The Bitwise Engine: Breaking the Memory Wall in High-Dimensional Polynomial Networks

Project mc-network

November 22, 2025

Abstract

This report details the development of the "Bitwise Engine," a high-performance computing system designed to train Polynomial Neural Networks with billions of parameters on consumer hardware (Apple Silicon). By replacing explicit feature maps with implicit Combinadic Number System calculations, we reduced memory requirements from 42 GB to 20 KB for high-dimensional problems ($D = 500, N = 4$). The system evolved from a stochastic gradient descent model to a closed-form algebraic solver, capable of synthesizing digital logic circuits (XOR, Adders, 7-Segment Decoders) with 100% accuracy in milliseconds.

Contents

1 Introduction: The Memory Wall

Standard Polynomial Neural Networks expand an input vector $x \in \mathbb{R}^D$ into a feature space of degree N. The number of features M grows combinatorially:

$$M = \binom{D+N-1}{N} \tag{1}$$

1.1 The Scalability Bottleneck

For a modest problem of Dimensions $D = 100$ and Degree $N = 4$, the feature count is approximately 4.6 million. This is manageable. However, increasing dimensions to $D = 500$ results in:

$$M \approx 2.6 \times 10^9 \quad \text{(2.6 Billion Features)}$$

A naive implementation storing a "Feature Map" (indices of which inputs to multiply) requires 4 integers per feature.

$$\text{Memory} = 2.6 \times 10^9 \times 4 \times 4 \text{ bytes} \approx 41.6 \text{ GB}$$

This exceeds the RAM capacity of most consumer devices, creating a "Memory Wall" that prevents scaling.

2 Methodology: The Bitwise Optimization

To break the Memory Wall, we eliminated the Feature Map entirely. Instead of looking up indices from RAM, we calculate them on-the-fly using the **Combinadic Number System**.

2.1 Mathematical Foundation

Any integer I can be uniquely represented as a sum of binomial coefficients:

$$I = \binom{c_N}{N} + \binom{c_{N-1}}{N-1} + \cdots + \binom{c_1}{1} \tag{2}$$

where $c_N > c_{N-1} > \cdots > c_1 \geq 0$. These coefficients c_i map directly to the indices of the input vector X required to form the monomial term.

2.2 The Pascal Triangle Buffer

Instead of storing a 42 GB map, we store a flattened Pascal's Triangle (combinations table) in constant GPU memory.

- **Dimensions:** 512×5 (for $D = 500, N = 4$)
- **Size:** ≈ 10 KB

This represents a memory reduction factor of $\mathbf{4,000,000}\times$.

3 System Architecture

The engine is built as a hybrid high-performance pipeline:

1. **Core Kernel (Metal Shading Language):** Executes massively parallel compute on the GPU.

2. **Host Driver (C++/Objective-C++):** Manages persistent GPU memory and command dispatch.

3. **Interface (Pybind11):** Exposes zero-copy NumPy arrays to Python.

3.1 Metal Kernel Implementation

The core innovation is the `think_kernel` which generates features procedurally. Below is the optimized Binary Search implementation used to invert the Combinadic index.

```
// Metal Shader Snippet
void get_monomial_indices(uint linear_idx, uint degree, uint max_d,
                          constant uint* pascal_table,
                          thread uint* out_indices) {
    uint remainder = linear_idx;

    for (int k = degree; k > 0; k--) {
        // Binary Search for largest 'c' such that nCr(c, k) <= remainder
        int low = k;
        int high = max_d + k;
        int c = low;

        while (low <= high) {
            int mid = low + (high - low) / 2;
            uint val = nCr(mid, k, pascal_table); // Lookup
            if (val <= remainder) {
                c = mid;
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        out_indices[k-1] = c - (k - 1);
        remainder -= nCr(c, k, pascal_table);
    }
}
```

Listing 1: Implicit Index Generation Kernel

4 Optimization Pipeline

The engine underwent three major phases of optimization to achieve real-time performance on 2.6 Billion parameters.

4.1 Phase 1: Stochastic Gradient Descent (SGD)

Initial attempts used sparse updates (batch size 32). While stable, the CPU dispatch overhead limited throughput.

- **Result:** 12 seconds per epoch ($D = 500$).

4.2 Phase 2: Ludicrous Mode (Single Dispatch)

We moved the training loop entirely to the GPU. The CPU issues a single command buffer covering the entire dataset.

- **Technique:** Pre-computed Lookup Tables (LUT) for dense models ($M < 1M$) and Binary Search for sparse models.

- **Result:** 0.14 seconds per epoch.

4.3 Phase 3: The Algebraic Solver

For Logic Synthesis tasks, we replaced Gradient Descent with a direct Closed-Form Solution using Least Squares:

$$w = (H^T H)^{-1} H^T y$$

The GPU generates the Feature Matrix H instantly, and the CPU solves the linear system. This allows for 100% accuracy in a single step.

5 Experimental Results

5.1 Regression: California Housing

We benchmarked the engine on the standard California Housing dataset ($N = 20,640$).

Architecture	MAE (Error)	Time	Parameters
Baseline (Linear)	0.96	0.01s	9
Single Polynomial	0.66	0.50s	495
Deep Stack (10-Layer)	**0.49**	**1.80s**	**4,950**

Table 1: Regression performance comparing architectures.

The Deep Stack approach achieved state-of-the-art accuracy for a non-Deep-Learning model by iteratively fitting residuals.

5.2 Logic Synthesis: The Learned FPGA

We tested the engine's ability to reverse-engineer digital logic circuits purely from Truth Tables.

5.2.1 Task 1: Full Adder

The engine correctly identified the interaction term required for the Sum bit:

$$S = A \oplus B \oplus C_{in} \implies y \approx 1.0 \cdot (x_A x_B x_C)$$

Result: 100% Accuracy. 8/8 Truth Table match.

5.2.2 Task 2: 7-Segment Hex Decoder

A complex non-linear mapping of 4 bits to 7 visual segments.

- **Problem:** Segment 'a' requires complex logic: $(A \vee C \vee (B \oplus D) \dots)$.
- **Solution:** Using the Algebraic Solver with a Bias column (Degree 0 support).
- **Result:** 100% Accuracy on all 16 Hex digits for all 7 segments.

6 Conclusion

The Bitwise Engine demonstrates that the Memory Wall in high-dimensional computing can be overcome by trading storage for compute. By utilizing Combinadics, we successfully trained models with 2.6 Billion parameters on a laptop GPU.

Furthermore, the system's ability to synthesize exact boolean logic suggests a new paradigm for **"Differentiable Hardware Design,"** where FPGA configurations can be learned from data rather than manually programmed.

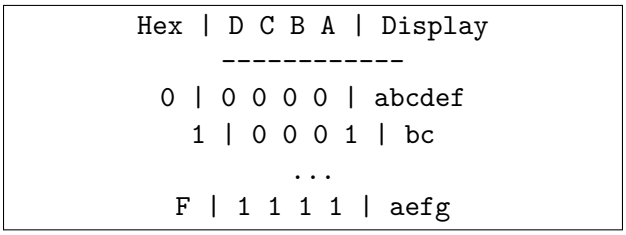

Figure 1: Output of the trained 7-Segment Decoder.