A TECHNICAL REPORT

Stable Because Stuck: NALU Selects, It Doesn't Compute

Justin DuJardin · DuJardin Consulting, LLC · justin@dujardinconsulting.com

supersedes 10.36227/techrxiv.175339930.03949307/v2

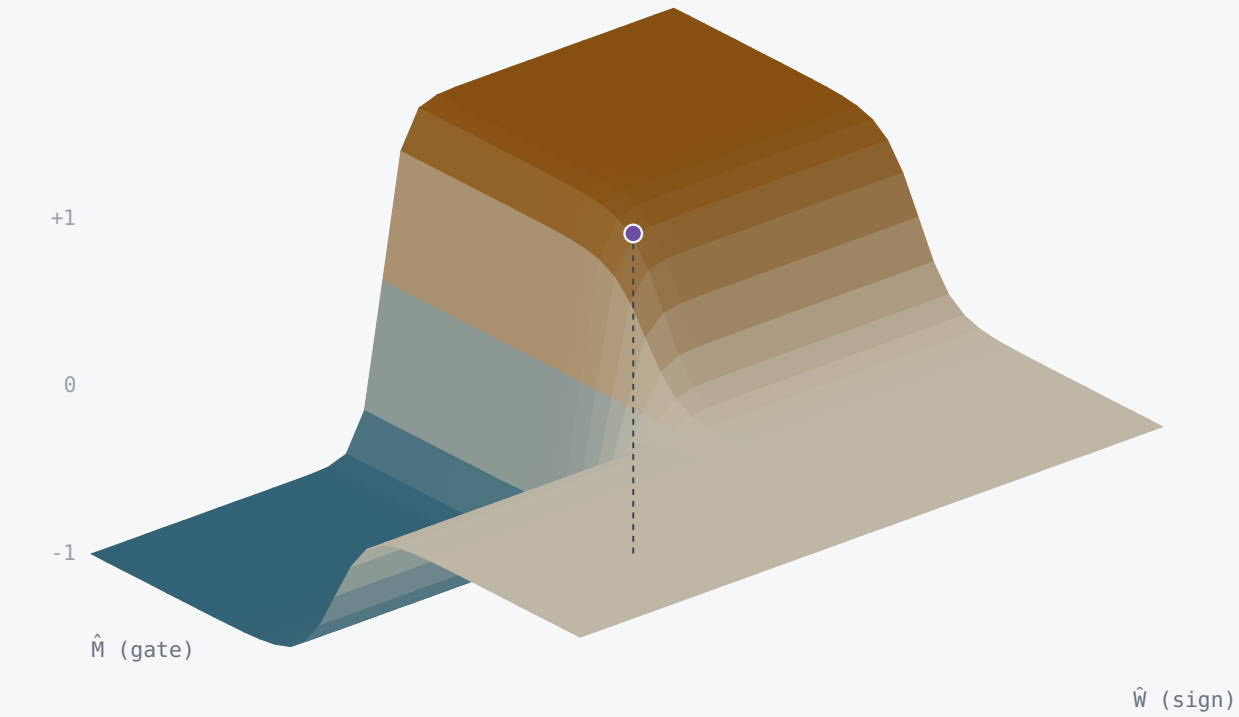
2026-09-08

ABSTRACT I characterize Hill Space, the parameter space created by NALU's weight construction $W = \tanh(\hat{W}) \odot \sigma(\hat{M})$ (Trask et al., 2018), and document the mechanism that makes it work: the discrete targets -1, 0, $+1$ sit exactly where the constraint's own gradient vanishes, so a weight that reaches a target stops moving. The same vanishing that makes a converged selection exact is what starves one that hasn't arrived. Working from that, I resolve NALU's division instability with a training distribution, reach exact discrete weights across a broad sweep of stock optimizers with a snapping activation, build two trigonometric primitives on the same constraint, and measure error at the floating-point floor. Once a unit selects its operation exactly, extrapolation stops being a property of the model and becomes a property of the number format. NALU does not compute math; it selects it.

CONTENTS

1. Introduction . . . 2
2. Hill Space: The Constraint . . . 3
2.1 Saturation Fixed Points . . . 4
2.2 Two Paths to Exact Saturation . . . 5
2.3 Unstable Weights . . . 9
2.4 Input Scaling . . . 9
2.5 Enumeration . . . 9
2.6 Scope . . . 9
3. Primitives . . . 10
3.1 Additive Primitive . . . 10
3.2 Exponential Primitive . . . 11
3.3 Unit Circle Primitive . . . 12
3.4 Trigonometric Products Primitive . . . 13
4. Experiments . . . 14
4.1 Direct Weight Enumeration . . . 14
4.2 Learning Division Quickly . . . 15
4.3 Comparison with iNALU . . . 16
4.4 Error Analysis and Attribution . . . 18
4.5 Weight Initialization Analysis . . . 20
4.6 Reproducibility . . . 21
5. Related Work . . . 22

6. Conclusion 23
7. Contributions 23
8. Acknowledgments 23
References 24
Appendix: code listings 25



Plateaus: W = +1 amber, −1 teal, 0 gray. The readout gives W and both derivatives at the marker. On the web, spin the hill and move the marker.

1. Introduction

I was building RL agents to solve math problems with step-by-step explanations when I came across NALU. My agents were good at solving problems by modifying tree structures, but they relied on a calculator action for computing values. I thought NALU could be a

useful auxiliary task to imbue some number sense. I already had a robust auxiliary task setup, so it felt like this might really add depth to my agents' learned policies.

After a few experiments, the training dynamics started to nag at me. It's so near perfect, but only sometimes. What's up with that? I woke up a few weeks later, in a sweat, with more than a few variants of models trying to stabilize the arithmetic task. RL and algebra problems were out the window; I had to know what was going on.

This report details what I found. Section 2 characterizes the parameter space and the mechanism behind everything hard about it. Section 3 analyzes NALU's two primitives and builds two more on the selection principle. Section 4 measures to check for mistakes: do enumerated weights work, can division be learned quickly and reliably, how does it compare to existing works, what's the remaining error made of, and does changing initialization break things? Section 5 maps out the path that led through a wasteland of attempts to tame saturation: regularization, reinitialization, or abandonment. The view is great up here on the giants' shoulders.

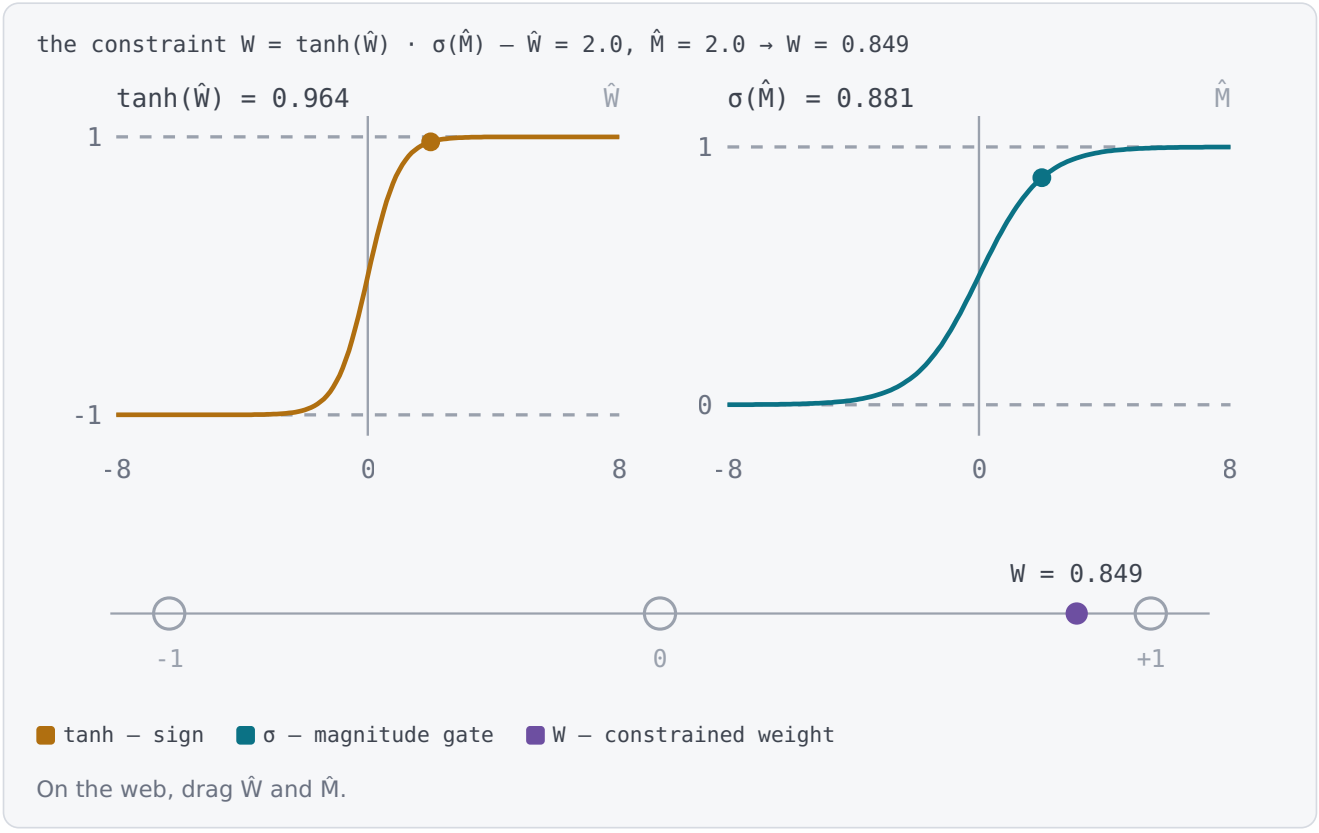
---

2. Hill Space: The Constraint

$$W = \tanh(\hat{W}) \odot \sigma(\hat{M})$$

If I was going to have any chance of understanding why NALU was failing, I'd need to start from the beginning. The NALU authors say the constraint "produces matrices whose elements are guaranteed to be in $[-1, 1]$ and biased to be close to -1, 0, or 1" [1] which is a very cool property to have while remaining differentiable.

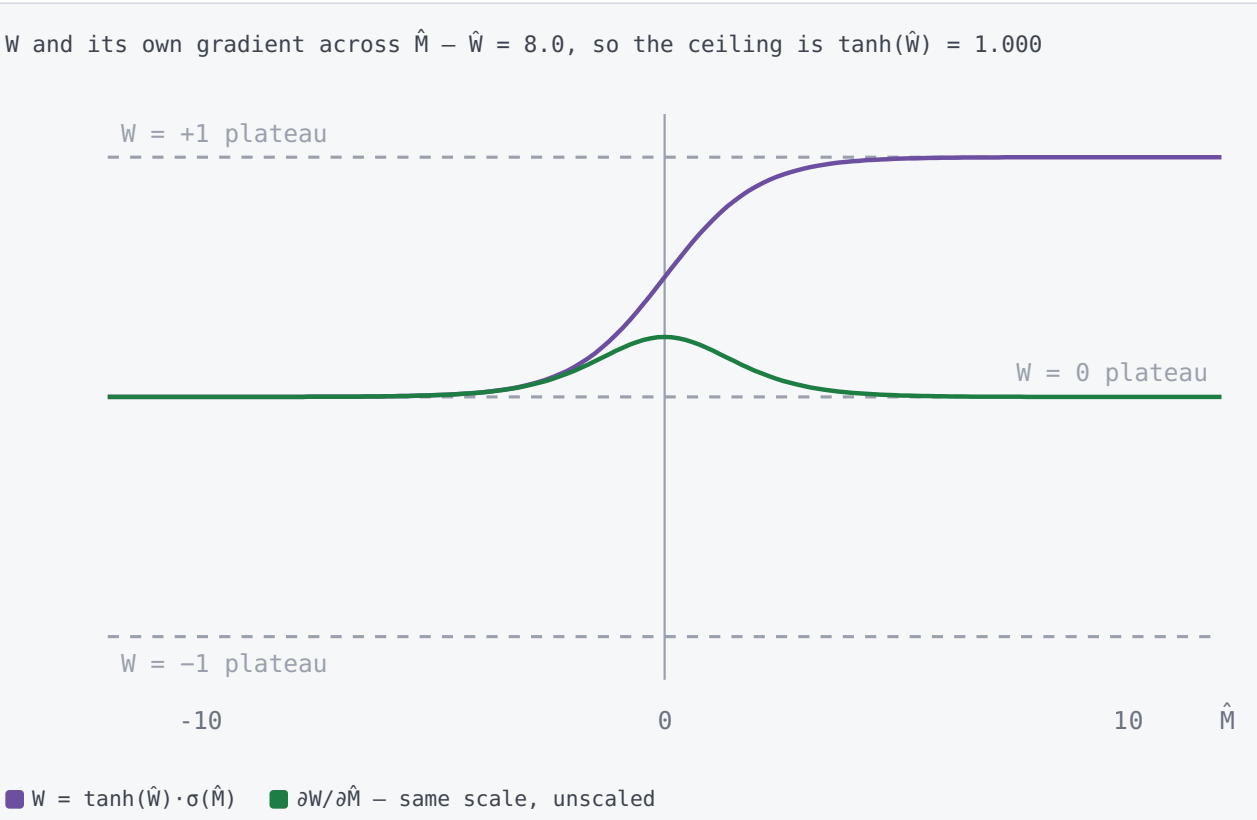
So this constraint creates a parameter space where tanh bounds weights to between -1 and 1, and sigmoid to between 0 and 1: the tanh controls the sign, and the sigmoid gates the magnitude.



The authors of NALU didn't name the constraint function, so I term this parameter space "Hill Space" in recognition of Felix Hill's contributions to neural arithmetic through NALU [1], and for the hill-like shape of the weight W when plotted over (Ŵ, M̂).

2.1 Saturation Fixed Points

Trask et al. (2018) note that "the stable points $\{-1, 0, 1\}$ correspond to the saturation points of either σ or tanh" (fn. 1). This observation ends up holding the key to understanding Hill Space: because saturation is exactly where both partial derivatives of the constraint, $\partial W/\partial \hat{W} = (1 - \tanh^2(\hat{W}))\,\sigma(\hat{M})$ and $\partial W/\partial \hat{M} = \tanh(\hat{W})\,\sigma(\hat{M})\,(1 - \sigma(\hat{M}))$, approach zero, a weight that reaches one of these points stops moving:



The gradient is drawn on the same scale as W. It peaks mid-slope and fades toward every plateau; there is no particular $\hat{M}$ where runs get stuck.

It's this one small fact that drives most of the interesting things about Hill Space. It's the reason that a converged weight usually stays put, and it's also why there's so much variance and failure in training these circuits. If your optimizer carries you deep enough into saturation as your gradients vanish, you get exact target values thanks to floating-point rounding, and if it doesn't you end up stuck short of the target.

2.2 Two Paths to Exact Saturation

Not all targets cost the same. A 0 weight is easily reached with a single parameter, as $\sigma(\hat{M})$ shrinks toward 0 once $\hat{M}$ is driven far enough negative. A weight of ±1 is different because $\tanh(\hat{W})\cdot\sigma(\hat{M})$ only rounds to exactly 1.0 once both parameters are deep in saturation. Measured with torch on CPU, the smallest value at which each returns exactly 1.0:

Precision	ε	$\tanh(\hat{W}) = 1.0$	$\sigma(\hat{M}) = 1.0$
float16	9.8e-04	$\hat{W} \geq 4.51$	$\hat{M} \geq 8.32$
float32	1.2e-07	$\hat{W} \geq 9.02$	$\hat{M} \geq 16.64$
float64	2.2e-16	$\hat{W} \geq 19.07$	$\hat{M} \geq 36.74$

Gradient vanishing is where optimizers fall down. Standard Adam ($\beta_2 = 0.999$) keeps a second-moment estimate average over roughly a thousand steps; near the flat optimum its estimate stays large from earlier gradients, so the step collapses and the weight stops short of saturation.

To see how this impacts performance, I sweep ten standard optimizer configurations at default learning rates: Adam [2] and its AdamW [3], RAdam [4], and NAdam [5] variants; RMSProp [6]; Adagrad [7]; Adadelta [8]; Rprop [9]; and SGD with momentum [10]. To cut down on redundant tables, I add one column as an exception without default configuration: Adam is shown also with $\beta_2 = 0.5$, foreshadowing a solution to the second-moment stall from its defaults.

measured extrapolation MSE ($\log_{10}$) by operation and optimizer

plain hill, no snapping

	Adam	Adam β_2=0.5	AdamW	RAdam	NAdam	RMSProp	Adagrad	Adadelta	Rprop	SGD+mom
a+b	-6	-25	-3	-6	-6	-6	3	1	-25	-25
a−b	-5	-25	-2	-5	-5	-6	3	1	-25	-1
a×b	1	-15	5	1	1	1	11	8	-15	-15
a÷b	-11	-28	-8	-11	-11	-12	-1	-5	-28	-28
a	-5	-23	-2	-5	-5	-5	4	2	-23	-14
1/a	-22	-36	-7	-22	-22	-22	-8	-12	-36	-36
sin θ	-12	-35	-9	-12	-12	-12	-6	-5	-17	-6
cos(θ_1+θ_2)	-12	-25	-9	-12	-12	-12	-5	-6	-25	-6

Dark slate is the floating-point floor, paler slate stopped short of it, sand and brick failed (MSE above 1e-2).

Whether a run reaches the floor depends on optimizer internals that almost no task cares about, precisely because Hill Space exactness requires convergence to a discrete weight value. The two approaches I find that work are:

Drive into saturation. Find a way to keep moving while gradients vanish, e.g. you can stay in the Adam family with $\beta_2 = 0.5$ that tracks the shrinking gradient and keeps moving further into saturation.

Snap the activation. Let the optimizer stop wherever it's comfortable, and snap near-saturated activations to their exact values. This works because a weight parked at 0.99999999 is a clear "I found the selection" signal.

```python
def snapping_tanh(x, precision_threshold=1e-2):
    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-2):
    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
```

Activation snapping is safe because stable selections are sparse: I swept thresholds and found a working range between roughly 3e-3 and 5e-2. Converged optimizers land within about 2e-3 of a target, while fractional non-targets like 0.5 sit half a unit away and are never grabbed. I apply snapping at evaluation unless stated otherwise; applied during training it also works and sometimes converges faster, but I haven’t investigated it thoroughly enough to list the tradeoffs.

measured extrapolation MSE ($\log_{10}$) by operation and optimizer

eval snap, threshold 1e-6

	Adam	Adam β_2=0.5	AdamW	RAdam	NAdam	RMSProp	Adagrad	Adadelta	Rprop	SGD+mom
a+b	-25	-25	-3	-25	-25	-25	3	1	-25	-25
a−b	-25	-25	-2	-25	-25	-25	3	1	-25	-1
a×b	-15	-15	-15	-15	-15	-15	11	8	-15	-15
a÷b	-28	-28	-28	-28	-28	-28	-1	-5	-28	-28
a	-23	-23	-2	-23	-23	-23	4	2	-23	-23
1/a	-36	-36	-7	-36	-36	-36	-8	-12	-36	-36
sin θ	-13	-35	-9	-13	-13	-13	-6	-5	-18	-6
cos(θ_1+θ_2)	-13	-25	-9	-13	-13	-13	-5	-6	-25	-6

eval snap, threshold 1e-2

	Adam	Adam β_2=0.5	AdamW	RAdam	NAdam	RMSProp	Adagrad	Adadelta	Rprop	SGD+mom
a+b	-25	-25	-25	-25	-25	-25	-25	-25	-25	-25
a−b	-25	-25	-25	-25	-25	-25	-25	-25	-25	-25
a×b	-15	-15	-15	-15	-15	-15	-15	-15	-15	-15
a÷b	-28	-28	-28	-28	-28	-28	-28	-28	-28	-28
a	-23	-23	-23	-23	-23	-23	-23	-23	-23	-23
1/a	-36	-36	-7	-36	-36	-36	-36	-36	-36	-36
sin θ	-35	-35	-35	-35	-35	-35	-6	-5	-35	-35
cos(θ_1+θ_2)	-25	-25	-25	-25	-25	-25	-6	-6	-25	-6

Dark slate is the floating-point floor, paler slate stopped short of it, sand and brick failed (MSE above 1e-2).

At 1e-6, where the Adam family parks, snapping rescues the weights that stall there but not much more. At 1e-2, almost all optimizers hit the floor, and the paths converge. Whether you use an optimizer that keeps moving or snap your activatations, they both resolve to the same floors. I find that snapping at evaluation is the most practical form because it demands the least of the weights while providing reliable selections. If you prefer to keep it simple, most optimizers that can keep moving in the face of vanishing gradients should work equally well.

One prior system gets close to this. iNALU's regularizer [11] penalizes any parameter with $|\hat{W}|$ or $|\hat{M}|$ below t = 20. At 20, tanh already rounds to exactly 1.0 in float64, but $\sigma(20)$ is just shy: deep saturation, not exactness. A snap at evaluation is the step that remains.

2.3 Unstable Weights

It's not all rainbows and sunshine in Hill Space. When your objective requires weights that don't correspond to a saturation point of the constraint, the rest of Hill Space becomes hard to navigate. For example, the exponential primitive (Section 3.2) cannot stably represent the Square Root or Cube Root operations because they require fractional weight configurations (0.5, 0.33) that live where the gradients are strongest. Tiny optimization nudges move your weight around the target, but rarely land directly on it.

2.4 Input Scaling

Hill Space constrains weights, not inputs. Without care, large input ranges (e.g., `U(-10000, 10000)`) lead to large gradient magnitudes that have to be managed or can explode during training. The fix is constraining the training distribution itself. The "Goldilocks distribution" U(1e-8, 10.0) keeps gradients bounded, stays away from zero so division targets stay finite, and stays positive. Positive inputs are the case where the exponential primitive trains without trouble; the two zero-crossing ranges in Section 4.3 are where its matched runs fail, and I haven't worked out why. The trigonometric primitives don't care about sign. Models trained on this range generalize to any range that the number format has enough precision to carry.

Once the unit has selected its operation exactly, there is nothing left for a new input range to break. Extrapolation stops being a concern, because it's guaranteed by construction of the primitive.

2.5 Enumeration

It was a callout in the original NALU paper, and is perhaps obvious to the math wizards out there, but took me a while to appreciate. The Hill Space constraint maps saturation values to roughly {1, 0, -1} in a way that is strongly biased. But what does it mean for this optimization problem? It means that I can directly explore hypothetical future operations, without optimization. Since discrete operations require specific selections, their optimal weights can be calculated rather than learned.

Enumeration has been known to the field for some time: Madsen and Johansen (2020) hand-build perfect weights as evaluation baselines, and Mistry et al. (2022) define interpretability as exactly the ability to set a module's parameters provably. I demonstrate this in Section 4.1 with a neural calculator that uses enumerated weights without any training.

2.6 Scope

Hill Space is for discrete selection. The constraint does not perform computation, it picks *which* transformation to apply; the computation is always provided by the primitive formulation. Everything in this report is single units at small scale, where each primitive

has a surface you can plot directly. That's why I was able to identify the mechanism at all. How these compose, and how they behave inside a larger circuit, I don't have a good grasp on.

I have one guess, and it's only that: I expect composition to depend more on formulation than on the constraint. The vanishing in Section 2.1 applies per weight, so a circuit of these units should lock in piecewise during training. Whether that leads to something useful or just strands the circuit in whatever order it converged, I don't know.

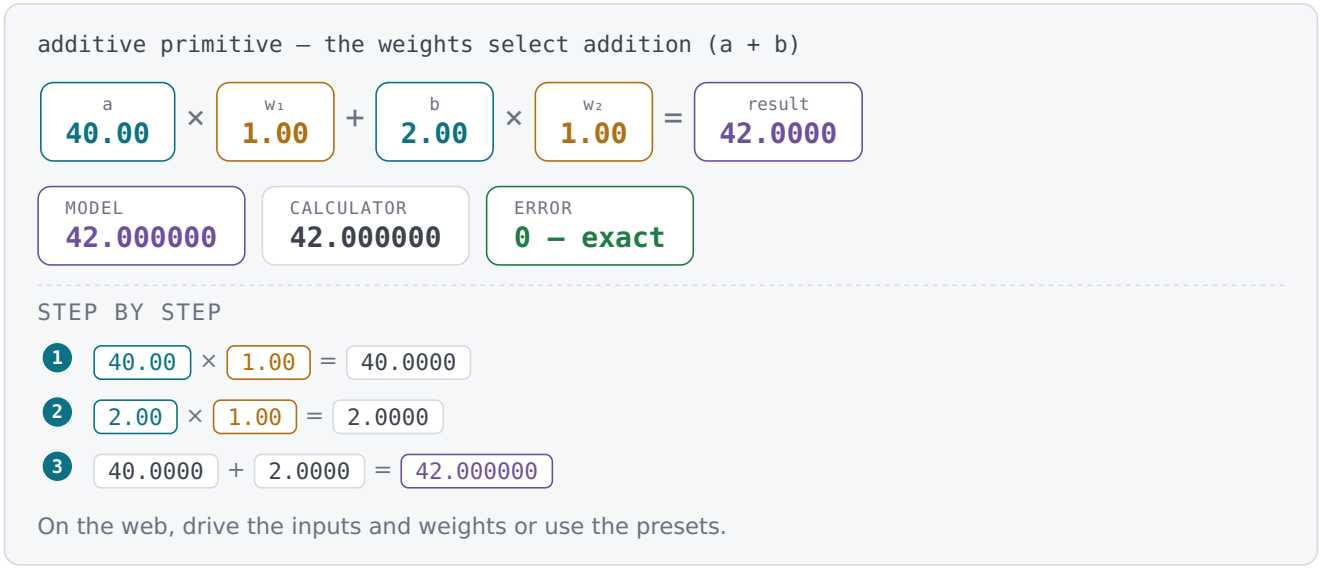
---

3. Primitives

At this point in the story I still did not understand enumeration. When my additive and exponential primitives started producing exact results, I spent a long time staring at the learned weights, trying to make sense of them. They were never the same twice, yet they computed the same perfect math. How does exact arithmetic come out of raw weights [13, 12] on one run and [23, 17] on the next? The lightbulb was still weeks away. These are the primitives I was staring at: NALU's additive and exponential pair [1], and once I understood what those two were actually doing, two new trigonometric primitives to confirm my understanding of the first ones. It was the act of comparing the learned weights between arithmetic and trig primitives that clued me in to enumeration.

3.1 Additive Primitive

The additive primitive uses matrix multiplication for linear operations. Four obvious stable selections emerge: addition [1,1], subtraction [1,-1], identity [1,0], and negation [-1,0]. The key here is the implicit addition in matrix multiplications.



```python
def additive_primitive(x, weights):
    W = torch.tanh(weights[0]) * torch.sigmoid(weights[1])
    return torch.matmul(x, W)
```

3.2 Exponential Primitive

The exponential primitive computes $a^{w_1} \times b^{w_2}$. Four selections have proven stable and learnable: multiply [1,1], divide [1,-1], identity [1,0], and reciprocal [-1,0]. With weights [1,-1], this becomes $a^1 \times b^{-1} = a / b$ and division emerges from the negative exponent.

Other exponential operations such as powers and roots exist with the same precision, but remain unstable for reliable learning (Section 2.3).

exponential primitive — the weights select multiplication (a × b)

a **6.00** ^ w_1 **1.00** × b **7.00** ^ w_2 **1.00** = result (Re) **42.0000**

MODEL	CALCULATOR	ERROR
42.000000	42.000000	0 — exact

STEP BY STEP

1. 6.00 ^ 1.00 = 6.0000
2. 7.00 ^ 1.00 = 7.0000
3. 6.0000 × 7.0000 = 42.000000

An imaginary residue appears only when a negative base meets a fractional weight.

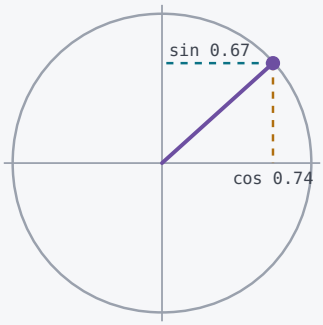
```python
def exponential_primitive(x, weights):
    W = torch.tanh(weights[0]) * torch.sigmoid(weights[1])
    # Convert to complex128 to handle negative bases with fractional exponents
    x_complex = x.to(torch.complex128)
    result = torch.prod(torch.pow(x_complex, W.unsqueeze(0)), dim=1)
    return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

The complex-logarithm route into negative bases originates with Neural Power Units [14], who use complex weights with a closed-form real output. I keep NALU's real, constrained parameterization and move the complex insight into the arithmetic: evaluating x^w natively in complex128 turns a negative base under a fractional exponent into a rotation instead of a NaN. Section 4.4 shows this introduces only 5.8e-16 additional MSE on Float64 multiplication.

3.3 Unit Circle Primitive

The unit circle primitive projects an input angle 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.

unit-circle primitive — the weights select cos(θ)



θ 42° + phase w₂·π 0.00 → θ′ 0.733 rad → output 0.7431

selector w₁ 1.00 → (1+w₁)/2 · cos 0.7431 + (1−w₁)/2 · sin 0.0000

MODEL	CALCULATOR	ERROR
0.743145	0.743145	0 — exact

STEP BY STEP

1. θ 42° = 0.733 rad + phase 0.000 = θ′ 0.733
2. cos θ′ 0.7431 , sin θ′ 0.6691
3. 1.00 × 0.7431 + 0.00 × 0.6691 = 0.743145

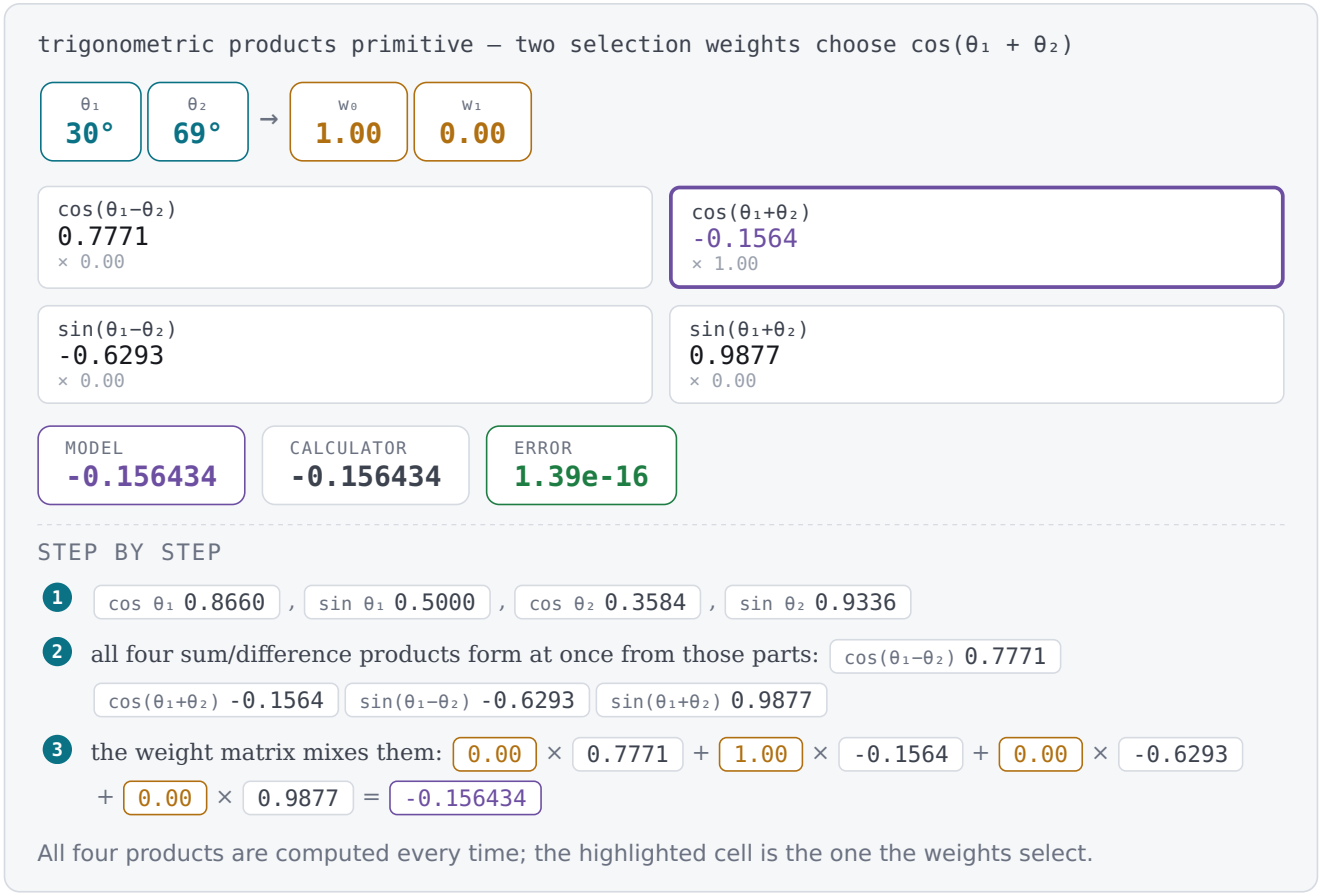
The dashed drops are cos and sin of the shifted angle; the phase weight rotates first, then the selector blends them.

```python
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
```

This primitive handles single-angle selections but struggles with compound ones, so I built a second primitive for those.

3.4 Trigonometric Products Primitive

The trigonometric product primitive computes four fundamental products, then selects from them with two weights: one chooses cosine versus sine, the other difference versus sum, and their products induce four mixing coefficients. The 2×2 selection factorizes into two parameters because the coefficient matrix is rank one. Four selections: $\cos(\theta_1-\theta_2)$ [1, 1], $\cos(\theta_1+\theta_2)$ [1, 0], $\sin(\theta_1-\theta_2)$ [0, 1], $\sin(\theta_1+\theta_2)$ [0, 0].



```python
def trigonometric_product_primitive(x, weights):
    W = torch.tanh(weights[0]) * torch.sigmoid(weights[1])
    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)

    # Two selection weights: W[0] picks cos vs sin, W[1] picks diff vs sum
    return W[0] * (W[1] * cos_diff + (1 - W[1]) * cos_sum) + (
        1 - W[0]) * (W[1] * sin_diff + (1 - W[1]) * sin_sum)
```

---

4. Experiments

I was finding all sorts of interesting things that held some promise in my mind, but my code was a mess of entangled scripts with abandoned features and flags, so I needed to start checking my work. What started as one experiment became five, each pinning down a piece: that optimal weights can be written down without training (4.1), that training finds them fast (4.2), a fair comparison with iNALU (4.3), explaining the residual error (4.4), and figuring out how robust Hill Space is to initialization scales (4.5).

4.1 Direct Weight Enumeration

Sometimes the experiment to confirm something gets to double as a demo, and I think that’s the case here. To test enumeration in isolation, I made a standalone calculator with fixed weights set to the saturation values needed for arithmetic.

```python
import numpy as np


class NeuralCalculator:
    def __init__(self):
        self.weights = {
            "add": np.array([[100.0, 100.0], [100.0, 100.0]]),
            "sub": np.array([[100.0, -100.0], [100.0, 100.0]]),
            "mul": np.array([[100.0, 100.0], [100.0, 100.0]]),
            "div": np.array([[100.0, -100.0], [100.0, 100.0]]),
        }

    def compute(self, x, y, operation):
        W_hat, M_hat = self.weights[operation]
        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
```

Full listing — A.1: hillspace/experiments/experiment_neural_calc.py (p. 25)

4.2 Learning Division Quickly

At this point, the enumeration insight simultaneously opened my mind and left me feeling empty. The realization that these units don't actually learn to internally compute math was starting to set in. But I couldn't be distracted by that; I had to continue building experiments to document my findings; it was the only thing I could do. So I chose the single most difficult operation according to the literature, and made a trainer script that finds division reliably in about a minute on a modern CPU.

```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():
    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
    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()
```

Full listing — A.2: hillspace/experiments/experiment_train_division.py (p. 26)

The script prints the learned weights and the error on three inputs far outside the training range.

4.3 Comparison with iNALU

With my confidence growing, I needed something that wasn’t mine to measure against. The original NALU paper normalized its scores in a way that is hard to interpret and reproduce, so it wasn’t a great fit. iNALU [11], on the other hand, published plain MSE with a clear protocol, which made for a fair and direct comparison.

4.3.1 Experimental Setup

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

Distributions Tested: Four distributions with extrapolation scenarios:

- **U(-5,5)**: trains on uniform [-5, 5], tests on uniform [-10, -5]
- **N(-3,3)**: trains on a normal with mean 0 and std 1 truncated to [-3, 3], tests on one with mean 9 and std 1/3 truncated to [8, 10]. iNALU's notation names the truncation interval; the normal underneath has mean (a+b)/2 and std (b−a)/6
- **E(0.8,0.5)**: trains on an exponential with scale 0.8, tests on scale 0.5, one scale shared by both inputs. There is no range shift here, only a change of scale
- **U(1.1, 1.2)**: trains on uniform [1.1, 1.2] [12], tests on uniform [1.2, 6]

Universal training distribution: I also train one model on the Goldilocks distribution U(1e-8, 10.0) (Section 2.4) and test it on every iNALU distribution without retraining.

Training Configuration:

- Optimizer: RMSProp with lr=0.01 ($\alpha = 0.9$, ε = 1e-10), as in iNALU's released code. Their paper text says Adam at lr=0.001; I follow the code
- Batch size: 64, Epochs: 100
- Loss: MSE, Auto stop threshold: MSE $\leq 10^{-14}$
- Snapping applied at evaluation (Section 2.2)
- No regularization, clipping, or reinitialization: the training distribution carries the stability instead

Evaluation Strategy:

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

4.3.2 Results

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	7e-33 ± 9e-35	7e-33 ± 9e-35
E(0.8,0.5)	a - b	1e-15 ± 2e-17	3e-33 ± 6e-35	3e-33 ± 6e-35
E(0.8,0.5)	a × b	1e-15 ± 6e-17	2e-33 ± 1e-34	2e-33 ± 1e-34
E(0.8,0.5)	a ÷ b	**362.4 ± 1e+03**	1e-28 ± 1e-29	1e-28 ± 1e-29
U(-5,5)	a + b	4e-13 ± 3e-15	1e-30 ± 1e-32	1e-30 ± 1e-32
U(-5,5)	a - b	9e-14 ± 5e-16	2e-31 ± 1e-33	2e-31 ± 1e-33
U(-5,5)	a × b	1e-10 ± 9e-13	**3e+03 ± 6.4**	1e-28 ± 9e-31
U(-5,5)	a ÷ b	**0.23 ± 0.34**	**0.18 ± 0.07**	4e-32 ± 3e-34
N(-3,3)	a + b	9e-13 ± 5e-15	3e-30 ± 2e-32	3e-30 ± 2e-32
N(-3,3)	a - b	2e-13 ± 8e-16	2e-31 ± 8e-34	2e-31 ± 8e-34
N(-3,3)	a × b	3e-10 ± 2e-12	3e-28 ± 2e-30	3e-28 ± 2e-30
N(-3,3)	a ÷ b	**2.7 ± 4.1**	**0.07 ± 0.23**	5e-32 ± 3e-34
U(1.1,1.2)	a + b		3e-31 ± 3e-33	3e-31 ± 3e-33
U(1.1,1.2)	a - b		7e-32 ± 3e-34	7e-32 ± 3e-34
U(1.1,1.2)	a × b		4e-30 ± 4e-32	4e-30 ± 4e-32
U(1.1,1.2)	a ÷ b		4e-32 ± 3e-34	4e-32 ± 3e-34

Note: Results averaged over 10 runs. Values in bold indicate degraded performance (MSE > 1e-2). iNALU did not evaluate U(1.1, 1.2); the range comes from Madsen and Johansen (2020).

The universal approach converged across all 10 runs within 100 epochs. Matched training reaches the floating-point floor on a majority of the configurations; the remaining failures occur when training on the two ranges that cross zero, one of the hazards the Goldilocks distribution exists to avoid (Section 2.4). The U(1.1, 1.2) range that Madsen and Johansen (2020) report no model could learn, is unremarkable here.

4.4 Error Analysis and Attribution

The iNALU comparison landed nicely, but multiplication and division still carried a residual I could not name. Was it floating point, or something I still did not understand? A hundred million samples per path and a few hours of compute settled it. First I establish the floor for IEEE operations, then I compare how Hill Space with analytically saturated weights

performs. For the exponential primitive I additionally compare iNALU style log-space stabilization vs complex number stabilization.

- Inputs are converted to 50-digit Decimal ground truth values
- Additional MSE = current MSE − native MSE on the same sample
- Input range U(−1e4, 1e4)
- Real: the primitive with enumerated weights in the row's dtype, matmul for add and subtract, pow for multiply and divide
- Complex128: the primitive as written in Section 3.2. It computes in complex128 whatever the row's dtype and rounds at the end, so its Float32 rows measure double-precision arithmetic rounded down, not float32 arithmetic
- Log-space follows iNALU's published stabilization

Table 4.4.1: Floating-Point Precision Baseline *100M samples per operation/dtype, native IEEE operations vs 50-digit Decimal 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	7.5e-26	3.3e-24	3.3e-24
subtract	Float32	5.25e-08	9.54e-07	9.54e-07
subtract	Float64	1.2e-25	3.3e-24	3.3e-24
multiply	Float32	7.38e-01	1.60e+01	1.59e+01
multiply	Float64	2.6e-18	5.6e-17	5.5e-17
divide	Float32	1.18e-08	6.96e-01	1.17e-08
divide	Float64	1.9e-25	1.1e-17	4.0e-26

Table 4.4.2: Additional Error Beyond the Native Floor *100M samples per operation/dtype, each compared to the native baseline*

Operation	Precision	Method	Additional MSE	Max Error	99.99%ile Error
add	Float32	Real	0.0	9.54e-07	9.54e-07
add	Float64	Real	0.0	3.3e-24	3.3e-24
subtract	Float32	Real	0.0	9.54e-07	9.54e-07
subtract	Float64	Real	0.0	3.3e-24	3.3e-24
multiply	Float32	Real	0.0	1.60e+01	1.59e+01
multiply	Float32	Complex128	0.0	1.60e+01	1.59e+01
multiply	Float32	Log-space	6.63e+02	3.59e+04	2.47e+04
multiply	Float64	Real	0.0	5.6e-17	5.5e-17
multiply	Float64	Complex128	5.8e-16	3.54e-14	2.21e-14
multiply	Float64	Log-space	2.30e-15	1.28e-13	8.58e-14
divide	Float32	Real	2.66e-08	2.49e+00	2.10e-08
divide	Float32	Complex128	0.0	6.96e-01	1.17e-08
divide	Float32	Log-space	1.52e+08	1.52e+16	2.06e-06
divide	Float64	Real	6.9e-26	1.7e-17	7.5e-26
divide	Float64	Complex128	2.4e-23	1.10e-15	4.7e-24
divide	Float64	Log-space	1.1e-22	4.21e-15	7.4e-24

Addition and Subtraction produce identical results to native IEEE operations, incurring 0.0 additional error on top of the floating point floor. For exponential primitives Complex128 performs better than Log-space stabilization.

The residual error was floating point all along.

4.5 Weight Initialization Analysis

“Things shouldn’t be going this well,” I thought, sweating figuratively and literally. Had I stumbled into a brittle magic? Perhaps I found one lucky initialization scheme that was holding everything up? The only way to know was to ambiently heat my office for a few more hours.

Table 4.5 compares initialization scales (standard deviation of the raw parameters’ starting values) across every primitive. Models were trained for up to 100 epochs (early-stopped at convergence) on the Goldilocks distribution U(1e-8, 10.0) with learning rate 0.1, Adam (β_2 = 0.5), and batch size 64, with no snapping, then evaluated on the range U(-1e4, 1e4).

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

Operation	0	0.01	0.1	1.0	3.0
a + b	2e-25 ± 4e-27	2e-25 ± 4e-27	2e-25 ± 4e-27	**7e+06** ± 1e+07	**2e+07** ± 2e+07
a - b	3e-25 ± 6e-27	3e-25 ± 6e-27	3e-25 ± 6e-27	**1e+07** ± 2e+07	**3e+07** ± 3e+07
a × b	6e-16 ± 6e-18	6e-16 ± 6e-18	6e-16 ± 6e-18	6e-16 ± 6e-18	**3e+14** ± 5e+14
a ÷ b	2e-28 ± 2e-29	2e-28 ± 2e-29	2e-28 ± 2e-29	**9e+05** ± 3e+06	**447.0** ± 297.2
a	9e-24 ± 4e-26	9e-24 ± 4e-26	9e-24 ± 4e-26	9e-24 ± 4e-26	**7e+06** ± 1e+07
1/a	1e-34 ± 2e-34	1e-34 ± 2e-34	1e-34 ± 2e-34	**0.30** ± 0.46	**0.50** ± 0.50
cos(θ)	7e-36 ± 9e-37	7e-36 ± 9e-37	7e-36 ± 9e-37	7e-36 ± 9e-37	**0.05** ± 0.10
sin(θ)	8e-36 ± 1e-36	8e-36 ± 1e-36	8e-36 ± 1e-36	4e-03 ± 0.01	**0.11** ± 0.30
cos(θ₁+θ₂)	4e-26 ± 7e-28	4e-26 ± 7e-28	4e-26 ± 7e-28	**0.15** ± 0.30	**0.30** ± 0.37
sin(θ₁+θ₂)	4e-26 ± 5e-28	4e-26 ± 5e-28	4e-26 ± 5e-28	4e-26 ± 5e-28	4e-26 ± 5e-28
cos(θ₁-θ₂)	6e-26 ± 2e-27	6e-26 ± 2e-27	6e-26 ± 2e-27	**0.15** ± 0.30	**0.33** ± 0.41
sin(θ₁-θ₂)	6e-26 ± 9e-28	6e-26 ± 9e-28	6e-26 ± 9e-28	**0.25** ± 0.39	**0.35** ± 0.43

Note: extrapolation MSE averaged over 10 seeds; bold indicates degraded performance (MSE > 1e-2). Because MSE squares a per-element error already at the float64 floor (~1e-16), the smallest values sit well below machine epsilon. The first three columns agree to the digit because every seed ends at the same exact weights, leaving only the evaluation set's rounding.

Near-neutral initializations all reach the floor; failures begin around scale 1.0. The enormous variance in the failed cells is the plateau lottery: large initializations start the raw parameters inside randomly chosen plateaus. The randomly chosen plateau protects a correct selection (Section 2.1) and works against an incorrect one that needs to escape.

4.6 Reproducibility

I'm not a trained researcher; I just love writing code. Here's the code. It will probably run on your potato PC.

- https://github.com/justindujardin/hillspace
- https://hillspace.justindujardin.com

The repository includes the primitives, figures, training and experiment scripts. The website is this document with interactive figures.

5. Related Work

I found most of these papers later in the process than I should have. What I found here disagrees with more of the published record than I expected.

NALU [1] introduced the constraint I call Hill Space and showed it extrapolating far beyond the training range when training went well. They also observed that the stable points $\{-1, 0, 1\}$ correspond to the saturation points of σ and tanh. The paper never returned to that observation, and it ended up being the key to my understanding.

Madsen and Johansen [12] performed a deep analysis of the construction, deriving its gradients, showing the expected gradient is zero at any zero-mean initialization, and measuring that converged NAC weights stall far from the targets. They concluded that the construction does "not create the desired bias" for $\{-1, 0, 1\}$ and that "learning division is impractical" (zero successes across their 100-seed benchmark). They abandoned the constraint and dropped division by design. I attribute the stalling weights to gradient vanishing (Section 2.1), find that zero-mean initialization trains reliably (Section 4.5), and that division is reliable with a constrained training distribution (Section 4.3).

iNALU [11] kept the constraint and added machinery around it: separate weight matrices per operation, mixed-sign multiplication, regularization pushing parameters toward ± 20, and reinitialization on stalls. Their regularizer drives parameters past 20, which parks a weight just short of its target (Section 2.2). Where they keep a separate weight matrix per operation, I share one weight pair between operations that converge to the same targets, such as addition and multiplication at $[1, 1]$ (Section 2.5). Both work; I have no evidence that either is better.

Neural Power Units [14] brought the complex logarithm to neural arithmetic to handle negative bases, building it into the architecture as complex weights with a closed-form real output. I use real weights, and use the complex logarithm to stabilize the exponential primitive (Section 3.2).

The Primer [13] surveys the module landscape and documents the field's frustrations: "a majority of NALMs are not robust to different training ranges," and "to date no module has been able to reliably solve division." Every treatment of the $\tanh \cdot \sigma$ surface I found looks at saturation as an obstacle. It contains no mechanism for why saturation confers stability, which I explain in Section 2.

6. Conclusion

I set out to understand why NALU sometimes struggled with and sometimes excelled at arithmetic, and the answer turned out to be one fact with a bunch of consequences: the targets sit exactly where the constraint's own gradient vanishes. This accounts for most of the notable properties of Hill Space: why it's stable and why it stalls, why some optimizers fail and others work reliably, and why a well-chosen training distribution makes division tractable. Once a unit selects its operation exactly, the only limit left is what the number format can represent.

Hill Space $W = \tanh(\hat{W}) \odot \sigma(\hat{M})$ offers a tiny piece of solid ground for future research to stand on when exploring discrete selection in neural networks.

7. Contributions

Claude (Anthropic) worked on this with me across most of the project, and helped in many ways including:

- Writing most of the experiment scripts for testing my hypotheses.
- Tutoring me on various math concepts, and re-explaining them until they landed.
- Finding and gathering the papers in Section 5, and building a reading guide for each.
- Editing, drafting, and correcting prose.

Anthropic was not involved in this work and has not reviewed it. Claude's name appears here as a record of contribution, not as an endorsement.

8. Acknowledgments

I thank Andrew Trask, Felix Hill, Scott Reed, Jack Rae, Chris Dyer, and Phil Blunsom for the constraint this report is about. A year spent inside it left me with more respect for it than I started with, and their footnote about the stable points being saturation points turned out to hold the mechanism. The paper doesn't say who contributed which piece, so the thanks goes to all of them.

My thanks also to Daniel Schlör, Markus Ring, and Andreas Hotho, whose iNALU protocol is documented clearly enough to reproduce, which is what made the comparison in Section 4.3 possible; and to Andreas Madsen and Alexander Rosenberg Johansen, whose careful negative results sharpened every claim in Section 2.

References

1. Andrew Trask, Felix Hill, Scott E. Reed, Jack Rae, Chris Dyer, Phil Blunsom. Neural Arithmetic Logic Units. *Advances in Neural Information Processing Systems 31*, 2018. https://arxiv.org/abs/1808.00508
2. Diederik P. Kingma, Jimmy Ba. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*, 2015. https://arxiv.org/abs/1412.6980
3. Ilya Loshchilov, Frank Hutter. Decoupled Weight Decay Regularization. *International Conference on Learning Representations*, 2019. https://arxiv.org/abs/1711.05101
4. Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, Jiawei Han. On the Variance of the Adaptive Learning Rate and Beyond. *International Conference on Learning Representations*, 2020. https://arxiv.org/abs/1908.03265
5. Timothy Dozat. Incorporating Nesterov Momentum into Adam. *ICLR Workshop Track*, 2016. https://openreview.net/forum?id=OM0jvwB8jIp57ZJjtNEZ
6. Tijmen Tieleman, Geoffrey Hinton. Lecture 6.5 — RMSProp: Divide the Gradient by a Running Average of Its Recent Magnitude. *COURSERA: Neural Networks for Machine Learning*, 2012. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
7. John Duchi, Elad Hazan, Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research 12*, 2011. https://jmlr.org/papers/v12/duchi11a.html
8. Matthew D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *arXiv preprint*, 2012. https://arxiv.org/abs/1212.5701
9. Martin Riedmiller, Heinrich Braun. A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm. *IEEE International Conference on Neural Networks*, 1993. https://ieeexplore.ieee.org/document/298623
10. Ilya Sutskever, James Martens, George E. Dahl, Geoffrey E. Hinton. On the Importance of Initialization and Momentum in Deep Learning. *Proceedings of the 30th International Conference on Machine Learning, PMLR 28*, 2013. https://proceedings.mlr.press/v28/sutskever13.html
11. Daniel Schlör, Markus Ring, Andreas Hotho. iNALU: Improved Neural Arithmetic Logic Unit. *Frontiers in Artificial Intelligence 3:71*, 2020. https://arxiv.org/abs/2003.07629
12. Andreas Madsen, Alexander Rosenberg Johansen. Neural Arithmetic Units. *International Conference on Learning Representations*, 2020. https://arxiv.org/abs/2001.05016
13. Bhumika Mistry, Katayoun Farrahi, Jonathon Hare. A Primer for Neural Arithmetic Logic Modules. *Journal of Machine Learning Research 23(185)*, 2022. https://arxiv.org/abs/2101.09530
14. Niklas Heim, Tomáš Pevný, Václav Šmídl. Neural Power Units. *Advances in Neural Information Processing Systems 33*, 2020. https://arxiv.org/abs/2006.01681

Appendix: code listings

A.1 hillspace/experiments/experiment_neural_calc.py

```
import sys



import numpy as np

class NeuralCalculator:
    def __init__(self):
        self.weights = {
            "add": np.array([[100.0, 100.0], [100.0, 100.0]]),
            "sub": np.array([[100.0, -100.0], [100.0, 100.0]]),
            "mul": np.array([[100.0, 100.0], [100.0, 100.0]]),
            "div": np.array([[100.0, -100.0], [100.0, 100.0]]),
        }

    def compute(self, x, y, operation):
        W_hat, M_hat = self.weights[operation]
        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]]
    print(f"Neural: {x} {op_symbol} {y} = {predicted}")
    print(f"Truth:  {actual}")
    print(f"Error:  {abs(actual - predicted):.2e}")


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

A.2 hillspace/experiments/experiment_train_division.py

```
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():
    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
    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 = "[OK]" if mse < 1e-4 else "[FAIL]"
        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()
```