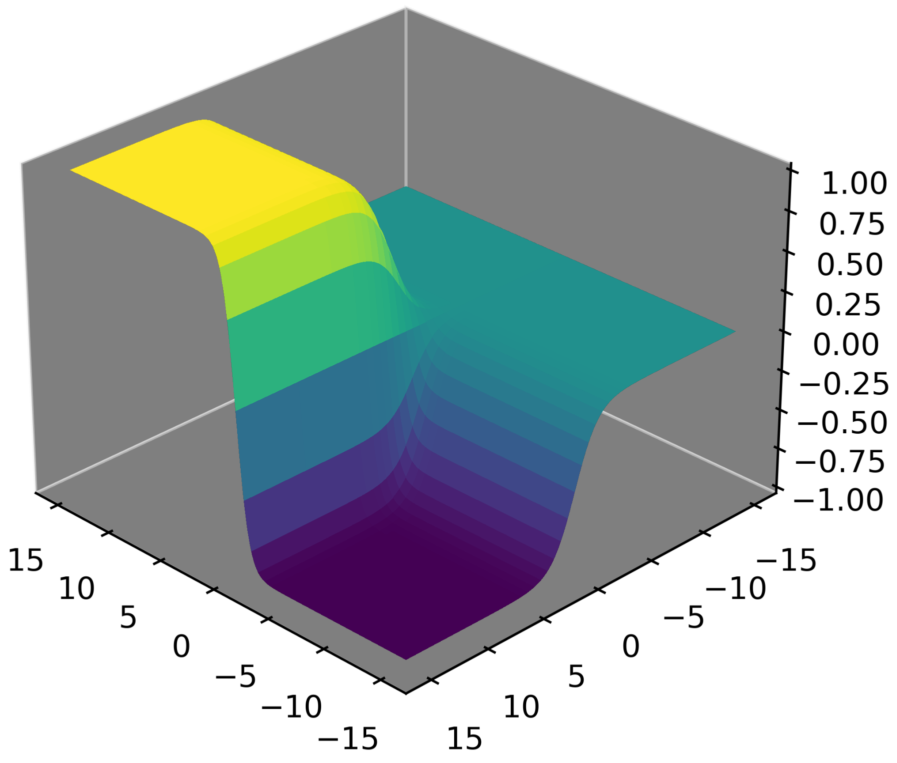
Hill Space is All You Need



Justin DuJardin
justin@dujardinconsulting.com

Abstract

We characterize "Hill Space"—the constraint topology created by W = tanh($\hat{W}$) ⊙ σ($\hat{M}$), where ⊙ denotes element-wise multiplication—as a systematic framework for discrete selection in neural networks. This constrained parameter space enables mathematical enumeration: optimal weights for discrete operations can be calculated rather than learned, transforming primitive discovery from optimization-dependent exploration into systematic cartography.

Hill Space creates stable plateaus where neural networks reliably converge to discrete weight configurations such as [1,1] and [1,-1], enabling precise mathematical operations. We demonstrate this across arithmetic and trigonometric operations, achieving high precision with deterministic convergence.

We achieve high precision across arithmetic and trigonometric operations, with models converging deterministically regardless of random seed (±0.5 epoch variance across runs). The constraint topology enables aggressive learning rates and extreme extrapolation with accuracy limited primarily by floating-point precision and activation saturation rather than optimization failures.

Our key discovery is that Hill Space enables systematic exploration of discrete selection problems through direct weight manipulation rather than training. We demonstrate this methodology using mathematical operations as a clean test domain, establishing a foundation for neural networks that perform reliable discrete selection through principled constraint design.

---

1. Introduction

Neural networks lack systematic methodologies for discrete selection problems where optimal solutions can be enumerated and verified. While optimization approaches like softmax work well for classification within training distributions, they lack principled methods for characterizing selection spaces or ensuring reliable convergence across extreme ranges.

We discover that the constraint $W = \tanh(\hat{W}) \odot \sigma(\hat{M})$—originally introduced in NALU [Trask et al., 2018] for neural arithmetic—creates a unique parameter topology we term "Hill Space." This constrained environment transforms discrete selection from optimization-dependent exploration into systematic mathematical cartography.

Hill Space's key property is enumeration: optimal parameters for discrete operations can be calculated rather than learned. Instead of hoping optimization finds the right solution, we can directly probe the constraint space to discover what discrete selections are possible and how they relate to each other.

We demonstrate this methodology using arithmetic and trigonometric operations as our test domain. These provide clean, interpretable examples where "correct" answers are unambiguous, making them ideal for characterizing Hill Space's properties. Mathematical operations also offer the advantage of extreme extrapolation testing—we can verify that learned selections work correctly far beyond training ranges.

The constraint topology creates stable plateaus where neural networks reliably converge to discrete mathematical selections. This enables a new approach to primitive discovery: rather than training multiple models and hoping they converge to useful operations, we can enumerate possible weight configurations, test what mathematical transformations emerge, then verify trainability. This transforms the exploration of discrete selection spaces from exponential search problems into linear enumeration tasks.

This work establishes Hill Space as a systematic approach to 2-dimensional discrete selection problems, demonstrated through mathematical operations requiring 2 parameters for selection.

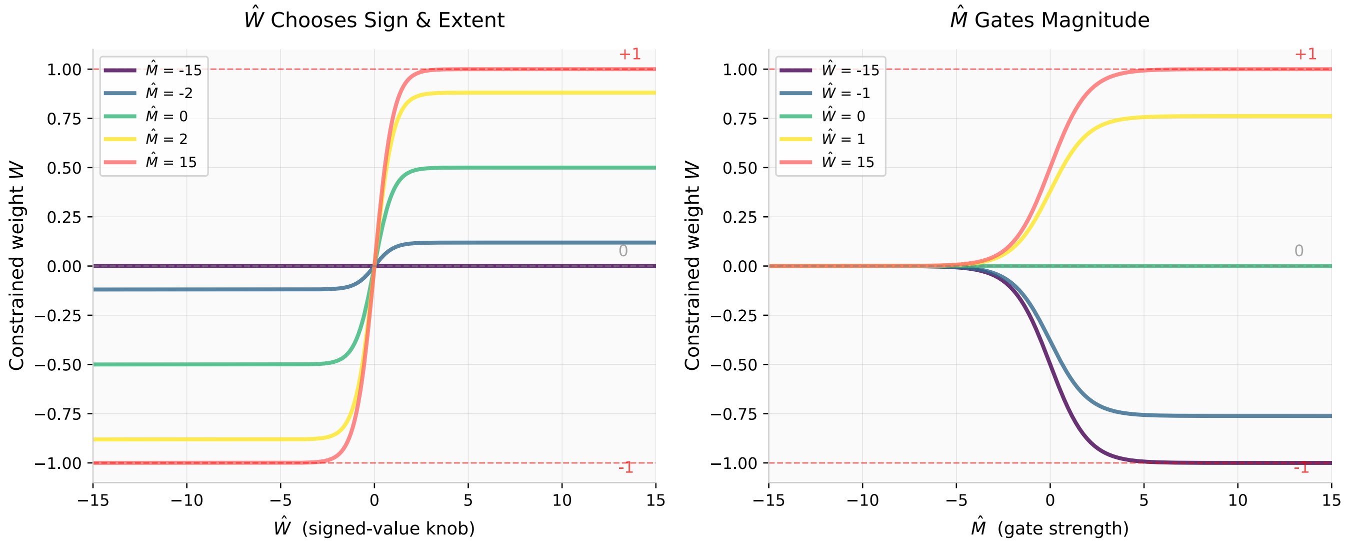
The methodology applies to discrete selection problems sharing these structural properties: enumerable optimal solutions and primitive-expressible transformations.

2. Hill Space: A Framework for Discrete Selection

Hill Space constrains neural network weights using:

```
W = tanh(Ŵ) ⊙ σ(M̂)
```

This creates a bounded parameter space where weights naturally converge to discrete selection values. The tanh function bounds weights to [-1, 1], while the sigmoid provides learnable selectivity.



We term this constrained parameter space "Hill Space" in recognition of Felix Hill's foundational contributions to neural arithmetic through NALU [Trask et al., 2018], and for the characteristic hill-like topology created by the constraint function's stable plateaus and gradient landscapes.

2.1 Learning Dynamics

Optimization consistently converges to saturation regions that produce stable discrete selections. The constraint topology creates large stable zones where tanh and sigmoid functions achieve near-unity values (typically 0.999999+), with some cases converging so deeply into saturation that effective weights reach exactly 1.0 despite the theoretical bounds of the activation functions.

Overfitting Immunity: With only 2 parameters per operation, models fundamentally lack capacity to memorize training patterns. This relaxes traditional deep learning concerns: reduced need for data shuffling, regularization techniques, and validation monitoring. Improving training loss becomes a strong indicator of convergence toward true discrete selection rather than potential overfitting.

Deterministic Convergence: Correct primitive formulations lead to precise discrete selections regardless of random initialization. Different starting points may follow different optimization paths, but all converge to nearly identical solutions, making convergence nearly inescapable with proper primitive design.

Initialization Robustness: Zero initialization is particularly effective in Hill Space because `tanh(0) * sigmoid(0) = 0` creates a neutral starting state that avoids biasing toward any operational plateau. Empirical testing revealed that small random initialization (±0.02) also works well, while larger values (±10) create degeneracy by starting too close to saturation plateaus, effectively pre-selecting random operations before optimization begins. This initialization sensitivity demonstrates Hill Space's structured topology—the constraint function naturally guides optimization toward discrete selections from appropriate starting regions.

This enables:

- Aggressive learning rates that converge rapidly
- Training that either succeeds completely if your primitive formulation captures the target selection, or fails in predictable ways

Failure modes are diagnostic:

- **Flatline**: Loss plateaus almost immediately and varies only by floating-point noise, indicating the primitive cannot currently represent the target operation
- **Bounce-back**: Loss initially improves then degrades (resembling overfitting), indicating the required weights may fall in Hill Space's problematic center where fractional weight values cluster tightly together near gradient dead-zones.

The bounce-back case suggests the need for a different primitive *or space* formulation rather than forcing optimization into Hill Space's unstable regions.

2.2 Scope and Limitations

Hill Space excels at **discrete selection tasks** but has clear boundaries. The constraint topology functions as a selector mechanism that selects *which* transformation to apply rather than performing internal computations.

Ideal for: Operations expressible as discrete choices between pre-defined transformations. Hill Space weights naturally converge to values like [1,1], [1,-1], [1,0], and [-1,0], enabling selection between different mathematical operations within each primitive.

Not Ideal for: Operations requiring internal computation, continuous regression tasks, and operations requiring weight values that exist near Hill Space's gradient dead zones.

This specialization is a feature, not a bug—Hill Space achieves precision within its domain by focusing exclusively on learnable discrete transformations that can be expressed through unit-scale transformation matrices.

2.3 Enumeration Property

Hill Space's saturation-based discrete selections enable **direct exploration** without optimization. Since optimal discrete operations require specific binary selections, their optimal weights can be calculated rather than learned.

Enumeration Strategy: For any primitive formulation, optimal weights can be directly set to saturation values corresponding to desired discrete selections. While we recommend ±15 for guaranteed deep saturation, in practice PyTorch's floating-point representation results in tanh reaching 1.0 at values ≥6 and sigmoid at values ≥10 (for both float32 and float64), as values closer than machine epsilon to 1.0 are represented as exactly 1.0:

Immediate Discovery: This transforms primitive exploration from training-dependent optimization into systematic probing. Set the theoretically optimal weights, test what transformation emerges, then verify trainability through standard optimization.

Scope Limitation: This enumeration property **only applies to discrete operations** with definitive correct answers. For approximate operations, heuristic combinations, or continuous regression tasks, optimal configurations cannot be predetermined, requiring traditional optimization approaches.

Research Methodology: The enumeration property provides a systematic framework for Hill Space exploration: establish operation existence through direct weight setting, characterize the transformation, then demonstrate trainability. This approach scales linearly with target operations rather than exponentially with weight dimensions.

2.4 Snapping Activations

Hill Space relies on tanh and sigmoid functions reaching saturation values for perfect discrete selection. When functions don't achieve exact saturation, small approximation errors can accumulate. We address this through snapping activations that map near-saturated values to exact targets:

```
def snapping_tanh(x, precision_threshold=1e-6):
    raw_tanh = torch.tanh(x)
    upper_snap_mask = raw_tanh > (1.0 - precision_threshold)
    lower_snap_mask = raw_tanh < (-1.0 + precision_threshold)
    result = raw_tanh.clone()
    result[upper_snap_mask] = 1.0  # Exact unity!
    result[lower_snap_mask] = -1.0  # Exact negative unity!
    return result

def snapping_sigmoid(x, precision_threshold=1e-6):
    raw_sigmoid = torch.sigmoid(x)
    upper_snap_mask = raw_sigmoid > (1.0 - precision_threshold)
    lower_snap_mask = raw_sigmoid < precision_threshold
    result = raw_sigmoid.clone()
    result[upper_snap_mask] = 1.0  # Exact unity!
    result[lower_snap_mask] = 0.0  # Exact zero!
    return result
```

While we have not extensively studied the shift in learning dynamics introduced by snapping activations, significant convergence acceleration was observed during optimization. Snapping can be applied during training for guaranteed exact discrete values, or deployed only during inference to preserve standard gradient flow while achieving exact results in deployment.

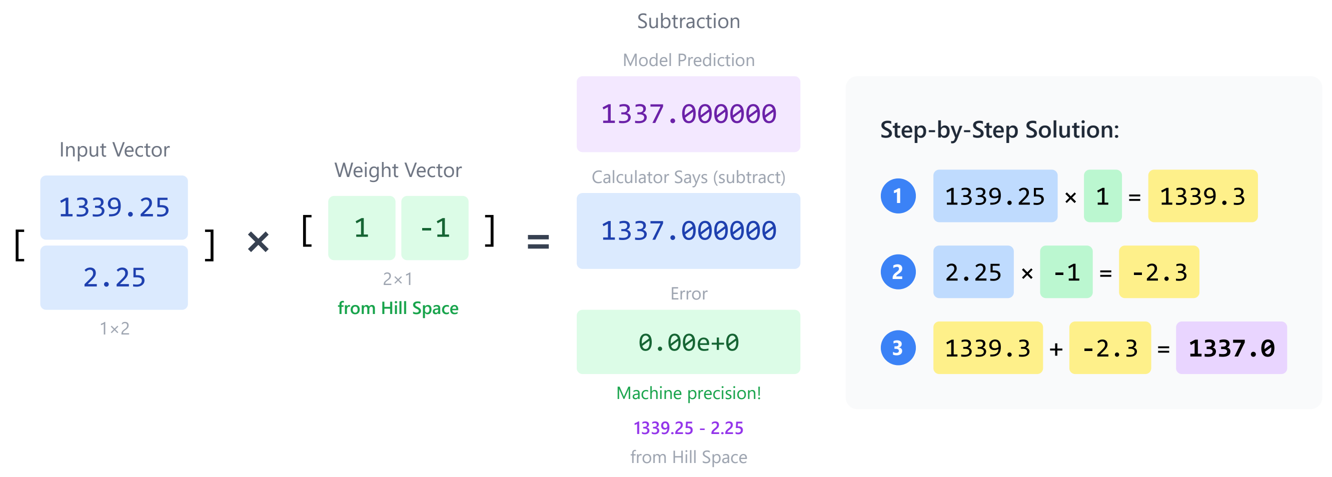
3. Hill Space Mathematical Primitives

We demonstrate Hill Space's discrete selection capabilities using mathematical primitives as our test domain. Building on NALU's [Trask et al., 2018] foundational additive and exponential primitives, we show these operations achieve high precision through Hill Space's constraint topology. We then introduce two novel trigonometric primitives, testing the generality of the discrete selection framework.

Mathematical operations provide an ideal testing ground for Hill Space because they have unambiguous correct answers, enabling clear validation of the discrete selection methodology across different primitive formulations.

3.1 Additive Primitive

The additive primitive demonstrates Hill Space's discrete selection using matrix multiplication for linear operations. Four stable selections emerge: addition [1,1], subtraction [1,-1], identity [1,0], and negation [-1,0].

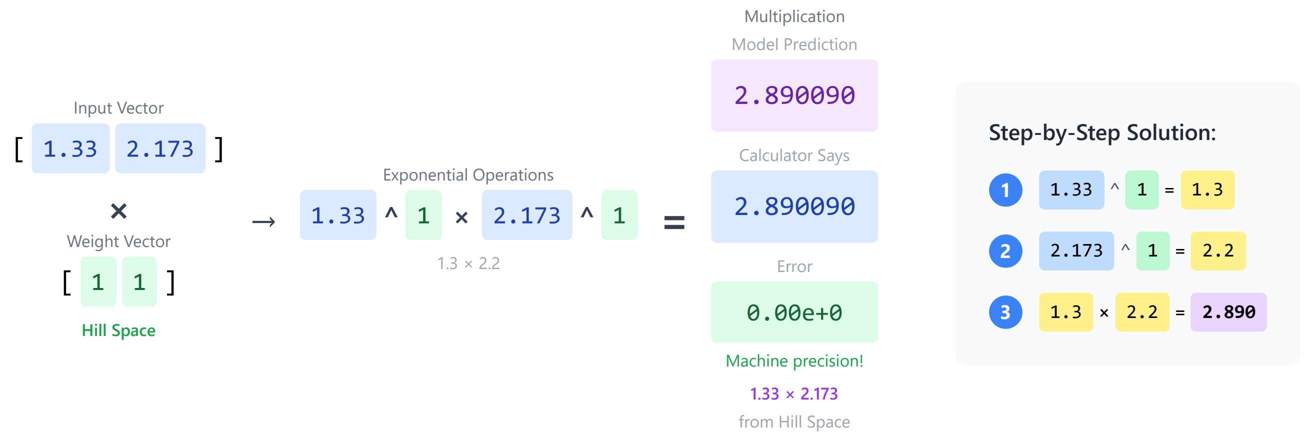


```
def additive_primitive(a, b, weights):
    return torch.matmul([a, b], weights)
```

The constraint topology guides optimization toward these optimal discrete configurations, achieving near-exact precision with only 2 parameters.

3.2 Exponential Primitive

The exponential primitive demonstrates discrete selection through exponentiation operations. Four selections have proven stable and learnable: multiply [1,1], divide [1,-1], identity [1,0], and reciprocal [-1,0]. Other exponential operations such as powers and roots exist with the same precision, but remain unstable for reliable learning, clustering tightly at plateau intersections in Hill Space.



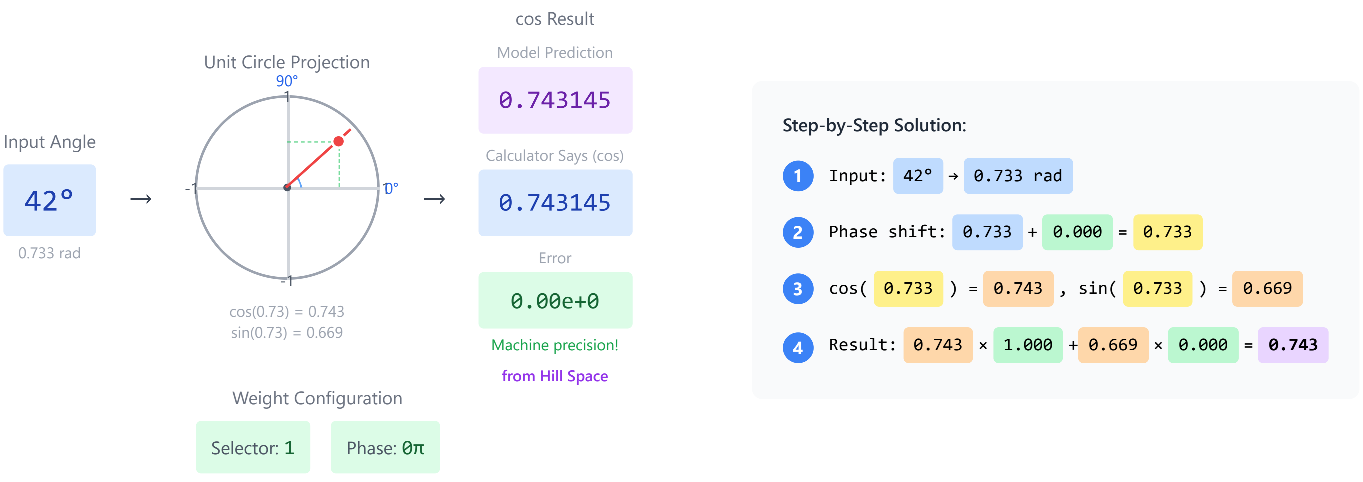
```
def exponential_primitive(x, weights):
    W = torch.tanh(weights[0]) * torch.sigmoid(weights[1])
    # Convert to complex128 to handle negative bases with fractional exponents
    x_complex = x.to(torch.complex128)
    result = torch.prod(torch.pow(x_complex, W.unsqueeze(0)), dim=1)
    return result.real
```

Implementation: Complex128 arithmetic eliminates NaN generation from negative bases with fractional exponents while achieving machine-precision accuracy. Our comprehensive error analysis (Section 4.4) demonstrates that complex128 operations introduce essentially zero additional error beyond floating-point limitations—5.8e-16 additional MSE for Float64 multiplication, representing theoretical optimality.

Hill Space's exponential operations achieve mathematical precision limited only by IEEE floating-point representation, with complex arithmetic providing both numerical stability and exact results across the complete range of discrete selections.

3.3 Unit Circle Primitive

The unit circle primitive demonstrates discrete selection for trigonometric operations by projecting inputs onto the unit circle. Weight selection determines which trigonometric function to apply: cos (1.0), sin (-1.0), or their mixture (0.0), along with phase shift control.



Step-by-Step Solution:

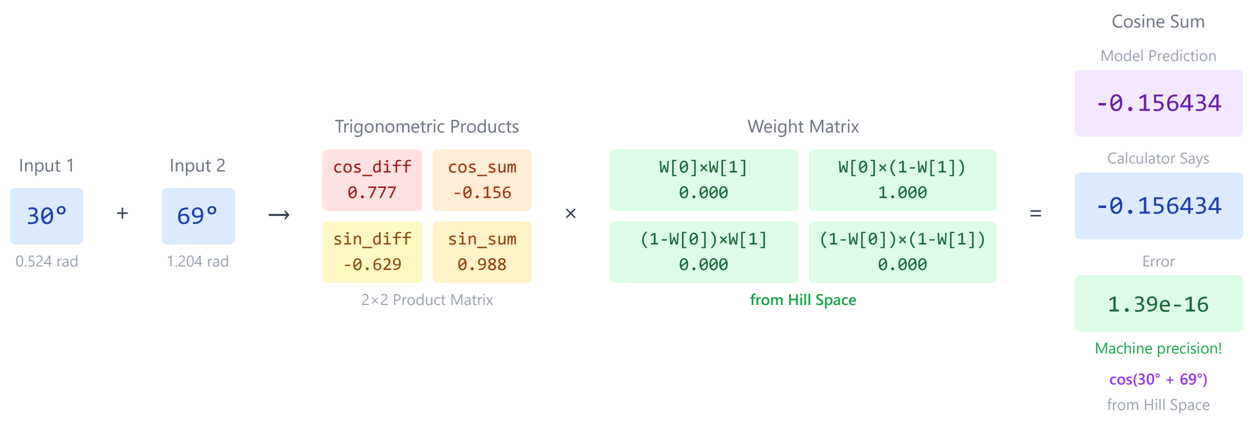
1. Input: 42° → 0.733 rad
2. Phase shift: 0.733 + 0.000 = 0.733
3. cos(0.733) = 0.743 , sin(0.733) = 0.669
4. Result: 0.743 × 1.000 + 0.669 × 0.000 = 0.743

```
def unit_circle_primitive(angle, weights):
    W = torch.tanh(weights[0]) * torch.sigmoid(weights[1])
    # Extract weights for selection and phase shift
    selector = W[0]  # [-1,1]: -1=sin, +1=cos, 0=mix
    phase_shift = W[1] * math.pi  # Phase shift in radians

    # Apply phase shift
    shifted_angle = angle + phase_shift

    # Compute unit circle components
    cos_comp = torch.cos(shifted_angle)
    sin_comp = torch.sin(shifted_angle)

    # Select component based on weight
    return (cos_comp * (1 + selector) + sin_comp * (1 - selector)) / 2
```

While excellent for basic trigonometric selections, this primitive struggles with compound binary operations, motivating the development of our trigonometric products primitive.

3.4 Trigonometric Products Primitive

The trigonometric product primitive demonstrates Hill Space's capability for complex discrete selection by computing four fundamental trigonometric products simultaneously, then using a 2×2 weight matrix to select between them. This enables compound operations requiring two angles: $\cos(\theta_1+\theta_2)$, $\sin(\theta_1+\theta_2)$, $\cos(\theta_1-\theta_2)$, and $\sin(\theta_1-\theta_2)$. Unlike the unit circle approach, this primitive excels at complex compound selections emerging from trigonometric product identities.



Step-by-Step Solution:

1. Compute trig components: cos(30°) = 0.866 , sin(30°) = 0.500
2. Compute trig components: cos(69°) = 0.358 , sin(69°) = 0.934
3. Compute four products: cos_diff = 0.777 , cos_sum = -0.156 sin_diff = -0.629 , sin_sum = 0.988
4. Apply weight matrix: 0.000 × 0.777 + 1.000 × -0.156 + 0.000 × -0.629 + 0.000 × 0.988 = -0.156434

```
def trigonometric_product_primitive(x, weights):
    cos1, sin1 = torch.cos(x[:, 0:1]), torch.sin(x[:, 0:1])
    cos2, sin2 = torch.cos(x[:, 1:2]), torch.sin(x[:, 1:2])

    # Four fundamental products
    cos_diff = cos1 * cos2 + sin1 * sin2  # cos(θ₁-θ₂)
    cos_sum = cos1 * cos2 - sin1 * sin2   # cos(θ₁+θ₂)
    sin_diff = sin1 * cos2 - cos1 * sin2  # sin(θ₁-θ₂)
    sin_sum = sin1 * cos2 + cos1 * sin2   # sin(θ₁+θ₂)

    # 2×2 matrix selection: selects cos vs sin and sum vs diff
    cos_component = weights[1] * cos_diff + (1 - weights[1]) * cos_sum
    sin_component = weights[1] * sin_diff + (1 - weights[1]) * sin_sum

    return weights[0] * cos_component + (1 - weights[0]) * sin_component
```

This primitive achieves high precision for compound operations, demonstrating Hill Space's effectiveness across different selection complexity levels.

3.5 Primitive Design Philosophy

Hill Space enables two distinct approaches to complex operations: sequential composition of simple selections and direct construction of sophisticated primitives. While sequential approaches like implementing multiplication through repeated addition demonstrate computational completeness, they introduce overhead that limits practical scalability.

The trigonometric product primitive exemplifies the superior alternative: batch-friendly discrete selection. Rather than computing compound operations through iterative manipulations, the primitive simultaneously calculates all fundamental components and uses Hill Space weights to select the desired combination. This approach achieves complex operations in a single forward pass.

4. Experiments

We conduct five experiments to validate Hill Space's theoretical properties and practical capabilities. Having established the constraint framework and primitive formulations, we first demonstrate the enumeration property through direct weight calculation, then show rapid convergence through a complete division implementation, benchmark performance against previous neural arithmetic approaches, characterize the fundamental precision limits achievable with our primitives, and systematically validate initialization robustness across all our mathematical primitives.

4.1 Direct Weight Enumeration

Before demonstrating Hill Space's training capabilities, we validate its fundamental enumeration property: optimal weights for discrete operations can be calculated rather than learned. This minimal implementation bypasses optimization entirely, directly setting Hill Space weights to their theoretically optimal saturation values.

The enumeration approach transforms primitive discovery from training-dependent exploration into direct mathematical calculation. Rather than hoping optimization converges to the correct discrete selections, we can immediately verify what operations are possible within Hill Space's constraint topology.

4.1.1 Complete Implementation

```
import sys
import numpy as np

class NeuralCalculator:
    def __init__(self):
        # Hill Space weights: tanh(15) ≈ 1.0, sigmoid(15) ≈ 1.0
        # After constraint W = tanh(W_hat) * sigmoid(M_hat):
        #   Addition/Multiply:  [15,  15] → [1.0,  1.0] → x + y / x * y
        #   Subtract/Division:  [15, -15] → [1.0, -1.0] → x - y / x / y
        self.weights = {
            "add": np.array([[15.0, 15.0], [15.0, 15.0]], dtype=np.float16),
            "sub": np.array([[15.0, -15.0], [15.0, 15.0]], dtype=np.float16),
            "mul": np.array([[15.0, 15.0], [15.0, 15.0]], dtype=np.float16),
            "div": np.array([[15.0, -15.0], [15.0, 15.0]], dtype=np.float16),
        }

    def compute(self, x, y, operation):
        """Neural computation with enumerated weights"""
        W_hat, M_hat = self.weights[operation]
        # Hill Space constraint: W = tanh(W_hat) * sigmoid(M_hat)
        W = np.tanh(W_hat) * (1 / (1 + np.exp(-M_hat)))
        inputs = np.array([x, y])
        if operation in ["add", "sub"]:
            return np.dot(inputs, W)  # Linear: x*w1 + y*w2
        else:  # mul, div
            return np.prod(np.power(inputs, W))  # Exponential: x^w1 * y^w2

def main():
    if len(sys.argv) != 4:
        print("Usage: python neural_calc.py <num1> <op> <num2>")
        sys.exit(1)
    x, op_symbol, y = float(sys.argv[1]), sys.argv[2], float(sys.argv[3])
    op_map = {"+": "add", "-": "sub", "x": "mul", "/": "div"}
    if op_symbol not in op_map or (op_symbol == "/" and y == 0):
        print(f"Invalid operation or division by zero")
        sys.exit(1)
    calc = NeuralCalculator()
    predicted = calc.compute(x, y, op_map[op_symbol])
    actual = {"add": x + y, "sub": x - y, "mul": x * y, "div": x / y}
[op_map[op_symbol]]
```

```python
    print(f"Neural: {x} {op_symbol} {y} = {predicted}")
    print(f"Truth:  {actual}")
    print(f"Error:  {abs(actual - predicted):.2e}")

if __name__ == "__main__":
    main()
```

NOTE: This conceptual implementation uses float16 weights and standard exponentiation for simplicity. For production precision, apply the Complex128 optimizations and snapping activations described in Sections 3.2 and 2.4.

4.1.2 Enumeration Validation

This implementation demonstrates Hill Space's core theoretical property: when we understand the constraint topology, optimal parameters become calculable. The saturated weights [±15, ±15] reliably produce discrete selections that approach [±1.0, ±1.0] asymptotically, reaching values like 0.999999+ that enable precise arithmetic operations.

Running the enumerated calculator on test cases confirms impressive precision across all operations:

- `python neural_calc.py 42.7 + 13.3` → Neural: 56.0, Truth: 56.0, Error: 0.00e+00
- `python neural_calc.py 156.8 / -23.4` → Neural: -6.701, Truth: -6.701, Error: 2.84e-14

The enumeration property enables systematic exploration of discrete selection spaces through direct weight manipulation rather than optimization-dependent training. This establishes the theoretical foundation that makes Hill Space's reliable convergence possible: the constraint topology creates stable attractors at precisely the locations we can calculate in advance.

Having validated that optimal weights can be enumerated directly, we now demonstrate that these same discrete selections emerge reliably through standard neural network training.

4.2 Learning Division in 60 Seconds

To demonstrate Hill Space's practical accessibility and rapid convergence, we present a complete implementation that teaches neural networks to select division reliably and consistently. This minimal example validates our theoretical framework while serving as an accessible entry point for practitioners. Training completes in under 60 seconds on consumer hardware without requiring GPU acceleration.

4.2.1 Complete Implementation

We provide the full implementation to ensure reproducibility and demonstrate that Hill Space's power emerges from fundamental simplicity, not architectural complexity.

```python
import torch

class Division(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.W_hat = torch.nn.Parameter(torch.zeros(2, 1))
        self.M_hat = torch.nn.Parameter(torch.zeros(2, 1))
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Exponential operation: x1^w1 * x2^w2"""
        W = torch.tanh(self.W_hat) * torch.sigmoid(self.M_hat)
        return torch.prod(torch.pow(x.unsqueeze(-1), W.unsqueeze(0)), dim=1)

def train_neural_division():
    # Setup: model, data, optimizer
    model = Division()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.3)
    loss_fn = torch.nn.MSELoss()
    # Goldilocks range: challenges precision without overwhelming gradients
    train_x = torch.rand(64000, 2) * (10.0 - 1e-8) + 1e-8
    train_y = train_x[:, 0:1] / train_x[:, 1:2]  # Division targets
    print("4 floating point numbers learning to induce division...")
    for epoch in range(50):
        for i in range(0, len(train_x), 64):  # batch_size = 64
            batch_x, batch_y = train_x[i:i+64], train_y[i:i+64]
            loss = loss_fn(model(batch_x), batch_y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        if epoch % 10 == 0:
            with torch.no_grad():
                full_loss = loss_fn(model(train_x), train_y)
                print(f"Epoch {epoch:2d}: Loss = {full_loss.item():.12f}")
    # Test on extreme extrapolation (far outside training range)
    test_cases = torch.tensor([[713.534, -0.13], [-0.252, -5244.0], [325751,
-161800]])
    test_pred = model(test_cases)
    test_true = test_cases[:, 0:1] / test_cases[:, 1:2]
    for i, (inputs, pred, actual) in enumerate(zip(test_cases, test_pred,
test_true)):
        a, b = inputs[0].item(), inputs[1].item()
        pred_val, actual_val = pred.item(), actual.item()
        mse = (pred_val - actual_val) ** 2
        status = "✅" if mse < 1e-4 else "❌"
        print(f"{a:.3f}/{b:.3f} = {pred_val:.6f} (true:{actual_val:.6f})
{status}")
    # Show learned weights approach [1.0, -1.0] for division
    final_weights = torch.tanh(model.W_hat) * torch.sigmoid(model.M_hat)
    print(f"\n🧠 Learned: {final_weights.flatten().tolist()}")
    print(f"Target: [1.0, -1.0]")

if __name__ == "__main__":
    train_neural_division()
```

4.2.2 Hill Space Advantages in Practice

This implementation demonstrates the optimization characteristics detailed in Section 2.1: aggressive learning rates, no data shuffling requirements, and no normalization. The division example showcases Hill Space's practical accessibility while validating the theoretical framework.

4.3 Comparison with iNALU

We compare our approach's performance against iNALU [Schlör et al., 2020], which provides clear experimental protocols with standardized MSE evaluation, enabling direct comparison of discrete selection convergence.

4.3.1 Experimental Setup

Datasets: Following iNALU protocol, we generate 64,000 samples for training and evaluation per operation. Each operation uses interpolation (same distribution) and extrapolation (different range) tasks.

Distributions Tested: Three distribution types with extrapolation scenarios:

- **U(-5,5)**: Trains on uniform[-5,5] and tests on uniform[-10,-5]
- **N(-3,3)**: Trains on normal(-3,3) and tests on normal(8,10)
- **E(0.8,0.5)**: Uses exponential distribution with parameters (0.8,0.5)

Universal Training Distribution: Our experiments also employ a "Goldilocks distribution" U(1e-8, 10.0) that avoids numerical instability near zero while preventing gradient saturation at extreme values. This range proves effective across all operations and primitives, enabling a single model to generalize across diverse test distributions without retraining.

Training Configuration:

- Optimizer: AdamWScheduleFree [Defazio et al., 2024] with lr=0.1 (vs iNALU's 0.001)

- Batch size: 64, Epochs: 10
- Loss: MSE, Auto stop threshold: MSE ≤ 10^{-10}
- No regularization, clipping, or reinitialization (unlike iNALU)
- Snapping functions used to ensure no error from activations

Evaluation Strategy: We conduct two complementary evaluations:

1. **Matched training**: Train models on each specific distribution (following iNALU exactly)
2. **Universal evaluation**: Train one model on Goldilocks distribution U(1e-8, 10.0), then test across all iNALU distributions without retraining

This tests whether our approach achieves discrete selection robustness across distributions.

4.3.2 Results

Our universal approach converged across all 10 seeds within 2 epochs, demonstrating remarkable training stability. Distribution-specific training showed more variability and occasional failures, highlighting the superior robustness of the Goldilocks distribution U(1e-8, 10.0) used in our universal training approach.

Table 4.3: Hill Space vs iNALU Performance (10 runs, Extrapolation MSE ± std)

Distribution	Operation	iNALU MSE	Matched MSE	Universal MSE
E(0.8,0.5)	a + b	2e-15 ± 3e-17	2e-14 ± 2e-16	2e-14 ± 2e-16
E(0.8,0.5)	a - b	1e-15 ± 2e-17	1e-14 ± 1e-16	1e-14 ± 1e-16
E(0.8,0.5)	a × b	1e-15 ± 6e-17	5e-14 ± 2e-15	5e-14 ± 2e-15
E(0.8,0.5)	a ÷ b	**362.4 ± 1e+03**	2e-12 ± 3e-13	2e-12 ± 3e-13
U(-5,5)	a + b	4e-13 ± 3e-15	2e-13 ± 2e-15	2e-13 ± 2e-15
U(-5,5)	a - b	9e-14 ± 5e-16	9e-14 ± 5e-16	9e-14 ± 5e-16
U(-5,5)	a × b	1e-10 ± 9e-13	**3e+03 ± 12.9**	7e-12 ± 3e-14
U(-5,5)	a ÷ b	**0.23 ± 0.34**	**0.20 ± 0.02**	2e-15 ± 1e-17
N(-3,3)	a + b	9e-13 ± 5e-15	5e-13 ± 2e-15	5e-13 ± 2e-15
N(-3,3)	a - b	2e-13 ± 8e-16	1e-11 ± 7e-12	2e-13 ± 9e-16
N(-3,3)	a × b	3e-10 ± 2e-12	**814.6 ± 2e+03**	1e-11 ± 7e-14
N(-3,3)	a ÷ b	**2.7 ± 4.1**	**96.7 ± 305.4**	3e-15 ± 2e-17

Note: Results averaged over 10 runs. Values in bold indicate degraded performance (MSE > 1e-2).

4.3.3 Discussion

The results demonstrate our approach's advantages across key performance metrics. The universal model achieves sub-10^{-10} precision on all operations while requiring less than 5% of iNALU's maximum training time (2 ± 0.5 out of 100 epochs) to converge with snapping activations. Without snapping, convergence occurs within 8 epochs—demonstrating that Hill Space naturally guides optimization toward discrete selections, with the primary bottleneck being the asymptotic nature of activation functions rather than finding correct weights. Most notably, universal training consistently outperforms distribution-specific training across all tested operations and distributions.

4.4 Error Analysis and Attribution

To validate that Hill Space approaches fundamental precision limits, we quantify errors using analytically perfect weights, isolating floating-point precision from implementation artifacts. We analyzed 800 million operations against 50-digit precision Decimal ground truth to characterize precision boundaries with statistical robustness.

Our analysis proceeds in two stages: first establishing the theoretical lower bounds for floating-point arithmetic errors (Table 4.4.1), then measuring additional error introduced by different stabilization methods for exponential primitives (Table 4.4.2). This second analysis reveals that while Complex128 arithmetic adds negligible error (4.1e-24 for division), the log-space stabilization approach suffers catastrophic precision loss with errors exceeding 10^15—possibly explaining iNALU's [Schlör et al., 2020] convergence failures on division tasks.

Table 4.4.1: Floating-Point Precision Baseline
100M+ samples per operation/dtype, analytical weights vs high-precision ground truth

Operation	Precision	Mean Squared Error	Max Error	99.99%ile Error
add	Float32	5.25e-08	9.54e-07	9.54e-07
add	Float64	2.0e-25	1.3e-23	1.3e-23
subtract	Float32	5.25e-08	9.54e-07	9.54e-07
subtract	Float64	2.9e-25	1.3e-23	1.3e-23
multiply	Float32	7.38e-01	1.60e+01	1.59e+01
multiply	Float64	7.9e-18	8.9e-16	2.2e-16
divide	Float32	6.64e-08	4.27e+00	2.20e-08
divide	Float64	3.4e-26	8.7e-19	5.2e-26

This table establishes the theoretical lower bound for floating-point arithmetic errors.

Table 4.4.2: Exponential Primitive Stabilization Errors
100M+ samples per operation/dtype, showing additional error each method introduces beyond floating-point baseline

Operation	Precision	Method	Additional MSE	Max Error	99.99%ile Error
multiply	Float32	Complex128	0.0	1.60e+01	1.59e+01
multiply	Float32	Log-space	1.11e+15	1.02e+16	9.80e+15
multiply	Float64	Complex128	5.8e-16	3.75e-14	2.22e-14
multiply	Float64	Log-space	1.11e+15	1.02e+16	9.80e+15
divide	Float32	Complex128	-9.6e-09	3.73e+00	1.19e-08
divide	Float32	Log-space	3.21e+07	1.54e+15	2.51e+07
divide	Float64	Complex128	4.1e-24	1.2e-16	3.3e-24
divide	Float64	Log-space	3.21e+07	1.54e+15	2.51e+07

Methodology:

- Ground truth computed with 50-digit precision Decimal arithmetic
- Additional error = Method MSE - Analytical MSE (floating-point baseline)
- Input range: U(-1e4, 1e4) matching extreme extrapolation experiments
- Complex128 uses 128-bit complex arithmetic (two 64-bit floats)
- Log-space uses iNALU-style log/exp transformations with stability clamping

These results demonstrate that Complex128 arithmetic approaches theoretical optimality: Float64 multiplication shows 5.8e-16 additional error (machine epsilon territory), while log-space methods exhibit catastrophic 10^15 error amplification. Hill Space with Complex128 reaches the fundamental limits of floating-point computation.

4.5 Weight Initialization Analysis

To validate Hill Space's initialization robustness across all operations and primitives, we conducted systematic evaluation across different initialization scales. **Table 4.5** demonstrates this initialization sensitivity across mathematical operations spanning arithmetic and trigonometry. Models were trained for 10 epochs on the Goldilocks distribution U(1e-8, 10.0) with learning rate 0.1, AdamWScheduleFree [Defazio et al., 2024] optimizer, and batch size 64, then evaluated on extreme extrapolation range U(-1e4, 1e4). Snapping functions used to ensure no error from activations.

Table 4.5: Impact of Weight Initialization (10 runs, Extrapolation MSE)

Operation	0	0.02	1e-08	1.0	3.0	10.0
a + b	0.0	0.0	0.0	0.0	0.0	**1e+07** ± 1e+07
a - b	0.0	0.0	0.0	0.0	0.0	**2e+07** ± 2e+07
a × b	6e-16 ± 7e-18	6e-16 ± 7e-18	6e-16 ± 7e-18	6e-16 ± 7e-18	6e-16 ± 7e-18	**4e+14** ± 5e+14
a ÷ b	0.0	0.0	0.0	0.0	0.0	**954.6** ± 1e+03
a	0.0	0.0	0.0	0.0	0.0	**7e+06** ± 8e+06
1/a	3e-07 ± 7e-07	2e-07 ± 6e-07	2e-07 ± 8e-07	7e-08 ± 2e-07	5e-08 ± 8e-08	**0.20** ± 0.39
cos(θ)	0.0	0.0	0.0	0.0	3e-13 ± 5e-13	**0.23** ± 0.27
sin(θ)	4e-09 ± 2e-09	4e-09 ± 2e-09	4e-09 ± 2e-09	1e-05 ± 3e-05	**0.30** ± 0.17	**0.33** ± 0.19
$\cos(\theta_1+\theta_2)$	2e-10 ± 1e-10	2e-10 ± 1e-10	2e-10 ± 1e-10	2e-09 ± 2e-09	2e-08 ± 3e-08	**0.35** ± 0.58
$\sin(\theta_1+\theta_2)$	2e-14 ± 2e-14	2e-14 ± 1e-14	2e-14 ± 1e-14	2e-10 ± 1e-10	6e-09 ± 4e-09	**0.10** ± 0.35
$\cos(\theta_1-\theta_2)$	2e-10 ± 1e-10	2e-10 ± 1e-10	2e-10 ± 8e-11	8e-12 ± 9e-12	8e-12 ± 1e-11	**0.31** ± 0.30
$\sin(\theta_1-\theta_2)$	2e-10 ± 8e-11	2e-10 ± 1e-10	1e-10 ± 7e-11	2e-09 ± 1e-09	5e-08 ± 4e-08	**0.36** ± 0.32

Note: Values in bold indicate degraded performance. Values shown as 0.0 are below machine epsilon (< 1e-16) and displayed for clarity.

The results confirm Hill Space's robust initialization properties: excellent performance across all 10 seeds with reasonable initialization scales (0 to 1.0), with degradation only at extreme scales (10.0) that effectively randomly pre-select operations before training begins.

4.6 Reproducibility

All code and interactive demonstrations are available at:

- GitHub: https://github.com/justindujardin/hillspace
- Interactive demos: https://hillspace.justindujardin.com

The repository includes implementations of all primitives, training scripts reproducing the experimental results, code for generating the paper figures, and interactive primitive widgets used on the site.

5. Related Work

5.1 Neural Arithmetic Logic Units

Neural Arithmetic Logic Units (NALU) [Trask et al., 2018] first demonstrated that neural networks could achieve systematic discrete selection through constrained weight parameterizations. NALU introduced the key insight of constraining weights via **$W = \tanh(\hat{W}) \odot \sigma(\hat{M})$** to approximate discrete values in {-1, 0, +1}, enabling reliable extrapolation far beyond training ranges. However, NALU suffered from training instability, particularly for division operations, and could not handle negative inputs in multiplicative operations due to log-space computation.

Improved NALU (iNALU) [Schlör et al., 2020] addressed these limitations through separate weight matrices for different operations, mixed-sign multiplication capabilities, and enhanced regularization strategies. **Neural Arithmetic Units (NAU/NMU)** [Madsen & Johansen, 2020] took a modular approach with specialized units for different operations, achieving more reliable convergence than the combined NALU architecture.

5.2 Alternative Approaches to Structured Neural Selection

Beyond explicit arithmetic modules, researchers have explored **translation-based methods** that treat discrete selection tasks as language translation problems [Lample & Charton, 2019], achieving remarkable success on symbolic integration tasks. **Graph Neural Networks** like GraphMR [Zhang et al., 2021] represent structured relationships to capture selection dependencies. **Meta-learning approaches** [Lake et al., 2023] have demonstrated human-like systematic generalization through compositional optimization procedures.

5.3 Systematic Generalization

The broader challenge of **systematic generalization** in neural networks has been extensively studied [Lake & Baroni, 2018; Fodor & Pylyshyn, 1988]. Recent surveys [Testolin, 2024] conclude that even state-of-the-art architectures struggle with systematic extrapolation and compositional reasoning, highlighting the continued importance of specialized approaches for discrete selection tasks.

Our work builds directly on NALU's constraint insight while systematically analyzing the underlying topology that enables reliable discrete selection. Unlike previous approaches that focused on architectural improvements, we isolate and characterize the constraint space itself, enabling principled exploration of discrete selection primitives across different domains.

6. Conclusion

We set out to understand why neural networks struggle with arithmetic and discovered a systematic framework for discrete selection in constrained parameter spaces. Hill Space—the constraint topology $W = \tanh(\hat{W}) \odot \sigma(\hat{M})$—transforms how we approach problems requiring reliable convergence to specific discrete outcomes.

Our key contributions:

Enumeration Property: Optimal weights for discrete operations can be calculated rather than learned, transforming primitive discovery from training-dependent exploration to direct mathematical calculation.

Exponential Primitive Stabilization: Complex128 arithmetic eliminates catastrophic precision loss in exponential operations (reducing error by 15 orders of magnitude compared to log-space methods), enabling reliable multiplication and division at machine precision.

The philosophical question of what constitutes "doing math" versus "selecting mathematical transformations" becomes central to understanding Hill Space. If humans don't actually perform multiplication through repeated addition but instead select the appropriate transformation directly, Hill Space may reflect more fundamental cognitive patterns than initially apparent.

We've demonstrated that appropriate constraint design combined with systematic exploration can guide neural networks toward specific discrete capabilities. Hill Space offers a methodology for finding more such capabilities—in mathematics and potentially beyond.

7. Acknowledgments

We thank the broader machine learning community for foundational work in systematic neural selection, particularly the NALU team at DeepMind whose pioneering insights enabled this research direction. We also appreciate Daniel Schlör, Markus Ring, and Andreas Hotho for their iNALU evaluation, whose clearly documented experimental setup enabled meaningful performance benchmarking.

Special thanks to Claude (Anthropic) for invaluable assistance in theoretical exposition, helping clarify complex topological concepts and constraint optimization principles that significantly improved the presentation of this work. Claude also contributed to the implementation of the visualizations.

We also acknowledge the open-source community for tools and frameworks that made rapid experimentation possible, enabling the systematic exploration of discrete selection spaces described in this work.

8. References

Defazio, A., Yang, X. A., Mehta, H., Mishchenko, K., Khaled, A., & Cutkosky, A. (2024). The Road Less Scheduled. arXiv preprint arXiv:2405.15682.

Feng, W., Liu, B., Xu, D., Zheng, Q., & Xu, Y. (2021). GraphMR: Graph neural network for mathematical reasoning. Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 3077-3088.

Fodor, J. A., & Pylyshyn, Z. W. (1988). Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1-2), 3-71.

Lake, B. M., & Baroni, M. (2018). Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. *Proceedings of the 35th International Conference on Machine Learning*, 2873-2882.

Lake, B. M., & Baroni, M. (2023). Human-like systematic generalization through a meta-learning neural network. Nature, 623(7985), 115-121.

Lample, G., & Charton, F. (2019). Deep learning for symbolic mathematics. *arXiv preprint arXiv:1912.01412*.

Madsen, A., & Johansen, A. R. (2019). Measuring arithmetic extrapolation performance. arXiv preprint arXiv:1910.01888.

Madsen, A., & Johansen, A. R. (2020). Neural arithmetic units. *arXiv preprint arXiv:2001.05016*.

Schlör, D., Ring, M., & Hotho, A. (2020). iNALU: Improved neural arithmetic logic unit. *Frontiers in Artificial Intelligence*, 3, 71.

Testolin, A. (2024). Can neural networks do arithmetic? A survey on the elementary numerical skills of state-of-the-art deep learning models. *Applied Sciences*, 14(2), 744.

Trask, A., Hill, F., Reed, S. E., Rae, J., Dyer, C., & Blunsom, P. (2018). Neural arithmetic logic units. *Advances in Neural Information Processing Systems*, 31.