AXILOG

SRS-008-POOLING

Max Pooling Layer

Draft v0.1 L2 [D1]

Last updated 20 January 2026
Compliance DO-178C · ISO 26262 · IEC 62304
Changelog 1 entry

Contents

Component: Core / Pooling Operations **Dependencies:** SRS-001 (Matrix) · SRS-002 (Fixed-Point)
Compliance: DO-178C · ISO 26262 · IEC 62304

---

1. Purpose

This module defines requirements for deterministic max pooling operations used in convolutional neural networks (CNNs) to reduce spatial dimensions while preserving salient features.

Critical Requirement: Max pooling must be deterministic, bit-perfect, and execute with zero dynamic memory allocation.

---

2. Background

2.1 Role in CNNs

Spatial Dimension Reduction:

- Reduces feature map size (typically by 2×)
- Decreases computational load in deeper layers
- Creates translation invariance

Feature Preservation:

- Retains maximum activation in each pool region
- Preserves most salient features
- Enables hierarchical feature learning

2.2 Max Pooling Operation

Standard 2×2 Max Pooling (Stride 2):

```
Input:  4×4 feature map
Output: 2×2 feature map (reduced by 2×)

For each 2×2 window, select maximum value:
 ┌─────────────────────┐
 │ a   b │ c   d │       ┌─────────────┐
 │ e   f │ g   h │   →   │max(a,b,e,f)  max(c,d,g,h)│
 ├─────────────────────┤       │max(i,j,m,n)  max(k,l,o,p)│
 │ i   j │ k   l │       └─────────────┘
 │ m   n │ o   p │
 └─────────────────────┘
```

Deterministic Selection:

- No floating-point comparisons
- Fixed-point max operation
- Bit-perfect reproducibility

---

3. Requirements

3.1 Functional Requirements

SRS-008.1: 2×2 Max Pooling

Implement deterministic 2×2 max pooling with stride 2.

Signature:

```
void fx_maxpool_2x2(const fx_matrix_t* in, fx_matrix_t* out);
```

Preconditions:

- `in→rows` and `in→cols` must be even
- `out→rows == in→rows / 2`
- `out→cols == in→cols / 2`
- Both matrices pre-allocated

Behavior:

- Divides input into non-overlapping 2×2 windows

- Selects maximum value from each window
- Writes result to output matrix

Rationale:

- Standard CNN architecture component
- 2×2 pooling most common in practice
- Stride 2 ensures non-overlapping windows

Verification: Unit tests verify correct max selection.

SRS-008.2: Deterministic Max Selection

Max selection must be bit-perfect and independent of floating-point comparison semantics.

Implementation:

```
// Deterministic max for fixed-point values
fixed_t max = window[0];
if (window[1] > max) max = window[1];
if (window[2] > max) max = window[2];
if (window[3] > max) max = window[3];
```

Rationale:

- Integer comparison (no FP rounding)
- Deterministic for all input values
- No NaN or special value handling needed

Verification: Bit-exact tests with boundary values.

SRS-008.3: Input Dimension Validation

Operations shall verify input dimensions are valid for pooling.

Validation:

```
assert(in→rows % 2 == 0);  // Even rows required
assert(in→cols % 2 == 0);  // Even cols required
assert(out→rows == in→rows / 2);
assert(out→cols == in→cols / 2);
```

Rationale:

- Prevents undefined behavior
- Ensures correct output dimensions
- Catches integration errors early

Verification: Tests with invalid dimensions should fail assertions.

3.2 Resource Requirements

SRS-008.4: Zero Dynamic Allocation

Max pooling shall perform no dynamic memory allocation.

Implementation:

- Input/output matrices pre-allocated by caller
- No temporary buffers required
- Stack usage: O(1)

Rationale:

- Consistent with SRS-001 (Matrix) and SRS-003 (Memory)
- Enables static memory analysis
- Required for certification

Verification: Memory leak tests confirm zero allocations.

SRS-008.5: Bounded Stack Usage

Stack usage shall be constant and documented.

Expected Usage:

- Local variables: ~4 bytes (loop counters)
- No recursion

- No variable-length arrays

Rationale:

- Stack overflow prevention
- Enables static analysis
- Required for embedded systems

Verification: Static analysis confirms O(1) stack.

3.3 Performance Requirements

SRS-008.6: Linear Time Complexity

Max pooling time complexity shall be O(M × N) where M×N is input size.

Expected Operations:

```
For input: M×N
Output: (M/2)×(N/2)
Comparisons: M×N (one per input element)
```

Rationale:

- Each input element examined exactly once
- No repeated computations
- Optimal complexity for this operation

Verification: Benchmark confirms linear scaling.

SRS-008.7: Deterministic Execution Time

Execution time shall be independent of input data values.

Implementation:

- Fixed iteration count (input dimensions)
- No data-dependent branches in inner loop
- No early exits

Rationale:

- Required for real-time systems (SRS-007)
- Prevents timing side-channels
- Enables WCET analysis

Verification: Timing tests with varied input patterns.

---

4. Mathematical Properties

4.1 Dimension Reduction

Input-Output Relationship:

```
Given input I with dimensions (H, W)
Output O has dimensions (H/2, W/2)

For each output position (i, j):
O[i][j] = max(I[2i][2j], I[2i][2j+1], I[2i+1][2j], I[2i+1][2j+1])
```

Spatial Preservation:

- Output preserves spatial topology
- Each output corresponds to 2×2 input region
- Non-overlapping windows (stride 2)

4.2 Value Properties

Range Preservation:

```
min(input) ≤ output ≤ max(input)
```

Monotonicity:

- If all input values increase, output values increase or stay same
- Max operation is monotonic

Identity Property:

- If 2×2 window contains identical values, output = that value
- max(x, x, x, x) = x

5. Integration with CNN Pipeline

5.1 Typical Usage Pattern

After Convolution + Activation:

```
// Convolution layer
fx_conv2d(&input, &kernel, &conv_out);  // 16×16 → 14×14

// Activation (ReLU)
fx_relu(&conv_out, &activated);         // 14×14 → 14×14

// Max Pooling
fx_maxpool_2x2(&activated, &pooled);    // 14×14 → 7×7
```

5.2 Memory Planning

Buffer Allocation:

```
// Pre-allocate all buffers
fixed_t conv_buf[196];    // 14×14 after convolution
fixed_t pool_buf[49];     // 7×7 after pooling

fx_matrix_t conv_out, pool_out;
fx_matrix_init(&conv_out, conv_buf, 14, 14);
fx_matrix_init(&pool_out, pool_buf, 7, 7);
```

5.3 Multi-Layer Networks

Stacking Multiple Layers:

```
Input: 32×32
  ↓ Conv3×3
Conv1: 30×30
  ↓ ReLU
Activated1: 30×30
  ↓ MaxPool2×2
Pooled1: 15×15
  ↓ Conv3×3
Conv2: 13×13
  ↓ ReLU
Activated2: 13×13
  ↓ MaxPool2×2
Pooled2: 6×6 (odd dimension, needs padding)
```

---

6. Handling Odd Dimensions

6.1 Problem

Max pooling requires even dimensions. CNNs may produce odd-sized feature maps.

Example:

```
13×13 feature map cannot be evenly pooled with 2×2, stride 2
```

6.2 Solutions

Solution 1: Padding (Recommended)

```
// Pad 13×13 to 14×14 with zeros
fixed_t padded_buf[196];
fx_matrix_t padded;
fx_matrix_init(&padded, padded_buf, 14, 14);
fx_pad_zeros(&input_13x13, &padded);  // Add padding
fx_maxpool_2x2(&padded, &output_7x7);
```

Solution 2: Truncation

```
// Pool only 12×12 region, discard last row/col
// Not recommended - loses information
```

Solution 3: Design Constraint

```
// Ensure all layer dimensions are even
// Plan CNN architecture accordingly
```

Recommendation: Use padding for flexibility, design constraint for optimization.

7. Example Implementation

7.1 Reference Code

```
void fx_maxpool_2x2(const fx_matrix_t* in, fx_matrix_t* out) {
    /* Validate dimensions */
    assert(in->rows % 2 == 0);
    assert(in->cols % 2 == 0);
    assert(out->rows == in->rows / 2);
    assert(out->cols == in->cols / 2);

    uint16_t out_row = 0;

    /* Process each 2×2 window */
    for (uint16_t i = 0; i < in->rows; i += 2) {
        uint16_t out_col = 0;

        for (uint16_t j = 0; j < in->cols; j += 2) {
            /* Extract 2×2 window */
            fixed_t a = in->data[i * in->cols + j];
            fixed_t b = in->data[i * in->cols + j + 1];
            fixed_t c = in->data[(i + 1) * in->cols + j];
            fixed_t d = in->data[(i + 1) * in->cols + j + 1];

            /* Deterministic max selection */
            fixed_t max_val = a;
            if (b > max_val) max_val = b;
            if (c > max_val) max_val = c;
            if (d > max_val) max_val = d;

            /* Write to output */
            out->data[out_row * out->cols + out_col] = max_val;
            out_col++;
        }
        out_row++;
    }
}
```

7.2 Correctness Properties

Fixed Iteration Count:

```
Outer loop: in→rows / 2 iterations
Inner loop: in→cols / 2 iterations
Total: (in→rows × in→cols) / 4 window comparisons
```

No Data-Dependent Branches:

- Loop bounds fixed by dimensions
- Comparison operations deterministic
- No early exits

Memory Access Pattern:

- Sequential reads from input
- Sequential writes to output
- Cache-friendly access

---

8. Testing Requirements

8.1 Unit Tests

Test 1: Basic Functionality

```
Input: 4×4 matrix with known values
Expected: 2×2 output with correct max values
Verify: Bit-exact output comparison
```

Test 2: Boundary Values

```
Test with: Fixed-point MIN, MAX, ZERO
Verify: Correct handling of extreme values
```

Test 3: Uniform Input

```
Input: All elements same value
Expected: Output all same value
```

Test 4: Identity Patterns

```
Input: Diagonal matrix, checkerboard pattern
Verify: Expected max selection
```

8.2 Integration Tests

Test 5: Conv + Pool Pipeline

```
Conv2D → ReLU → MaxPool
Verify: End-to-end correctness
```

Test 6: Multi-Layer Network

```
Conv → Pool → Conv → Pool
Verify: Dimension reduction chain
```

8.3 Performance Tests

Test 7: Timing Consistency

```
Run 10,000 iterations
Measure: Min, max, jitter
Expected: <5% variance (per SRS-007)
```

---

9. Extensions (Future)

SRS-008.8: (Planned) Average Pooling

```
void fx_avgpool_2x2(const fx_matrix_t* in, fx_matrix_t* out);
```

SRS-008.9: (Planned) Configurable Pool Size

```
void fx_maxpool_generic(const fx_matrix_t* in, fx_matrix_t* out,
                        uint16_t pool_h, uint16_t pool_w);
```

SRS-008.10: (Planned) Stride Configuration

```
void fx_maxpool_stride(const fx_matrix_t* in, fx_matrix_t* out,
                       uint16_t pool_size, uint16_t stride);
```

---

10. Commercial Value

10.1 CNN Completeness

With Conv2D + ReLU + MaxPool, we now support:

- ✅ Feature extraction (Convolution)
- ✅ Non-linearity (ReLU)
- ✅ Dimension reduction (MaxPool)

This is a complete CNN building block!

10.2 Real-World Applications

Medical Imaging:

- Chest X-ray classification
- Tumor detection in CT scans
- Retinal disease screening

Automotive:

- Lane detection
- Traffic sign recognition
- Pedestrian detection

Aerospace:

- Satellite image analysis
- Terrain classification
- Target detection

10.3 Certification Advantage

vs. TensorFlow Lite:

- TensorFlow: Dynamic pooling kernels
- Us: Fixed, analyzable implementation ✅

vs. ONNX Runtime:

- ONNX: Generic pooling with many branches
- Us: Specialized 2×2, minimal branching ✅

---

11. References

- **LeCun et al. (1998)** - "Gradient-Based Learning Applied to Document Recognition"
- **Krizhevsky et al. (2012)** - "ImageNet Classification with Deep CNNs"
- **DO-178C** - Software Considerations in Airborne Systems
- **ISO 26262-6:2018** - Automotive functional safety

---

12. Revision History

Version	Date	Author	Changes
1.0	2026-01-15	William Murray	Initial version

Document Classification: Technical Specification
Approval Status: Approved for Implementation
Next Review: 2026-04-15
