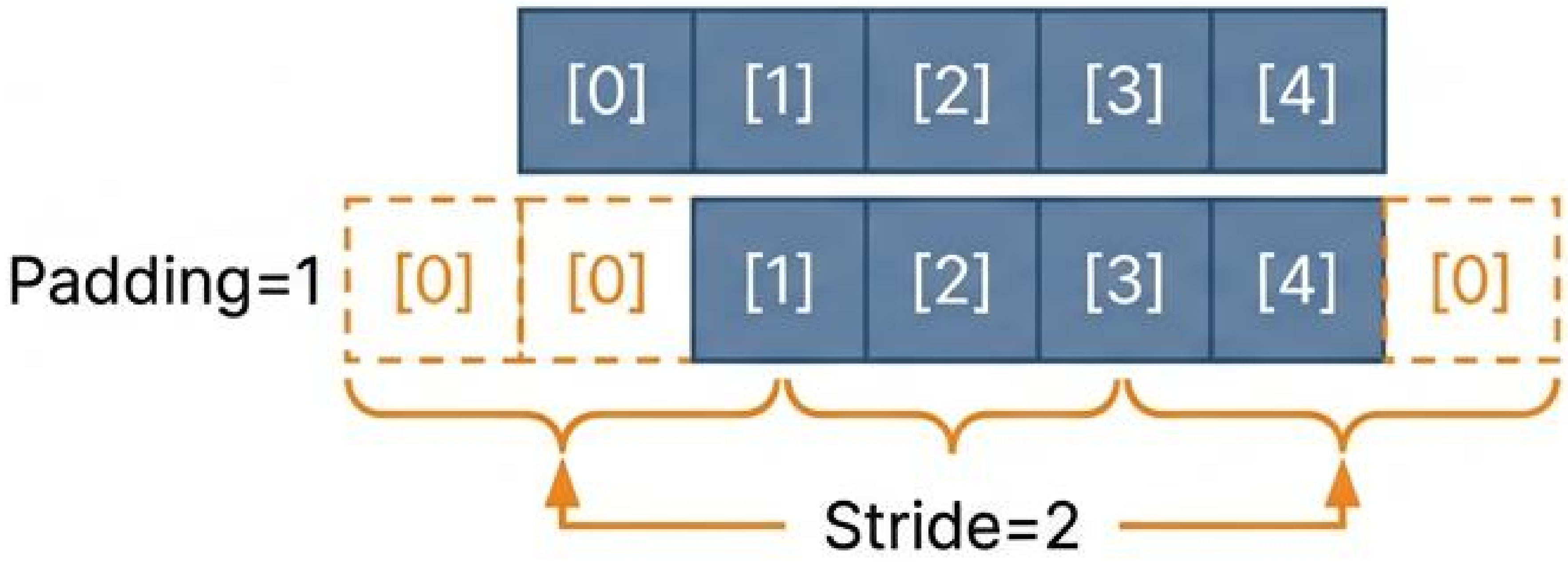
```python
class Conv2d:
    def __init__(self, in_channels, out_channels, kernel_size, …):
        kernel_h, kernel_w = self.kernel_size

        # He Initialization for ReLU networks
        fan_in = in_channels * kernel_h * kernel_w
        std = np.sqrt(2.0 / fan_in)

        self.weight = Tensor(np.random.normal(0, std,
            (out_channels, in_channels, kernel_h, kernel_w)))
```



WEIGHT SHAPE
(out_channels, in_channels, kernel_h, kernel_w)
4D Tensor

STRATEGY
He Initialization: sqrt(2 / fan_in)
Critical for networks using ReLU to prevent vanishing gradients.

Calculating Output Dimensions



Output = floor((Input + 2 * Padding - Kernel) / Stride) + 1

```
out_h = (in_h + 2 * self.padding - kernel_h) // self.stride + 1
out_w = (in_w + 2 * self.padding - kernel_w) // self.stride + 1
```

PADDING: Adds zeros to border. Preserves spatial dimensions.
STRIDE: Steps taken by the window. Downsamples dimensions.
FLOOR DIVISION (//): Truncates edges if dimensions don't align perfectly.

The Algorithmic Reality: Nested Loops

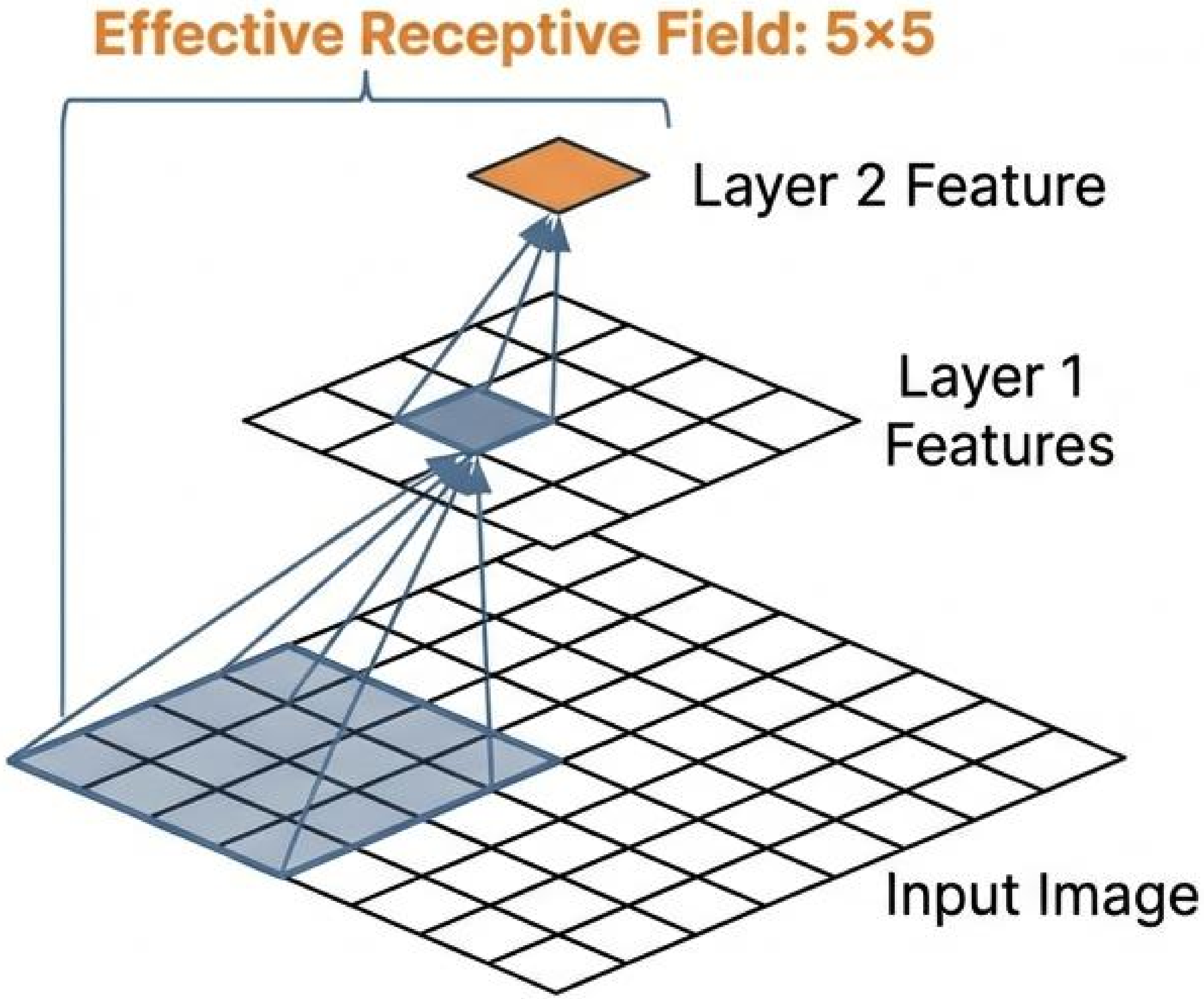
```
# Explicit loop convolution
for b in range(batch_size):
    for out_ch in range(out_channels):    # Loop 2
        for out_h in range(out_height):    # Loop 3 (Spatial)
            for out_w in range(out_width):  # Loop 4 (Spatial)

                # Calculate input region...

                for k_h in range(kernel_h):   # Loop 5 (Kernel)
                    for k_w in range(kernel_w): # Loop 6 (Kernel)
                        # Inner accumulation across in_channels (Loop 7)
                        # conv_sum += input * weight ...

                output[b, out_ch, out_h, out_w] = conv_sum
```

OUTER LOOPS: Iterate every output pixel position.

INNER LOOPS: Iterate every kernel weight position.

SYSTEMS INSIGHT: Every output pixel requires a full kernel calculation.

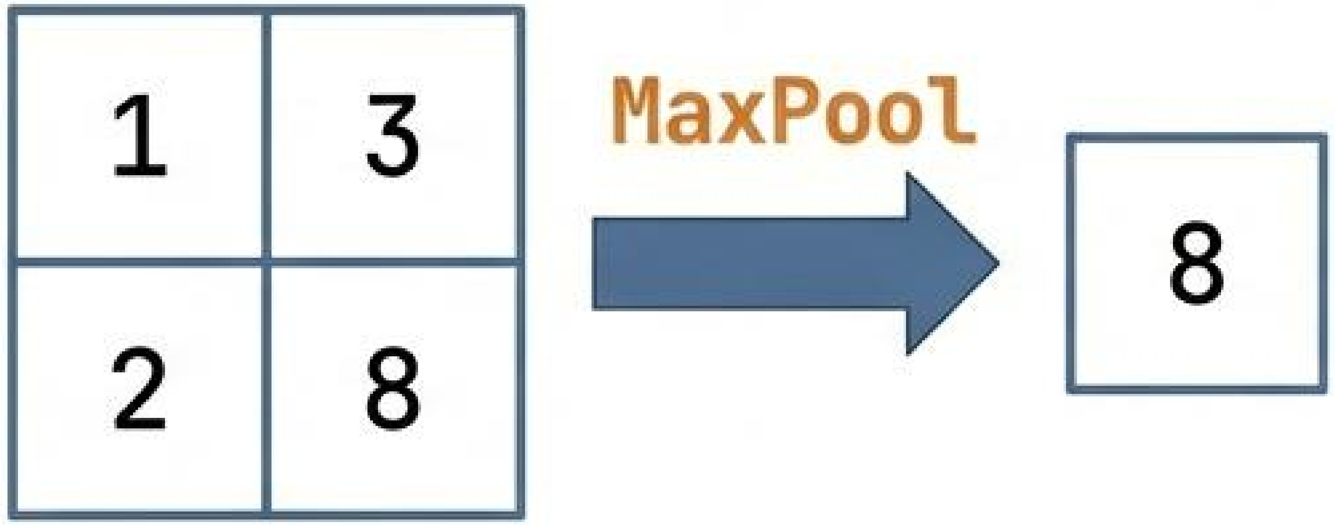
Receptive Fields



- **DEFINITION:** The region in the original input that influences a specific output neuron.
- **GROWTH:** Stacking 3x3 convolutions linearly increases the field (3 -> 5 -> 7).
- **IMPLICATION:** Deeper layers “see” larger parts of the image (Edges -> Shapes -> Objects).

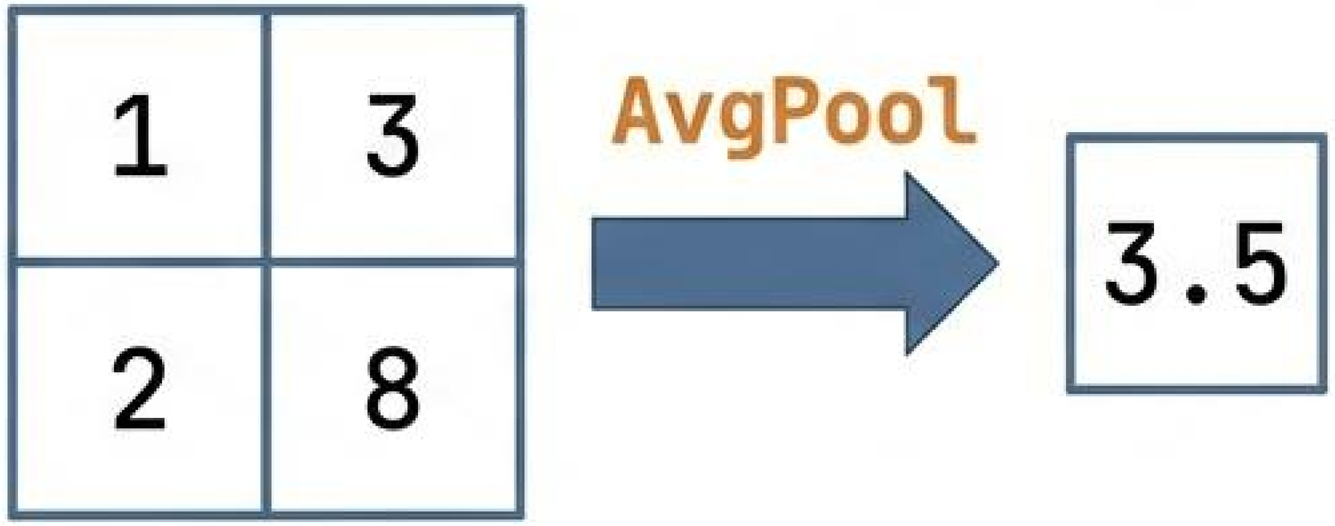
The Compressor: Pooling

MAX POOLING



- Selects strongest signal.
- Preserves edges and textures.
- Invariant to small translations.

AVG POOLING



- Computes mean signal.
- Smooths features.
- Used for backgrounds or global summaries.

GOAL: Reduce spatial dimensions (H, W) to save memory and compute.

Implementing MaxPool2d

```
# Find maximum in window
max_val = -np.inf  # Vital Initialization

for k_h in range(kernel_h):
    for k_w in range(kernel_w):
        input_val = padded_input[b, c,
                                 in_h_start + k_h,
                                 in_w_start + k_w]
        max_val = max(max_val, input_val)

    output[b, c, out_h, out_w] = max_val
```

- **INITIALIZATION:** Must start at -np.inf (Zero is incorrect for negative inputs).
- **STRIDE:** Defaults to kernel_size for non-overlapping windows.
- **CONSTRAINT:** Operates per-channel. No mixing of channels.

Implementing AvgPool2d

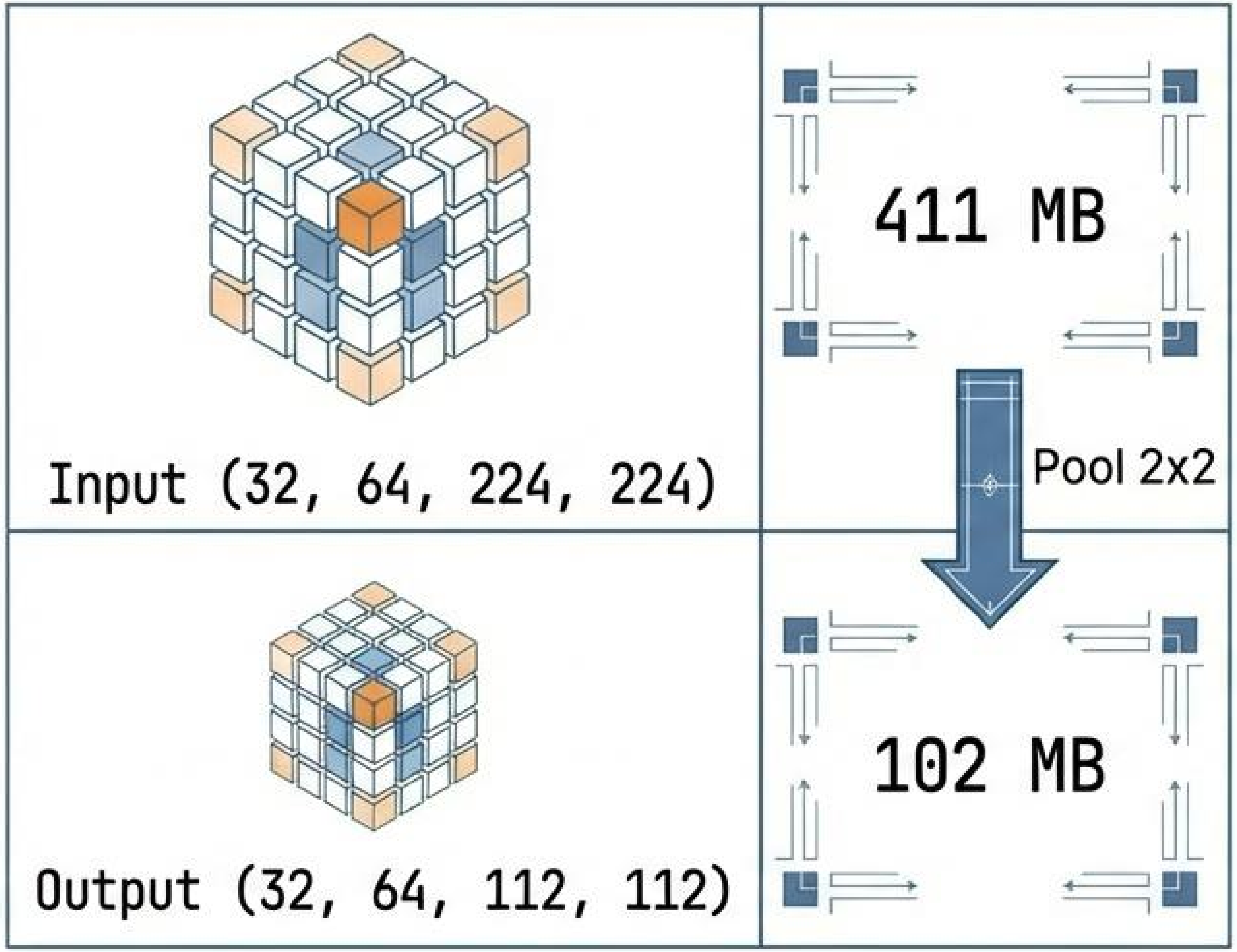
```
# Compute sum in window
window_sum = 0.0

for k_h in range(kernel_h):
    for k_w in range(kernel_w):
        window_sum += padded_input[...]

# Compute average
avg_val = window_sum / (kernel_h *
                        kernel_w)
output[b, c, out_h, out_w] = avg_val
```

MECHANISM: Sum all values in window, divide by area.

USE CASE: Global Average Pooling (late network) or smoothing.

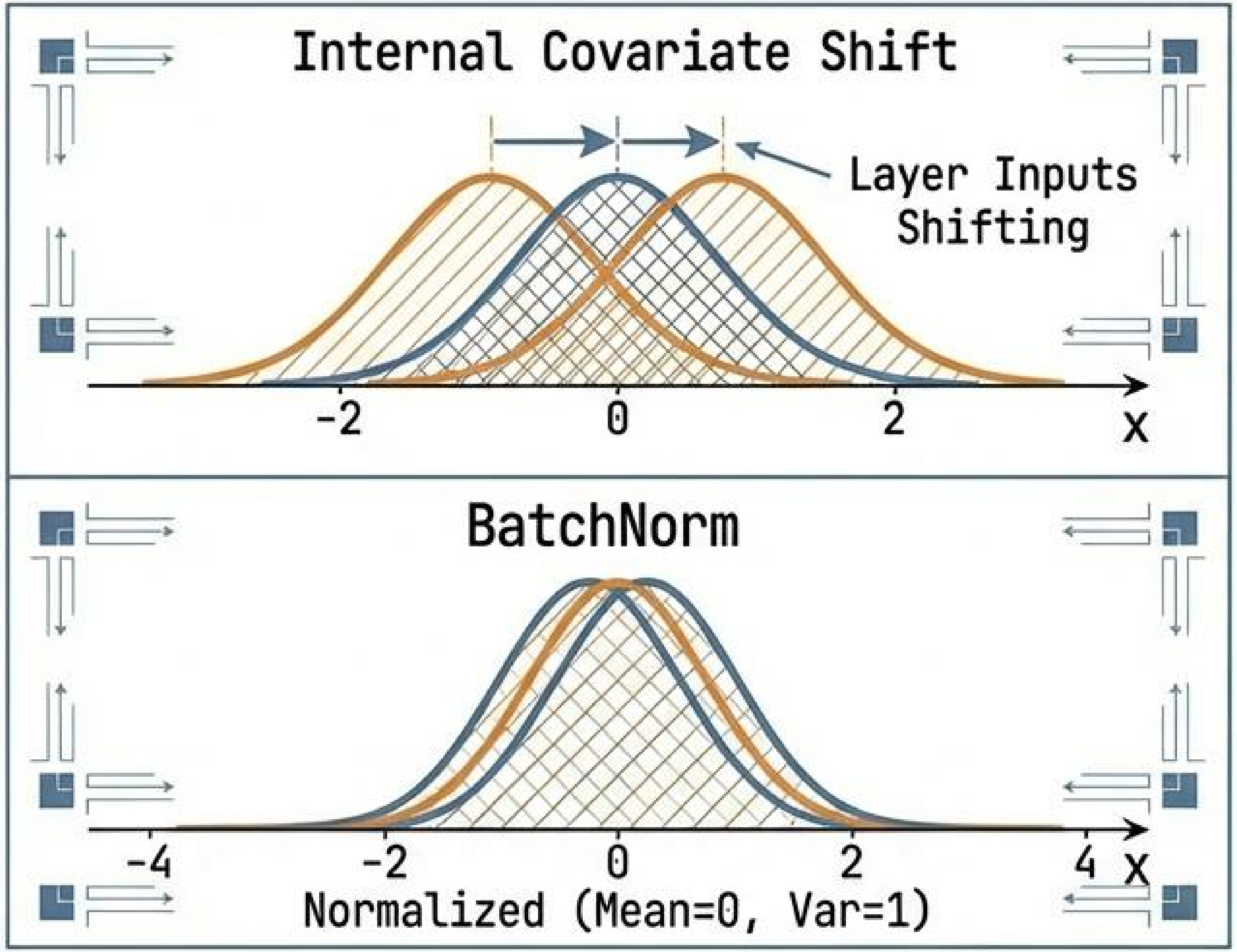
EFFECT: Reduces noise but blurs high-frequency detail.

Systems Analysis: The Cost of Space



- **MEMORY:** Pooling provides a 4x reduction in activation memory.
- **COMPUTE:** Reduces the workload for the *subsequent* convolution layer by 4x.
- **TRADEOFF:** Information loss vs. Efficiency gain.
- **NECESSITY:** Impossible to train deep models on consumer GPUs without aggressive downsampling.

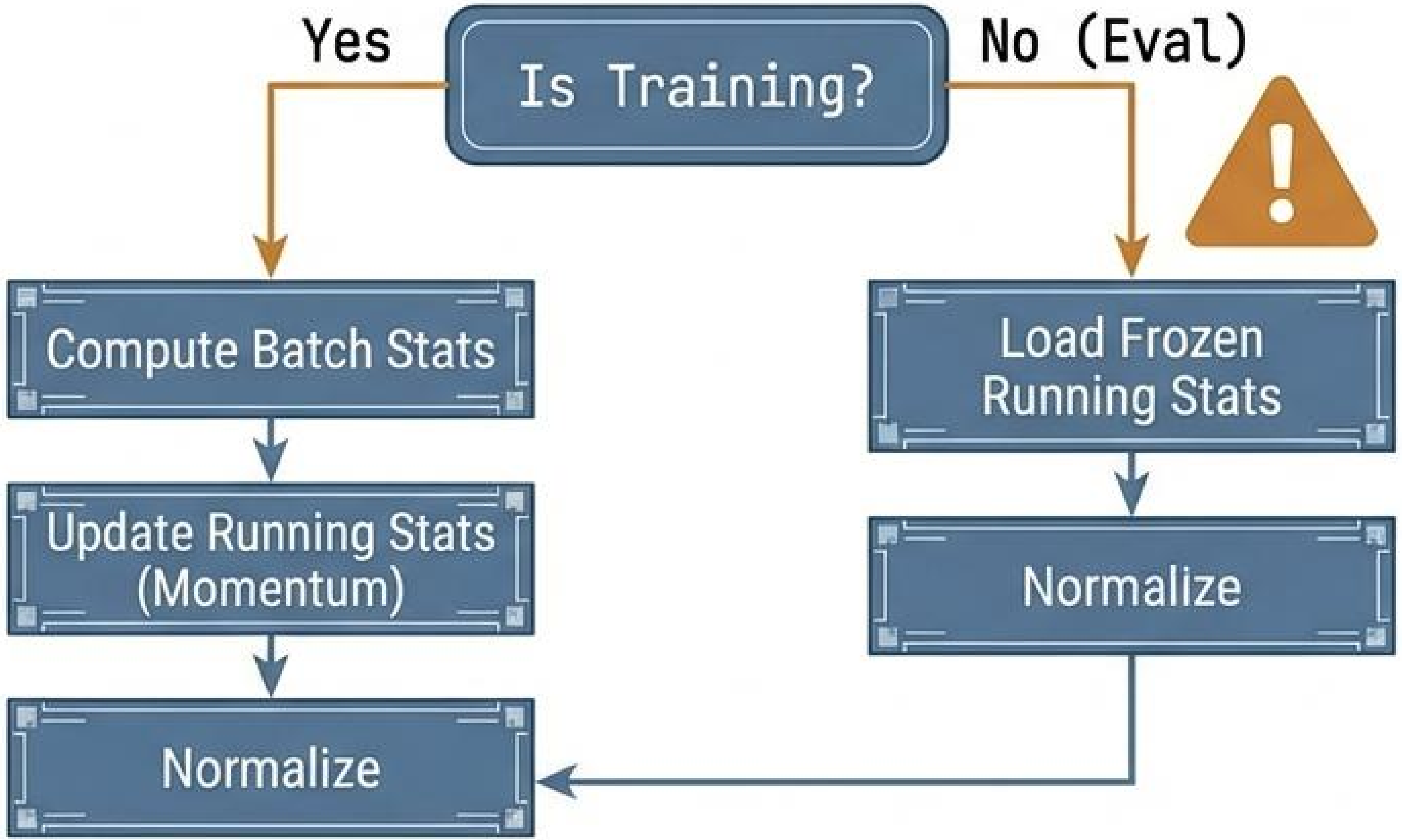
The Stabilizer: BatchNorm



- **PROBLEM:** Deep networks diverge because layer inputs keep changing distribution.
- **SOLUTION:** Normalize mean to 0, variance to 1.
- **RESTORATION:** Learnable parameters Gamma (scale) and Beta (shift) allow the network to “undo” normalization if beneficial.

The Systems Trap: Training vs. Evaluation

TRAINING MODE: Normalize using current batch. Update moving average.



EVAL MODE: Normalize using frozen running stats.

CRITICAL BUG: Using batch stats on a single test image destroys the signal (normalizing a point against itself).

Implementing BatchNorm2d Logic

```
# Training vs. Evaluation Logic
if self.training:
    # Compute batch statistics over (Batch, Height, Width)
    # Aggregates over axes 0, 2, 3
    batch_mean = np.mean(x.data, axis=(0, 2, 3))

    # Update running stats (momentum)
    self.running_mean = (1 - m) * self.running_mean + m *
    mean = batch_mean
else:
    # Use frozen running statistics
    mean = self.running_mean

# Normalize and Apply Gamma/Beta
# State is persistent, Params are learnable
out = gamma * (x - mean) / sqrt(var + eps) + beta
```

AXES: (0, 2, 3). Aggregates over Batch and Space, keeps Channels independent.

STATE: running_mean/var are persistent buffers, not trained parameters.

PARAMS: gamma/beta require gradients.

Building SimpleCNN

```
class SimpleCNN:
    def __init__(self):
        # Block 1: Detect Edges
        self.conv1 = Conv2d(3, 16, kernel_size=3, padding=1)
        self.pool1 = MaxPool2d(2, stride=2)

        # Block 2: Detect Shapes
        self.conv2 = Conv2d(16, 32, kernel_size=3, padding=1)
        self.pool2 = MaxPool2d(2, stride=2)

        # Bridge to Classification
        self.flattened_size = 32 * 8 * 8
```

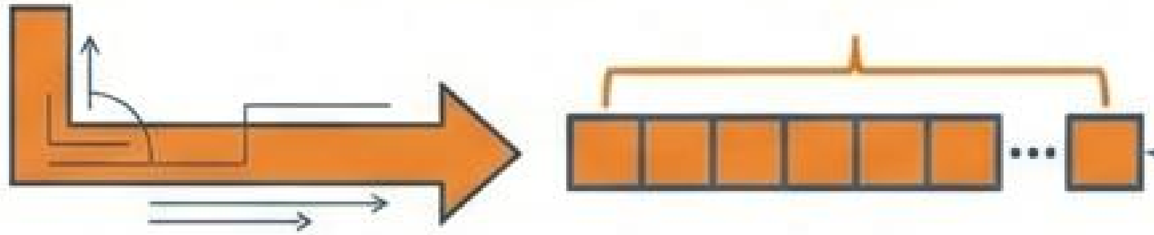
PATTERN:
Conv-ReLU-Pool blocks.

CHANNEL GROWTH:
3 -> 16 -> 32
(Features get richer).

SPATIAL REDUCTION:
32 -> 16 -> 8
(Resolution gets coarser).

The Forward Pass & Flattening

```python
def forward(self, x):
    # Feature Extraction
    x = self.pool1(self.relu(self.conv1(x)))
    x = self.pool2(self.relu(self.conv2(x)))

    # Flattening: Space to Vector
    batch_size = x.shape[0]
    return Tensor(x.data.reshape(batch_size, -1))
```



COMPOSITION: Passing tensors through the chain.

FLATTENING: Bridges the gap between Spatial Tensors (4D) and Vector classifiers (2D).

CNN vs. Dense Networks

Metric	Dense (MLP)	CNN (SimpleCNN)
Input	3072 features (Flat)	32x32x3 (Spatial)
Hidden Layer	1000 units	16 channels (Conv)
Parameters	~3,000,000	~25,000
Efficiency	1x	120x Fewer Params

INDUCTIVE BIAS: CNNs assume spatial structure, allowing massive parameter sharing and **better generalization**.

Systems Takeaways

`CONV2D`: Feature extraction via sliding windows. Expensive (O(N^2 K^2)).

`POOLING`: Necessary for memory management and receptive field growth.

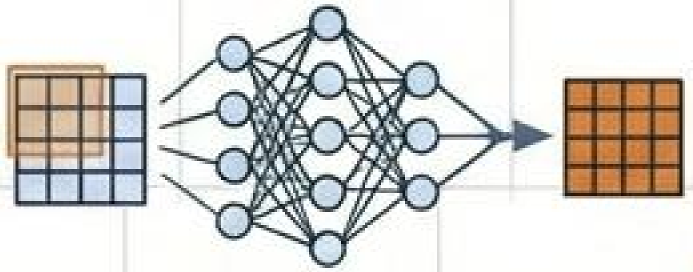
`BATCHNORM`: State-dependent normalization essential for deep convergence.

`IMPLEMENTATION`: Explicit loops reveal the massive parallelism potential (and the bottleneck of Python).

What's Next

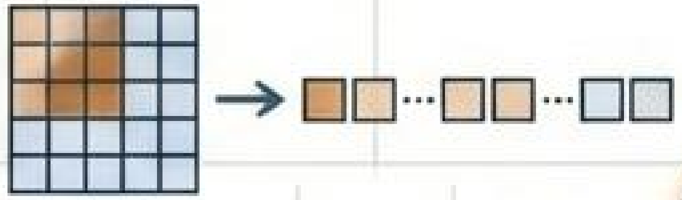
MILESTONE 3

Use this module to train a CNN on CIFAR-10 (Real data).



MODULE 10

Tokenization. Moving from Space (Images) to Sequence (Text).



MODULE 17

We will return to Conv2d to make it fast using im2col and Matrix Multiplication.