ETH zürich

Why Can't RNNs Learn Math? Automata-Inspired RNNs for Exact Computation

Simon Bührer [1] **Valentin Abadie** [2 3] **Prof. Dr. Helmut Bölcskei** [2 4]

Abstract

Recurrent Neural Networks (RNNs) exhibit a paradoxical inability to learn exact arithmetic operations despite their theoretical Turing completeness. This work bridges the gap between discrete computation and continuous optimization by introducing a systematic method for translating p-stack machines—Turing-complete automata with parallel stacks—into trainable RNN architectures. Our approach leverages *stack splitting* to decompose deep stacks into shallow substacks, transforming exponential state complexity into linear scaling while preserving computational equivalence. We construct a 5-layer Clipped Rectified Linear Unit (CReLU) neural network that exactly emulates p-stack machine semantics, enabling precise arithmetic reasoning. Through analysis of the optimization landscape, we identify geometric barriers preventing convergence to exact solutions under standard training and propose remedies, including proximity-aware initialization and layer-wise training. Experiments across 13 arithmetic and bitwise operations demonstrate that our architecture achieves exact computation, with hybrid initialization strategies improving accuracy by $\sim$20% over naive methods. This framework not only enables RNNs to learn mathematical operations reliably but also illuminates fundamental challenges in neural algorithmic reasoning.[1]

[1]Department of Information Technology and Electrical Engineering, ETH Zürich, Switzerland [2]Chair for Mathematical Information Science, ETH Zürich, Switzerland [3]Supervisor who provided invaluable guidance, feedback, and support throughout this research [4]Professor who supported this work through academic guidance and research infrastructure. Correspondence to: Simon Bührer <sbuehrer@ethz.ch>, Valentin Abadie <vabadie@mins.ee.ethz.ch>.

Semester Project, Department of Information Technology and Electrical Engineering , ETH Zurich, 2025.
This work was carried out as part of the mandatory semester project for Master's students, under the supervision of a professor of the Department.

[1]Code available at: gitlab.ethz.ch/sbuehrer/turing-machine-nn-comparison

1. Introduction

Despite their remarkable success in pattern recognition and decision-making tasks, recurrent neural network (RNN) exhibit a persistent disability in learning simple arithmetic operations such as addition or multiplication, often resorting to statistical approximations that do not generalize beyond training distributions. This limitation is paradoxical given that RNN are theoretically Turing complete, capable of simulating any computational process (Siegelmann & Sontag, 1995), yet the optimal solution remains inaccessible through conventional gradient-based training methods.

We address this gap by introducing a systematic method for translating p-stack machines into learnable RNN architectures. This framework enables us to investigate why gradient-based training does not converge to exact solutions through the examination of minimal relaxations from computational optimality. By analyzing the optimization landscape near ideal solutions, we identify the architectural constraints and algorithmic conditions necessary to achieve precise arithmetic reasoning in RNN.

2. Related Work

2.1. Theoretical Computational Power of Neural Networks

Siegelmann & Sontag (1995) established the theoretical foundation for neural computation by demonstrating that RNN with rational weights and saturated-linear activation functions are Turing complete. Their construction encoded Turing machines as two-stack automata, implementing stack operations through affine transformations and thresholding using a piecewise-linear activation function. This foundational work serves as the basis for this paper's methodology of constructing RNNs from p-stack machine.

2.2. Neural Approaches to Mathematical Problem Solving

Wang et al. (2017) proposed a neural approach to math word problems using sequence-to-sequence models with Gated Recurrent Units netowrk (GRU) encoders and Long Short-Term Memory network (LSTM) decoders. Their system

translated problem text directly into mathematical equations, achieving 64.7% accuracy on their Math23K dataset of 23,161 problems through a hybrid approach combining neural methods with retrieval-based techniques.

However, their approach remained fundamentally limited to statistical pattern matching rather than genuine mathematical understanding. The model was restricted to single-variable linear equations and relied entirely on learning correlations between problem descriptions and equation templates, lacking any principled connection to the underlying mathematical operations being performed. Despite these limitations, this work provides a learnable RNN architecture whose abstracted form serves as the foundation for the model we used.

3. Methodology

Our approach proceeds through several key stages: first modeling the problem as a p-stack machine, then decomposing stacks via parallel splitting before encoding discrete operations in continuous space. We subsequently construct equivalent clipped rectified linear unit (CReLU) neural networks, finally transforming them into learnable architectures by mapping embeddings to output probabilities. This methodology guarantees that the resulting neural network preserves the exact computational semantics of the original mathematical problem while remaining fully compatible with gradient-based optimization.

3.1. p-Stack Machines

As a foundational step, we require a computational model capable of representing our mathematical problem as an automaton. A *p-stack machine* is a computational model extending classical pushdown automaton (PDA) through parallel operation on p distinct stacks, as illustrated in Figure 1(a).

Definition 3.1 (p-Stack Machine)**.** A p-stack machine is a quintuple

$$\mathcal{P} = (Q, p, \Sigma, \delta, q_0),$$

where:

- Q: Finite set of control states
- $p \geq 2 \in \mathbb{N}$: Number of stacks
- $\Sigma = \{0, 1\}$: Binary stack alphabet
- $\delta : Q \times (\Sigma \cup \{\varnothing\})^p \to Q \times \{\mathsf{push}_0, \mathsf{push}_1, \mathsf{pop}, \mathsf{id}\}^p$ where:
  - push_a: Push symbol $a \in \{0, 1\}$ onto stack
  - pop: Remove top symbol (requires $S^i \neq \varnothing$)
  - id: Identity operation
- $q_0 \in Q$: Initial state

Theorem 3.2 (Turing Completeness)**.** *No: For $p \geq 2$, the set of all p-stack machines is Turing complete. Any p-stack machine with $p \geq 2$ is Turing complete.*

Proof. (Sketch) We simulate an arbitrary Turing machine $\mathcal{M}$ using two stacks S^L and S^R:

1. **Tape representation**:
   - S^L: Symbols to the left of the tape head (reversed order)
   - S^R: Symbols at and to the right of the tape head
2. **Head movements**:
   - Right movement: $\mathsf{pop}(S^R), \mathsf{push}(S^L)$
   - Left movement: $\mathsf{pop}(S^L), \mathsf{push}(S^R)$
   - Stationary: Apply id to both stacks

Additional stacks beyond the first two use id operations. This construction preserves the unbounded storage capacity and bidirectional traversal capabilities of the original Turing machine. □

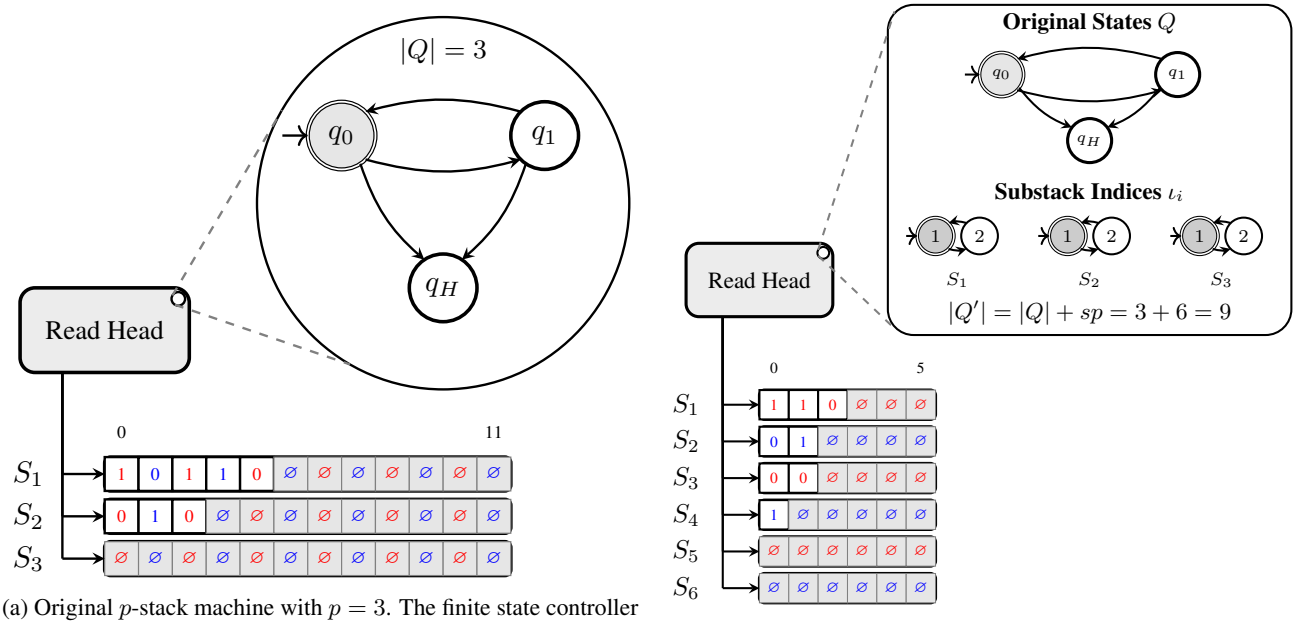
The translation of mathematical problems into p-stack machines can be achieved through direct compilation from algorithmic descriptions. As demonstrated in Appendix C, we implement this by compiling Python functions implementing basic arithmetic and bitwise logical operations into equivalent p-stack machine instructions.

3.1.1. Stack Splitting Mechanism

When transitioning from theoretical to practical implementation, stack lengths must be reduced due to precision constraints in neural networks. Since stacks are encoded as rational numbers in the neural network (Definition 3.13), their capacity is limited by available precision bits—typically 24 bits in float32 arithmetic, preventing exact computation of extremely large numbers (up to 2^{1024} or beyond). Stack splitting addresses this fundamental limitation by transforming individual stacks into multiple coordinated substacks, reducing individual stack depths while maintaining computational power through increased parallelization.

This capability becomes crucial when predicting probabilities over all possible stack combinations, which would otherwise scale exponentially with respect to the number of bits used in each stack. Through splitting, this exponential complexity is transformed into linear scaling, making the approach computationally tractable for practical applications.

Let $\mathcal{P} = (Q, p, \Sigma, \delta, q_0)$ be a p-stack machine. We define a split operation with factor $s \in \mathbb{Z}^+$ that generates a derived



(a) Original p-stack machine with $p = 3$. The finite state controller (currently in state q_0) simultaneously reads the top elements of all stacks. White cells contain binary values, gray cells marked ∅ indicate empty positions. Stack operations are applied synchronously across all stacks according to the transition function δ.

(b) Transformed machine after splitting with factor $s = 2$. Each original stack is decomposed into two substacks with cyclic element distribution. The augmented state space tracks both the main control state and active substack indices for coordinated operations.

Figure 1. Transformation from standard p-stack machine to split configuration, demonstrating the decomposition of computational resources while preserving functional equivalence.

$(s \cdot p)$-stack machine $\mathcal{P}' = (Q', sp, \Sigma, \delta', q_0')$. This transformation systematically distributes the elements of each original stack across multiple substacks.

Definition 3.3 (Cyclic Stack Splitting)**.** For stack $S^n = S_1^n \cdots S_{l_n}^n$ $(1 \le n \le p)$, create s substacks:

$$\{S^{n\cdot s+m}\}_{m=1}^{s} \text{ where}$$
$$S^{n\cdot s+m} := \{S_k^n \mid k \equiv m \pmod{s},\ 1 \le k \le l_n\}$$

with cyclic mapping:

- $S_k^n \mapsto S^{n\cdot s+(k-1 \bmod s)}_{\lceil \frac{k}{s} \rceil}$

The cyclic splitting operation induces a modular distribution of stack elements across the derived substacks. This distribution follows a deterministic pattern wherein the k-th element of the original stack is assigned to the substack indexed by the congruence class of k modulo s.

Remark 3.4. The splitting operation preserves content through isomorphism:

$$S^n \cong \bigsqcup_{m=1}^{s} S^{n\cdot s+m}$$

where $\sqcup$ denotes ordered concatenation. Substack sizes differ by at most 1 element.

Definition 3.5 (Augmented State Space)**.** The new state space is defined as:

$$Q' = (Q \setminus \{q_H\}) \times [s]^p \cup \{q_H\}, \quad [s] := \{1, \ldots, s\}$$

States $(q, \vec{\iota}) \in Q'$ contain:

- $q \in Q \setminus \{q_H\}$: Original control state
- $\vec{\iota} = (\iota_1, \ldots, \iota_p) \in [s]^p$: Substack index pointers

The implementation of stack splitting necessitates an augmentation of the state space to facilitate the coordination of operations across the derived substacks. Each non-halting state in the original machine expands to incorporate a vector of indices that identify the currently active substack for each original stack. This augmentation enables precise tracking of the computational state across the distributed stack structure.

Example 3.6. For $p = 3, s = 5$, state $q_a \neq q_H$ generates:

$$\{(q_a, i, j, k) \mid i, j, k \in \{1, \ldots, 5\}\}$$

yielding $(|Q| - 1) \cdot 5^3 + 1$ states. The halting state q_H remains atomic.

Definition 3.7 (Transformed Transitions)**.**

$$\delta' : Q' \times (\Sigma \cup \{\varnothing\})^{sp} \to Q' \times \{\mathsf{push}_0, \mathsf{push}_1, \mathsf{pop}, \mathsf{id}\}^{sp}$$

For original transition $\delta(q, S_1^1, \ldots, S_1^p) = (q', \mathsf{op}_1, \ldots, \mathsf{op}_p)$:

- For stack i with index ι_i:

$$\begin{aligned}\mathsf{op}'_{(i-1)s+\iota'_i} &= \mathsf{push}_a, \quad \text{if } \mathsf{op}_i = \mathsf{push}_a \\ \mathsf{op}'_{(i-1)s+\iota_i} &= \mathsf{pop}, \quad \text{if } \mathsf{op}_i = \mathsf{pop} \\ \mathsf{op}'_{(i-1)s+\iota_i} &= \mathsf{id}, \quad \text{otherwise}\end{aligned}$$

- Index update:

$$\iota'_i = \begin{cases}(\iota_i + 1) \bmod s & \text{if } \mathsf{op}_i = \mathsf{push} \\ (\iota_i - 1) \bmod s & \text{if } \mathsf{op}_i = \mathsf{pop} \\ \iota_i & \text{if } \mathsf{op}_i = \mathsf{id}\end{cases}$$

The transformed transition function maps operations on the original stacks to corresponding operations on the appropriate substacks. Each operation is directed to a specific substack as determined by the current index vector. Push operations target the next substack in the rotation, while pop operations extract elements from the current substack. The index vectors are updated in accordance with the executed operations, maintaining the cyclic access pattern that ensures the proper distribution of elements across substacks.

Proposition 3.8 (Operational Equivalence)**.** *There exists a bijection ϕ that preserves transitions:*

$$(q, \{S^n\}) \leftrightarrow ((q, \vec{\iota}), \{S^{n \cdot s + m}\})$$

satisfying:

$$\begin{aligned}\delta(q, S_1^1, \ldots, S_1^p) = (q', \vec{\mathsf{op}}) \iff \\ \delta'(\phi(q), \phi(S_1^1, \ldots, S_1^p)) = (\phi(q'), \phi(\vec{\mathsf{op}}))\end{aligned}$$

3.1.2. Multistate Stack Splitting

The exponential growth in state complexity associated with naive stack splitting necessitates the development of more efficient state representation techniques. Multistate splitting addresses this limitation by introducing a factored state representation that scales linearly with the number of stacks and the splitting factor, as demonstrated in Figure 1(b).

Definition 3.9 (s-split p-stack machine)**.** An s-split p-stack machine is a tuple $\mathcal{M} = (Q, p, s, \Sigma, \delta, q_0, \iota_0)$ where:

- Q is a finite set of control states
- $p \in \mathbb{N}$ is the number of logical stacks
- $s \in \mathbb{N}$ is the split factor per stack
- Σ is the stack alphabet
- $\delta : Q \times [s]^p \times (\Sigma \cup \{\varnothing\})^{ps} \to Q \times [s]^p \times \{\mathsf{push}_0, \mathsf{push}_1, \mathsf{pop}, \mathsf{id}\}^{ps}$ is the transition function
- $q_0 \in Q$ is the initial control state
- $\iota_0 = (0, 0, \ldots, 0) \in [s]^p$ is the initial index vector

In this model, the machine's configuration is represented by $(q, \vec{\iota}, \vec{S})$ where:

- $q \in Q$ is the current control state
- $\vec{\iota} = (\iota_1, \ldots, \iota_p) \in [s]^p$ tracks the active substack index for each logical stack
- $\vec{S} = (S_1, S_2, \ldots, S_{ps})$ represents the ps physical substacks, where each S_j is a string over Σ

Definition 3.10 (Transition Semantics)**.** For a transition $\delta(q, \vec{\iota}, \vec{\sigma}) = (q', \vec{\iota}', \vec{op})$ where $\vec{\sigma} = (\sigma_1, \ldots, \sigma_{ps})$ contains the top symbols of all substacks (or $\varnothing$ if empty):

1. The control state updates: $q \to q'$
2. For each logical stack $i \in [p]$, the active substack index updates according to the operation on substack $(i-1)s + \iota_i$:

$$\iota'_i = \begin{cases}(\iota_i + 1) \bmod s & \text{if } op_{(i-1)s+\iota_i} = \mathsf{push}_a \\ (\iota_i - 1) \bmod s & \text{if } op_{(i-1)s+\iota_i} = \mathsf{pop} \\ \iota_i & \text{if } op_{(i-1)s+\iota_i} = \mathsf{id}\end{cases}$$

3. Operations $\vec{op} = (op_1, \ldots, op_{ps})$ are applied to the corresponding physical substacks.

Definition 3.11 (Construction from p-stack Machine)**.** Given a p-stack machine $\mathcal{P} = (Q, p, \Sigma, \delta_P, q_0)$, we construct the equivalent s-split p-stack machine $\mathcal{M} = (Q, p, s, \Sigma, \delta_M, q_0, \iota_0)$ where:

For each transition $\delta_P(q, \sigma_1, \ldots, \sigma_p) = (q', op_1, \ldots, op_p)$ in the original machine, we define the corresponding transition in the split machine as:

$$\delta_M(q, \vec{\iota}, \vec{\sigma}) = (q', \vec{\iota}', \vec{op}')$$

where:

- σ_i is the top symbol of the active substack for logical stack i (i.e., $\sigma_{(i-1)s+\iota_i}$)
- $\vec{\iota}'$ is computed according to the transition semantics above
- $\vec{op}'$ is defined as:

$$op'_j = \begin{cases}op_i & \text{if } j = (i-1)s + \iota'_i \text{ and } op_i = \mathsf{push}_a \\ op_i & \text{if } j = (i-1)s + \iota_i \text{ and } op_i = \mathsf{pop} \\ op_i & \text{if } j = (i-1)s + \iota_i \text{ and } op_i = \mathsf{id} \\ \mathsf{id} & \text{otherwise}\end{cases}$$

where $i \in [p]$ is the logical stack index such that j corresponds to a substack of logical stack i.

The factored transition function operates analogously to naive splitting but leverages the compact state representation. Operations are directed to the appropriate substacks based on the current indices, and the indices are updated cyclically to distribute operations across substacks. This representation supports the same operational semantics as naive splitting while achieving significant space complexity reduction.

Proposition 3.12 (State Complexity Reduction)**.** *The s-split p-stack machine achieves a state representation complexity of:*

$$\mathcal{O}(|Q| + ps) \text{ bits vs } \mathcal{O}(|Q|s^p) \text{ explicit states}$$

while preserving the computational capabilities of naive splitting.

3.2. Encoded p-Stack Machines

We present an encoded reformulation of p-stack machines, as proposed by (Siegelmann & Sontag, 1995; Neeser, 2023), that enables numerical manipulation of stack operations through continuous embeddings, while preserving computational equivalence.

The fundamental innovation in this approach lies in representing inherently discrete stack structures as points in a continuous space, thereby enabling the application of analytical techniques from real analysis and numerical computation. This transformation is essential for constructing neural network implementations that can leverage gradient-based optimization methods.

Definition 3.13 (Stack Encoding)**.** For any stack $S^n = S_1^n S_2^n \cdots S_k^n \in \{0,1\}^*$ with S_i^n denoting the i-th symbol (top-first), define:

$$\mathsf{enc}(S^n) := \sum_{i=1}^{k} \frac{2S_i^n + 1}{4^i} \in [0,1)$$

with $\mathsf{enc}(\varnothing) = 0$ for the empty stack. This creates a bijection:

$$\mathsf{enc} : \{0,1\}^* \hookrightarrow C \subset [0,1)$$

where C is the Cantor set of valid encodings.

This bijective mapping ensures that no information is lost in the encoding process, allowing for perfect reconstruction of the original stack from its numerical representation.

3.2.1. Encoded Stack Operations

The encoding admits algebraic implementations of stack operations, transforming discrete manipulations of stack elements into continuous functions on real numbers.

Definition 3.14 (Encoded Operations)**.** Given an encoded stack $\tilde{S}^n = \mathsf{enc}(S^n)$ and binary value $a \in \{0,1\}$, the fundamental operations are defined as follows:

$$\begin{aligned}
\mathsf{push}_a(\tilde{S}^n) &:= \frac{1}{4}\tilde{S}^n + \frac{2a+1}{4}, \\
\mathsf{pop}(\tilde{S}^n) &:= 4\tilde{S}^n - \left(2\zeta(\tilde{S}^n) + 1\right), \\
\zeta(\tilde{S}^n) &:= \sigma(4\tilde{S}^n - 2) \quad \text{(top element operator)}, \\
\tau(\tilde{S}^n) &:= \sigma(4\tilde{S}^n) \quad \text{(nonempty operator)},
\end{aligned}$$

where $\sigma : \mathbb{R} \to [0,1]$ represents the CReLU, defined through piecewise linear saturation:

$$\sigma(x) := \begin{cases} 0, & x < 0 \\ x, & 0 \le x < 1 \\ 1, & x \ge 1 \end{cases}.$$

As demonstrated in (Siegelmann & Sontag, 1994), this piecewise linear activation function maintains universal computational properties - in fact, a broad class of smooth sigmoidal activations offers no additional computational power beyond this simple saturated form.

The algebraic nature of these operations allows for their direct implementation within neural network architectures through carefully designed linear transformations and activation functions.

3.3. CReLU Neural Network from p-Stack Machine

This section presents an aproach for converting a multistate p-stack machine into an equivalent CReLU neural network. The resulting network achieves exact computational equivalence with the original automaton.

Definition 3.15 (CReLU Neural Network)**.** A CReLU neural network is a composition of affine transformations and CReLU activation functions. Formally, a CReLU neural network $\Phi : \mathbb{R}^d \to \mathbb{R}^{d'}$ is given by

$$\Phi = \sigma \circ F^L \circ \sigma \circ F^{L-1} \circ \cdots \circ \sigma \circ F^1,$$

where each $F^l : \mathbb{R}^{N_{l-1}} \to \mathbb{R}^{N_l}$ is an affine transformation $F^l(x) = W^l x + b^l$ with weight matrix W^l and bias vector b^l. The CReLU activation function σ is applied componentwise.

For a multistate p-stack machine $\mathcal{P} = (Q, p, \Sigma, \delta, q_0)$ with split factor s, we construct a 5-layer CReLU neural network $\mathcal{N}_\mathcal{P}$ where each layer applies the transformation:

$$\mathbf{x}^{(i+1)} = \sigma(W_i \mathbf{x}^{(i)} + b_i)$$

3.3.1. Network State Representation

The network's input vector $\mathbf{x}^{(1)}$ encapsulates the complete computational state of the p-stack machine through three essential components:

$$\mathbf{x}^{(1)} = \begin{bmatrix} \underbrace{e_{\iota_q}}_{\text{Main state}} \\ \underbrace{e_{\iota_1}, \ldots, e_{\iota_p}}_{\text{substack indices}} \\ \underbrace{\mathsf{enc}(S^1), \ldots, \mathsf{enc}(S^{s \cdot p})}_{\text{Encoded stacks}} \end{bmatrix} \in \mathbb{R}^{|Q|+2s \cdot p}$$

where:

- **Main state vector** $e_{\iota_q} \in \{0,1\}^{|Q|}$: One-hot encoding of the current main control state, with $(e_{\iota_q})_j = \delta_{\iota_q, j}$ for $j = 1, \ldots, |Q|$. Only one element equals 1 (the current state), while all others are 0.
- **substack index vectors** $e_{\iota_i} \in \{0,1\}^s$ for $i \in \{1, \ldots, p\}$: One-hot encodings of the current active substack index for each original stack. These indicate which of the s substacks is currently accessed for operations on each of the p original stacks.
- **Stack encodings** $\mathsf{enc}(S^j) \in [0,1)$ for $j \in \{1, \ldots, s \cdot p\}$: Real-valued encodings of all substacks as defined in Definition 3.13. Each encoding represents the entire contents of one substack as a single real number.

3.3.2. Layer 1: Conditional Top and Empty Bit Extraction

The first layer extracts essential information from the encoded stacks, but only for the currently active substacks as determined by the substack index vectors.

$$W_1 = \begin{bmatrix} \underbrace{\mathbf{I}_{|Q|+2s \cdot p}}_{\text{Identity mapping}} \\ \underbrace{\begin{bmatrix} \mathbf{0}_{s \cdot p \times |Q|} & 2\mathbf{I}_{s \cdot p} & 4\mathbf{I}_{s \cdot p} \\ \mathbf{0}_{s \cdot p \times |Q|} & 2\mathbf{I}_{s \cdot p} & 4\mathbf{I}_{s \cdot p} \end{bmatrix}}_{\text{Top and nonempty extraction}} \end{bmatrix} \in \mathbb{R}^{(|Q|+4s \cdot p) \times (|Q|+2s \cdot p)}$$

$$b_1 = \begin{bmatrix} \underbrace{\mathbf{0}_{|Q|+2s \cdot p}}_{\text{Pass-through bias}} \\ \underbrace{-4 \cdot \mathbf{1}_{s \cdot p}}_{\text{Top bit threshold}} \\ \underbrace{-2 \cdot \mathbf{1}_{s \cdot p}}_{\text{Non-empty threshold}} \end{bmatrix} \in \mathbb{R}^{|Q|+4s \cdot p}$$

After this transformation, the output vector $\mathbf{x}^{(2)}$ has the structure:

$$\mathbf{x}^{(2)} = \begin{bmatrix} \underbrace{e_{\iota_q}}_{\text{Main state (preserved)}} \\ \underbrace{e_{\iota_1}, \ldots, e_{\iota_p}}_{\text{substack indices (preserved)}} \\ \underbrace{\mathsf{enc}(S^1), \ldots, \mathsf{enc}(S^{s \cdot p})}_{\text{Encoded stacks (preserved)}} \\ \underbrace{\zeta'(S^1), \ldots, \zeta'(S^{s \cdot p})}_{\text{Conditional top bit extraction}} \\ \underbrace{\tau'(S^1), \ldots, \tau'(S^{s \cdot p})}_{\text{Conditional non-empty indicators}} \end{bmatrix} \in \mathbb{R}^{|Q|+4s \cdot p}$$

where:

- **Conditional top bit extraction** $\zeta'(S^j)$: For an active substack (i.e., the current index points to this substack), this extracts the topmost element. If the substack is not active or is empty, the output is 0.
- **Conditional non-empty indicator** $\tau'(S^j)$: For an active substack, this indicates whether the stack contains elements (1) or is empty (0). If the substack is not active, the output is 0.

The key insight is that the high negative bias values (-4 and -2) ensure that only active substacks with sufficient contents produce non-zero outputs after CReLU activation. The operations are implicitly conditioned on the substack indices through the structure of the full vector.

Element-wise, for each stack encoding $i \in \{1, \ldots, p\}, j \in \{1, \ldots, s\}$:

$$\begin{aligned} k &= i \cdot s + j \\ \zeta'(S^k) &= \sigma(2e_{\iota_i} \cdot e_k + 4 \cdot \mathsf{enc}(S^k) - 4) \\ \tau'(S^k) &= \sigma(2e_{\iota_i} \cdot e_k + 4 \cdot \mathsf{enc}(S^k) - 2) \end{aligned}$$

These operations effectively implement the ζ and τ functions from Section 3.2, but only for active substacks.

3.3.3. Layer 2: substack Aggregation

The second layer aggregates information across substacks for each original stack, combining the conditional top bit extractions and non-empty indicators from Layer 1.

$$W_2 = \begin{bmatrix} \underbrace{\mathbf{I}_{|Q|+2s \cdot p}}_{\text{Identity for states and stacks}} & \underbrace{\mathbf{0}_{(|Q|+2s \cdot p) \times 2s \cdot p}}_{\text{Zero padding}} \\ \underbrace{\mathbf{0}_{2p \times (|Q|+2s \cdot p)}}_{\text{Zero padding}} & \underbrace{W_{\text{combine}}}_{\text{Aggregation matrix}} \end{bmatrix} \in \mathbb{R}^{(|Q|+2s \cdot p+2p) \times (|Q|+4s \cdot p)}$$

$$b_2 = \mathbf{0}_{|Q|+2s \cdot p+2p} \in \mathbb{R}^{|Q|+2s \cdot p+2p}$$

The key component $W_{\text{combine}} \in \mathbb{R}^{2p \times (2s \cdot p)}$ contains horizontal stripes of ones connecting each output position to its corresponding substacks:

$$(W_{\text{combine}})_{i,j} = \begin{cases} 1 & \text{if } j \in [is,\ (i+1)s - 1] \\ 0 & \text{otherwise} \end{cases}$$
$$\text{for } 0 \le i < 2p,\ 0 \le j < 2s \cdot p$$

After this transformation, the output vector $\mathbf{x}^{(3)}$ has the structure:

$$\mathbf{x}^{(3)} = \left[\begin{array}{c} \underbrace{e_{\iota_q}}_{\text{Main state}} \\ \underbrace{e_{\iota_1}, \ldots, e_{\iota_p}}_{\text{substack indices}} \\ \underbrace{\mathsf{enc}(S^1), \ldots, \mathsf{enc}(S^{s \cdot p})}_{\text{Encoded stacks}} \\ \underbrace{\zeta(S^1), \ldots, \zeta(S^p)}_{\text{Aggregated top bits}} \\ \underbrace{\tau(S^1), \ldots, \tau(S^p)}_{\text{Aggregated non-empty indicators}} \end{array}\right] \in \mathbb{R}^{|Q|+2s \cdot p+2p}$$

where:

- **Aggregated top bits** $\zeta(S^i) \in \{0, 1\}$: Combined top bit information for each original logical stack, equal to 1 if the active substack of that logical stack has top bit 1.
- **Aggregated non-empty indicators** $\tau(S^i) \in \{0, 1\}$: Combined emptiness information, equal to 1 if the active substack of that logical stack is non-empty.

3.3.4. Layer 3: Pattern Matching

The third layer implements precise state-stack configuration matching through parallel transition rule verification. Let $\mathcal{P} = (Q, p, \Sigma, \delta, q_0)$ be the original p-stack machine with split factor s, and let n_δ denote the number of distinct transition rules. The layer architecture is:

$$W_3 = \left[\begin{array}{c} \underbrace{W_{\text{pattern}}}_{\substack{\text{Transition rule} \\ \text{conditions}}} \\ \left[\mathbf{I}_{|Q|+2ps+p} \quad \mathbf{0}_{(|Q|+2ps+p) \times p}\right] \end{array}\right]$$
$$\in \mathbb{R}^{(n_\delta + |Q|+2ps+p) \times (|Q|+4ps+2p)}$$

$$b_3 = \left[\begin{array}{c} \underbrace{b_{\text{pattern}}}_{\substack{\text{Match} \\ \text{thresholds}}} \\ \underbrace{\mathbf{0}_{|Q|+2ps+p}}_{\text{Passthrough bias}} \end{array}\right] \in \mathbb{R}^{n_\delta + |Q|+2ps+p}$$

Pattern Matrix Construction The pattern matrix $W_{\text{pattern}} \in \mathbb{R}^{n_\delta \times (|Q|+2p)}$ encodes n_δ transition rules, where each row i corresponds to a specific transition rule $\delta_i(q, S_1^1, \ldots, S_1^p) = (q', \mathsf{op}_1, \ldots, \mathsf{op}_p)$. The construction follows a systematic three-component encoding:

1. **State Matching Component (Columns 1 to $|Q|$)**: Each transition rule targets a specific current state $q \in Q$. The encoding uses one-hot with conflict prevention:

   $$(W_{\text{pattern}})_{i,k} = \begin{cases} 1 & \text{if } k = \text{index}(q_i) \text{ (target state)} \\ -1 & \text{if } k \neq \text{index}(q_i) \text{ (other states)} \end{cases}$$

   This ensures that only the correct current state activates the pattern, while incorrect states suppress activation.

2. **Stack Top Conditions (Columns $|Q| + 1$ to $|Q| + p$)**: For each stack $n \in \{1, \ldots, p\}$, the top symbol requirement is encoded via:

   $$(W_{\text{pattern}})_{i,|Q|+n} = \phi_{\text{top}}(S_{1,i}^n)$$

   where $\phi_{\text{top}} : \Sigma \cup \{\varnothing, X\} \to \{-1, 0, 1\}$ is defined as:

   $$\phi_{\text{top}}(S_1^n) = \begin{cases} 1 & \text{if } S_1^n = 1 \text{ (top bit = 1)} \\ -1 & \text{if } S_1^n \in \{0, \varnothing\} \text{ (top bit = 0)} \\ 0 & \text{if } S_1^n = X \text{ (wildcard, no constraint)} \end{cases}$$

3. **Stack Nonempty Conditions (Columns $|Q| + p + 1$ to $|Q| + 2p$)**: The stack emptiness/non-emptiness requirements are encoded via:

   $$(W_{\text{pattern}})_{i,|Q|+p+n} = \phi_{\text{pres}}(S_{1,i}^n)$$

   where $\phi_{\text{pres}} : \Sigma \cup \{\varnothing, X\} \to \{-1, 1, 0\}$ is defined as:

   $$\phi_{\text{pres}}(S_1^n) = \begin{cases} 1 & \text{if } S_1^n \in \{0, 1\} \text{ (non-empty stack)} \\ -1 & \text{if } S_1^n = \varnothing \text{ (empty stack)} \\ 0 & \text{if } S_1^n = X \text{ (wildcard, no constraint)} \end{cases}$$

Threshold Calibration for AND Logic The bias vector b_{pattern} implements strict AND logic through carefully calibrated thresholds. For each pattern i, the threshold is set to:

$$(b_{\text{pattern}})_i = -\left(\sum_{j=1}^{|Q|+2p} \mathbb{I}_{(W_{\text{pattern}})_{i,j}=1}\right) + 1$$

where $\mathbb{I}_{(W_{\text{pattern}})_{i,j}=1}$ counts the number of positive conditions (value 1) in pattern i. This creates an activation threshold that requires *all* specified positive conditions to be simultaneously satisfied.

After this layer, the output vector $\mathbf{x}^{(4)}$ has the structure:

$$\mathbf{x}^{(4)} = \begin{bmatrix} \underbrace{\pi_1, \ldots, \pi_{n_\delta}}_{\text{Pattern activations}} \\ \underbrace{e_{\iota_q}}_{\text{Main state (passthrough)}} \\ \underbrace{e_{\iota_1}, \ldots, e_{\iota_p}}_{\text{substack indices (passthrough)}} \\ \underbrace{\mathsf{enc}(S^1), \ldots, \mathsf{enc}(S^{s \cdot p})}_{\text{Encoded stacks (passthrough)}} \\ \underbrace{\zeta(S^1), \ldots, \zeta(S^p)}_{\text{Aggregated top bits (passthrough)}} \end{bmatrix} \in \mathbb{R}^{n_\delta + |Q| + 2s \cdot p + p}$$

where:

- **Pattern activations** $\pi_i \in \{0, 1\}$ for $i = 1, \ldots, n_\delta$: Binary indicators where $\pi_i = 1$ if and only if the current machine configuration $(e_{\iota_q}, \zeta(S^1), \ldots, \zeta(S^p), \tau(S^1), \ldots, \tau(S^p))$ exactly matches the i-th transition rule pattern. At most one π_i should be active for deterministic machines.
- **Passthrough components**: All state and stack information from the input is preserved unchanged through the identity block in the lower portion of W_3, except that the aggregated non-empty indicators $\tau(S^1), \ldots, \tau(S^p)$ are not passed through as they were only needed for pattern matching.

3.3.5. Layer 4: Operation Selection and Execution

The fourth layer implements the core computational logic by mapping recognized patterns to state transitions and stack operations through a precisely structured weight matrix. This layer transforms pattern activations from Layer 3 into concrete state machine operations, effectively implementing the transition function δ of the p-stack machine.

Let n_δ denote the number of distinct pattern vectors recognized in Layer 3, and let $\mathcal{L} = \{(q', \mathsf{op}_1, \ldots, \mathsf{op}_p)\}$ be the set of unique transition labels, where each label defines the next main state q' and p stack operations.

The layer's architecture is defined by the weight matrix W_4 and bias vector b_4:

$$W_4 = \begin{bmatrix} \begin{bmatrix} W_{\text{trans}} & \mathbf{0}_{|Q| \times (|Q| + 2s \cdot p + p)} \end{bmatrix} \\ W_{\text{index}} \\ W_{\text{stack}} \end{bmatrix} \in \mathbb{R}^{(|Q| + 9s \cdot p) \times (n_\delta + |Q| + 2s \cdot p + p)}$$

$$b_4 = \begin{bmatrix} \mathbf{0}_{|Q|} \\ b_{\text{index}} \\ b_{\text{stack}} \end{bmatrix} \in \mathbb{R}^{|Q| + 9s \cdot p}$$

State Transition Matrix Construction The state transition component $W_{\text{trans}} \in \mathbb{R}^{|Q| \times n_\delta}$ maps pattern activations directly to next states. For each unique transition label $\ell_j = (q'_j, \mathsf{op}_{1,j}, \ldots, \mathsf{op}_{p,j}) \in \mathcal{L}$ with associated pattern group $\mathcal{P}_j$ containing $|\mathcal{P}_j|$ patterns, the matrix entries are defined as:

$$(W_{\text{trans}})_{q'_j, k} = \begin{cases} 1 & \text{if pattern } k \text{ belongs to label group } j \\ 0 & \text{otherwise} \end{cases}$$

where patterns are indexed sequentially across all label groups. This construction ensures that when a pattern π_k is activated in Layer 3, the corresponding next state q'_j receives unit activation, implementing the state transition component of δ.

Substack Index Selection Matrix Construction The substack index selection matrix $W_{\text{index}} \in \mathbb{R}^{(3s \cdot p) \times (n_\delta + |Q| + 2s \cdot p + p)}$ determines which substack configurations become active after each operation. This is crucial because different operations require different substack index updates as described in Definition 3.10.

The matrix is structured as:

$$W_{\text{index}} = \begin{bmatrix} W_{\text{id}} & \mathbf{0}_{sp \times |Q|} & \mathbf{I}_{sp} & \mathbf{0}_{sp \times p} \\ W_{\text{push}} & \mathbf{0}_{sp \times |Q|} & \mathbf{C}_{+1}^{(p)} & \mathbf{0}_{sp \times p} \\ W_{\text{pop}} & \mathbf{0}_{sp \times |Q|} & \mathbf{C}_{-1}^{(p)} & \mathbf{0}_{sp \times p} \end{bmatrix}$$

Definition 3.16 (Circular Shift Matrix)**.** For a given dimension n, the circular shift matrix $\text{CircShift}(k) \in \mathbb{R}^{n \times n}$ with shift parameter k is defined as:

$$(\text{CircShift}(k))_{i,j} = \begin{cases} 1 & \text{if } j = (i - k - 1) \bmod n + 1 \\ 0 & \text{otherwise} \end{cases}$$

where indices are 1-based. For $k = +1$, this shifts elements one position forward (cyclically), and for $k = -1$, it shifts elements one position backward.

Definition 3.17 (Stack Shift Matrix)**.** Let $\mathbf{C}_k^{(p)} \in \mathbb{R}^{sp \times sp}$ denote the block diagonal matrix:

$$\mathbf{C}_k^{(p)} = \text{Diag}(\text{CircShift}(k)_s, \ldots, \text{CircShift}(k)_s)$$

with p blocks of $\text{CircShift}(k)_s \in \mathbb{R}^{s \times s}$.

The operation-specific submatrices are defined as follows:

1. **Identity Operations** (W_{id}): For stack i and substack r, the entry at position $(i-1)s + r$ is:

$$(W_{\text{id}})_{(i-1)s+r, k} = \begin{cases} 1 & \text{if pattern } k \text{ has } \mathsf{op}_i = \mathsf{id} \\ 0 & \text{otherwise} \end{cases}$$

2. **Push Operations** (W_{push}): For stack i and substack r, positioned at rows $s \cdot p + (i-1)s + r$:

$$(W_{\text{push}})_{(i-1)s+r,k} = \begin{cases} 1 & \text{if pattern } k \text{ has } \mathsf{op}_i \in \mathsf{push} \\ 0 & \text{otherwise} \end{cases}$$

3. **Pop Operations** (W_{pop}): For stack i and substack r, positioned at rows $2s \cdot p + (i-1)s + r$:

$$(W_{\text{pop}})_{(i-1)s+r,k} = \begin{cases} 1 & \text{if pattern } k \text{ has } \mathsf{op}_i = \mathsf{pop} \\ 0 & \text{otherwise} \end{cases}$$

The bias vector b_{index} contains threshold values of -1 for all operation selection neurons, ensuring proper activation only when the corresponding operation is selected.

Stack Processing Matrix Construction The stack processing component $W_{\text{stack}} \in \mathbb{R}^{(6s\cdot p)\times(n_\delta+|Q|+2s\cdot p+p)}$ implements the actual stack manipulations through six distinct operational blocks. Rather than viewing this as separate column matrices, it's more meaningful to understand it as six row blocks, each implementing a specific operational pathway:

$$W_{\text{stack}} = \begin{bmatrix} W_{\text{block-1}} \\ W_{\text{block-2}} \\ W_{\text{block-3}} \\ W_{\text{block-4}} \\ W_{\text{block-5}} \\ W_{\text{block-6}} \end{bmatrix}$$

where each $W_{\text{block-i}} \in \mathbb{R}^{sp\times(n_\delta+|Q|+2s\cdot p+p)}$ handles a specific aspect of stack operations.

Each block decomposes as:

$$W_{\text{block-i}} = \begin{bmatrix} W^{(i)}_{\text{pattern-ops}} & \mathbf{0}_{sp\times|Q|} & W^{(i)}_{\text{microstate-index}} & W^{(i)}_{\text{stack-transform}} \end{bmatrix}$$

Block 1 - Identity Operations: Direct passthrough for stacks with identity operations.

$$(W^{(1)}_{\text{pattern-ops}})_{(i-1)s+j,k} = \begin{cases} 1 & \text{if pattern } k \text{ has } \mathsf{op}_i = \mathsf{id} \\ 0 & \text{otherwise} \end{cases}$$
$$W^{(1)}_{\text{microstate-index}} = \mathbf{I}_{sp}$$
$$W^{(1)}_{\text{stack-transform}} = \mathbf{I}_{sp}$$

Block 2 - Push-0 Operations: Applies push_0 by shifting to next substack position.

$$(W^{(2)}_{\text{pattern-ops}})_{(i-1)s+j,k} = \begin{cases} 1 & \text{if pattern } k \text{ has } \mathsf{op}_i = \mathsf{push}_0 \\ 0 & \text{otherwise} \end{cases}$$
$$W^{(2)}_{\text{microstate-index}} = \mathbf{C}^{(p)}_{+1}$$
$$W^{(2)}_{\text{stack-transform}} = 0.25 \cdot \mathbf{I}_{sp}$$

Block 3 - Push-1 Operations: Applies push_1 by shifting to next substack position.

$$(W^{(3)}_{\text{pattern-ops}})_{(i-1)s+j,k} = \begin{cases} 1 & \text{if pattern } k \text{ has } \mathsf{op}_i = \mathsf{push}_1 \\ 0 & \text{otherwise} \end{cases}$$
$$W^{(2)}_{\text{microstate-index}} = \mathbf{C}^{(p)}_{+1}$$
$$W^{(2)}_{\text{stack-transform}} = 0.25 \cdot \mathbf{I}_{sp}$$

Block 4 - Pop Operations: Handles pop operations with microstate index gating.

$$(W^{(4)}_{\text{pattern-ops}})_{(i-1)s+j,k} = \begin{cases} 1 & \text{if pattern } k \text{ has } \mathsf{op}_i = \mathsf{pop} \\ 0 & \text{otherwise} \end{cases}$$
$$W^{(2)}_{\text{microstate-index}} = \mathbf{C}^{(p)}_{-1}$$
$$W^{(2)}_{\text{stack-transform}} = 4 \cdot \mathbf{I}_{sp}$$

Block 5 - Push Passthrough: Ensures continuation of substacks during push operations.

$$(W^{(5)}_{\text{pattern-ops}})_{(i-1)s+j,k} = \begin{cases} 1 & \text{if pattern } k \text{ has } \mathsf{op}_i = \mathsf{push} \\ 0 & \text{otherwise} \end{cases}$$
$$W^{(2)}_{\text{microstate-index}} = -2 \cdot \mathbf{C}^{(p)}_{+1}$$
$$W^{(2)}_{\text{stack-transform}} = \mathbf{I}_{sp}$$

Block 6 - Inactive Substack Passthrough: Maintains inactive substacks not affected by operations.

$$(W^{(6)}_{\text{pattern-ops}})_{(i-1)s+j,k} = \begin{cases} -1 & \text{if pattern } k \text{ has } \mathsf{op}_i = \mathsf{push} \\ 0 & \text{otherwise} \end{cases}$$
$$W^{(2)}_{\text{microstate-index}} = -2 \cdot \mathbf{I}^{(p)}_{sp}$$
$$W^{(2)}_{\text{stack-transform}} = \mathbf{I}_{sp}$$

Bias Configuration by Block:

$$b_{\text{stack}} = \begin{bmatrix} -\mathbf{1}_{sp} \\ -\mathbf{1}_{sp} \\ -\mathbf{1}_{sp} \\ -2 \cdot \mathbf{1}_{sp} \\ -\frac{7}{4} \cdot \mathbf{1}_{sp} \\ -\frac{5}{4} \cdot \mathbf{1}_{sp} \end{bmatrix}$$

The transformation produces the output vector with 10 dis-

tinct components:

$$\mathbf{x}^{(5)} = \begin{bmatrix} \underbrace{e_{\iota_{q'}}}_{\text{1. Main state passthrough}} \\ \underbrace{\Phi^{(1)}_{\text{id}}(S^1), \ldots, \Phi^{(1)}_{\text{id}}(S^p)}_{\text{2. Conditional substack index id}} \\ \underbrace{\Phi^{(2)}_{\text{push}}(S^1), \ldots, \Phi^{(2)}_{\text{push}}(S^p)}_{\text{3. Conditional substack index push}} \\ \underbrace{\Phi^{(3)}_{\text{pop}}(S^1), \ldots, \Phi^{(3)}_{\text{pop}}(S^p)}_{\text{4. Conditional substack index pop}} \\ \underbrace{\Psi^{(1)}_{\text{id}}(S^1), \ldots, \Psi^{(1)}_{\text{id}}(S^p)}_{\text{5. Conditional stack id}} \\ \underbrace{\Psi^{(2)}_{\text{push0}}(S^1), \ldots, \Psi^{(2)}_{\text{push0}}(S^p)}_{\text{6. Conditional stack push-0}} \\ \underbrace{\Psi^{(3)}_{\text{push1}}(S^1), \ldots, \Psi^{(3)}_{\text{push1}}(S^p)}_{\text{7. Conditional stack push-1}} \\ \underbrace{\Psi^{(4)}_{\text{pop}}(S^1), \ldots, \Psi^{(4)}_{\text{pop}}(S^p)}_{\text{8. Conditional stack pop}} \\ \underbrace{\Psi^{(5)}_{\text{pass-push}}(S^1), \ldots, \Psi^{(5)}_{\text{pass-push}}(S^p)}_{\text{9. Passthrough substack active in push}} \\ \underbrace{\Psi^{(6)}_{\text{pass-active}}(S^1), \ldots, \Psi^{(6)}_{\text{pass-active}}(S^p)}_{\text{10. Passthrough inactive substack}} \end{bmatrix} \in \mathbb{R}^{|Q|+9s\cdot p}$$

This architecture ensures that Layer 4 faithfully implements the transition function $\delta : Q \times [s]^p \times (\Sigma \cup \{\varnothing\})^{ps} \to Q \times [s]^p \times \{\mathsf{push}_0, \mathsf{push}_1, \mathsf{pop}, \mathsf{id}\}^{ps}$ of the p-stack machine, where each component of the output vector corresponds to a specific aspect of the machine's next configuration.

3.3.6. Layer 5: Output Formatting

The final layer reconstructs the machine's state representation by selectively combining the operational components computed in Layer 4, ensuring the output maintains the same structure as the input for recursive application:

$$W_5 = \begin{bmatrix} \underbrace{\mathbf{I}_{|Q|}}_{\text{State passthrough}} & \underbrace{\mathbf{0}_{|Q|\times 9s\cdot p}}_{\text{Zero padding}} \\ \underbrace{\mathbf{0}_{2s\cdot p\times|Q|}}_{\text{Zero padding}} & \underbrace{W_{\text{select}}}_{\text{Selective combination}} \end{bmatrix} \in \mathbb{R}^{(|Q|+2s\cdot p)\times(|Q|+9s\cdot p)}$$

$$b_5 = \mathbf{0}_{|Q|+2s\cdot p} \in \mathbb{R}^{|Q|+2s\cdot p}$$

Selective Combination Matrix Construction The selection matrix $W_{\text{select}} \in \mathbb{R}^{2s\cdot p\times 9s\cdot p}$ implements a structured combination of the nine operational pathways from Layer 4. Rather than complex aggregation, this layer performs selective identity mappings across the operational components:

$$W_{\text{select}} = \begin{bmatrix} W_{\text{index-select}} \\ W_{\text{stack-select}} \end{bmatrix}$$

where:

1. **Substack Index Selection** ($W_{\text{index-select}} \in \mathbb{R}^{s\cdot p\times 9s\cdot p}$): The first $s \cdot p$ rows combine the three index operation types (identity, push, pop) from components 2-4 of Layer 4's output:
$$W_{\text{index-select}} = \begin{bmatrix} \mathbf{I}_{s\cdot p} & \mathbf{I}_{s\cdot p} & \mathbf{I}_{s\cdot p} & \mathbf{0}_{s\cdot p\times 6s\cdot p} \end{bmatrix}$$
This ensures that exactly one of the three index update modes (identity, push shift, or pop shift) becomes active based on the operation determined in Layer 4.
2. **Stack Content Selection** ($W_{\text{stack-select}} \in \mathbb{R}^{s\cdot p\times 9s\cdot p}$): The remaining $s\cdot p$ rows combine all six stack operation types from components 5-10 of Layer 4's output:
$$W_{\text{stack-select}} = \begin{bmatrix} \mathbf{0}_{s\cdot p\times 3s\cdot p} & \mathbf{I}_{s\cdot p} & \cdots & \mathbf{I}_{s\cdot p} \end{bmatrix}$$
This combines the identity operations, push-0 operations, push-1 operations, pop operations, push passthrough, and inactive substack passthrough into the final stack representation.

The zero bias vector ensures that the combination is purely additive without threshold effects, allowing the CReLU activations from Layer 4 to determine which pathways contribute to the final state.

After this transformation, the output vector $\mathbf{x}^{(6)}$ has exactly the same structure as the input vector $\mathbf{x}^{(1)}$:

$$\mathbf{x}^{(6)} = \begin{bmatrix} \underbrace{e_{\iota_{q'}}}_{\text{Updated main state}} \\ \underbrace{e_{\iota'_1}, \ldots, e_{\iota'_p}}_{\text{Updated substack indices}} \\ \underbrace{\mathsf{enc}(S'^1), \ldots, \mathsf{enc}(S'^{s\cdot p})}_{\text{Updated encoded stacks}} \end{bmatrix} \in \mathbb{R}^{|Q|+2s\cdot p}$$

3.4. From CReLU to RNN

The last step is to transform our CReLU neural network into a trainable RNN architecture for sequential data processing. The core challenge lies in bridging the gap between continuous neural network outputs and discrete stack operations while maintaining computational tractability.

The overall architecture (Figure 2) operates on state vectors where stack contents are first encoded according to Definition 3.13, and the state are initialized with one-hot encoding.

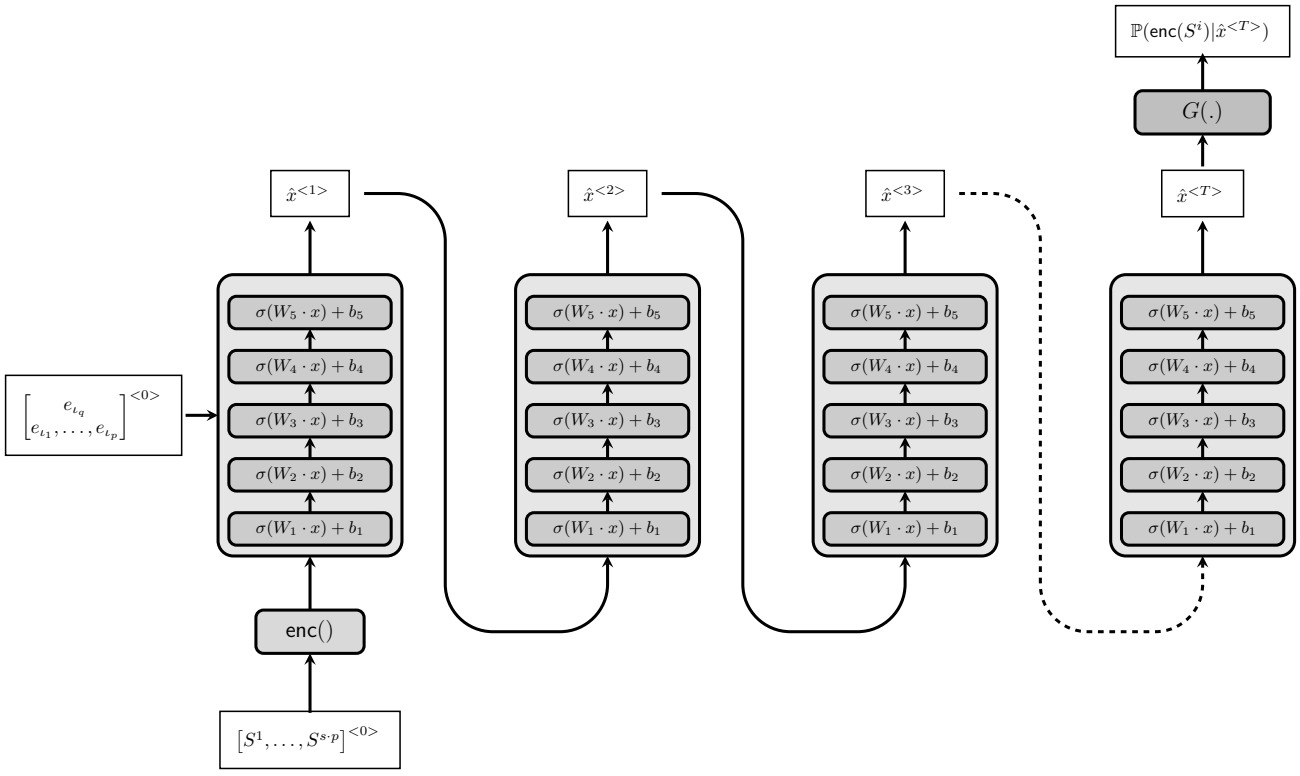


Figure 2. RNN architecture implementing a p-stack machine with split factor s. The 5-layer CReLU network processes state vectors $\hat{x}^{\langle t \rangle}$ through time steps $t = 0, 1, \ldots, T$. The encoding function $\mathsf{enc}()$ converts stack contents to Cantor set representations, while $G(\cdot)$ generates probability distributions $\mathbb{P}(\mathsf{enc}(S^i)|\hat{x}^{\langle T \rangle})$ over possible stack symbols.

3.4.1. RNN Architecture Implementation

As illustrated in Figure 2, the recurrent structure applies the 5-layer CReLU network $\mathcal{N}_{\text{CReLU}}$ at each time step t:

$$\hat{x}^{\langle t+1 \rangle} = \mathcal{N}_{\text{CReLU}}(\hat{x}^{\langle t \rangle})$$

The state vector $\hat{x}^{\langle t \rangle}$ contains both the finite automaton state and the encoded stack representations.

3.4.2. Output Generation

The final hidden state $\hat{x}^{\langle T \rangle}$ is mapped to probability distributions over Cantor set elements through a learned function $G(\cdot) : \mathbb{R}^{|Q|+2s \cdot p} \to \Delta^{|C|}$, where $\Delta^{|C|}$ denotes the probability simplex over the Cantor set C.

Gaussian Embedding for Continuous-Discrete Mapping Since continuous network outputs rarely coincide with discrete Cantor elements, we employ a Gaussian embedding scheme for differentiable mapping. Stack values are extracted from the final state:

$$\mathbf{v}_{\text{stacks}} = [\hat{x}^{\langle T \rangle}_{|Q|+s \cdot p+1}, \ldots, \hat{x}^{\langle T \rangle}_{|Q|+2s \cdot p}] \in \mathbb{R}^{s \cdot p}$$

For each substack value v_i, probabilities over Cantor elements $c_j \in C$ are computed using:

$$\phi_{i,j}(v_i) = \exp\left(-\frac{(v_i - c_j)^2}{2\sigma^2}\right)$$

$$\mathbb{P}(\mathsf{enc}(S^i) = c_j|\hat{x}^{\langle T \rangle}) = \frac{\exp(\beta \cdot \phi_{i,j}(v_i))}{\sum_{k=1}^{|C|} \exp(\beta \cdot \phi_{i,k}(v_i))}$$

where $\sigma = \sigma_{\text{base}}/2^b$ scales with bit depth b, and β controls the distribution sharpness. Hyperparameter selection strategies are detailed in Appendix A.

As illustrated in Figure 3, this embedding scheme maps continuous input values to distributed activations across multiple Gaussian functions centered at Cantor set elements. The approach ensures: (1) smooth, differentiable gradients for training, and (2) non-zero probability coverage across the encoding space. When an input value falls between Cantor elements, the Gaussian functions create a natural probability distribution that peaks at the nearest neighbors.

Proposition 3.18 (Exponential to Linear Scaling via Stack Splitting)**.** *Without stack splitting, predicting probabilities*

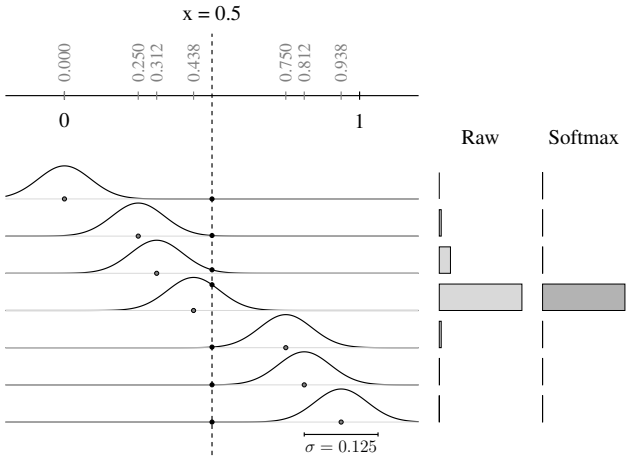


Figure 3. Gaussian embedding for continuous-to-discrete mapping. Cantor set elements serve as Gaussian centers. Input value $x = 0.3$ (dashed line) generates distributed activations across multiple Gaussians, with softmax normalization producing the final probability distribution.

over all possible stack combinations requires:

$$|Output|_{orig} = O(p \cdot 2^b) \tag{1}$$

Stack splitting reduces this to:

$$|Output|_{split} = O(p \cdot s \cdot 2^{b/s}) \tag{2}$$

where b denotes the number of bits used to represent each stack. Achieving linear scaling $O(p \cdot b)$ when $s = \Theta(b)$.

3.5. Training

We employ teacher forcing (Williams & Zipser, 1989) with intermediate supervision during training, utilizing ground-truth targets $\hat{x}^{\langle t \rangle}_{\text{target}}$ instead of network predictions for intermediate computational steps. This approach stabilizes the training process by decomposing sequential learning into independent feedforward mappings at each time step while preserving the underlying sequential dependencies inherent to the stack machine dynamics.

Proposition 3.19 (Teacher Forcing vs. Autoregressive Loss Difference)**.** *When using the true target state $\hat{x}^{\langle t \rangle}_{target}$ instead of the predicted state $\hat{x}^{\langle t \rangle}_{pred}$, and approximating the CReLU network as a linear transformation $\mathcal{N}_{CReLU}(\hat{x}^{\langle t \rangle}) \approx W\hat{x}^{\langle t \rangle}$, the gradient scaling factor between teacher forcing and autoregressive training differs by:*

$$\frac{\partial \mathcal{L}^{\langle t \rangle}_{teacher}}{\partial \theta} \Big/ \frac{\partial \mathcal{L}^{\langle t \rangle}_{autoregressive}}{\partial \theta} \propto (W^T)^{t-1}$$

This transformation effectively mitigates the risk of vanishing and exploding gradients that typically plague long-sequence training.

This formulation transforms sequential RNN training into a collection of supervised feedforward problems, learning independent mappings $\hat{x}^{\langle t \rangle} \mapsto \hat{x}^{\langle t+1 \rangle}$ for each time step. While this approach sacrifices some sequential consistency during the training phase, it significantly improves convergence stability and enables efficient parallel computation across temporal dimensions.

Loss Function We minimize the cross-entropy loss to reduce prediction error across all substacks and time steps:

$$\mathcal{L} = -\sum_{t=1}^{T} \sum_{i=1}^{s \cdot p} \sum_{j=1}^{|C|} y_{t,i,j} \log \mathbb{P}(\mathsf{enc}(S^i) = c_j | \hat{x}^{\langle t \rangle})$$

where $y_{t,i,j} \in \{0, 1\}$ represents the one-hot encoded ground-truth target for substack i at time step t, and $c_j \in C$ denotes the j-th element in the Cantor set encoding.

Weight Initialization Strategy

- **Bias Initialization**: Set bias to zero to prevent systematic boundary violations that could destabilize the Cantor set encoding. Non-zero biases can shift outputs outside the valid $[0, 1]$ interval, violating the fractal structure's geometric constraints.
- **Weight Initialization**: Employ He initialization (He et al., 2015) to maintain proper variance scaling for ReLU-based activations. The $w \sim \mathcal{N}(0, \sqrt{2/n_{in}})$ scaling prevents vanishing/exploding gradients and ensures the network can learn precise stack operation.

Scheduler We implement a warmup-exponential learning rate scheduler with the Adam optimizer, consisting of two phases:

- **Warmup Phase**: Linear increase from 1×10^{-5} to 0.01 over 40 training steps.
- **Exponential Decay Phase**: Exponential decay over 500 cooldown steps to 5×10^{-5}.

Regularization We apply L_1 regularization to encourage sparsity in the learned representations:

$$\mathcal{L}_{\text{total}} = \mathcal{L} + \lambda \sum_{i=0}^{4} (\|W_i\|_1 + \|b_i\|_1)$$

where $\lambda = 0.005$ is the regularization strength, W_i denotes the weight matrix of the i-th layer, and b_i denotes the corresponding bias vector.

Computational Setup Training is performed on 4 NVIDIA TITAN X (Pascal) GPUs using TensorFlow's mirrored distributed strategy. With 32K training samples and batch size 256, each epoch requires approximately 10 minutes of computation time.

4. Results

We evaluate our approach on a comprehensive dataset of 13 p-stack automata implementing fundamental arithmetic and bitwise logical operations. The evaluation suite includes:

- **Arithmetic Operations**: Addition (add), subtraction (subtract), multiplication (multiply), modulo arithmetic (modulo), negation (neg)
- **Bitwise Logical Operations**: AND (bitwise_and), OR (bitwise_or), NAND (bitwise_nand), XOR (bitwise_xor)
- **Bit Shift Operations**: Left shift (lshift), right shift (rshift)
- **Stack Operations**: Stack copying (copy), constant loading (const)

Figure 4 illustrates the structural complexity distribution of our evaluation suite, characterized by state-transition relationships.

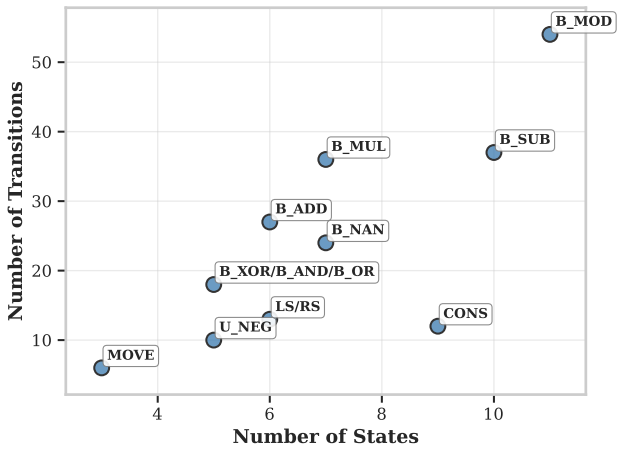


Figure 4. Distribution of state-transition complexity across the 13 synthesized p-stack automata. Each point represents a unique (states, transitions) configuration, with overlapping instruction types indicated using forward slash notation. The clustering pattern demonstrates structural similarity within instruction categories while maintaining diversity across the evaluation suite.

4.1. Experimental Setup

All experiments employ p-stack machines with split factor $s = 6$ and bitstring length $b = 12$. Training consists of 32,000 samples distributed across 240 optimization steps using L1 regularization with $\lambda = 0.005$. Each configuration is evaluated across all 13 automata to ensure statistical significance. Hyperparameters are optimized via Bayesian optimization over 256 independent runs.

To investigate the conditions necessary for learning exact theoretical weights, we examine three initialization paradigms: (1) **Random initialization** using He initialization as detailed in Section 3.5, (2) **Hybrid initialization** where all layers receive perfect weights except layer ℓ, which uses He initialization (Aut.+He), and (3) **Noisy perfect initialization** where all layers are initialized with theoretical optima corrupted by Gaussian noise with standard deviation σ_{noise}.

4.2. Weight Initialization Impact

Table 1 reveals substantial performance variations across initialization strategies, highlighting the critical role of proximity to theoretical optimal weights.

Table 1. Validation accuracy across initialization strategies. Results represent mean ± standard deviation over 13 p-stack automata. Parameter ℓ indicates the randomly initialized layer in hybrid approaches.

	INIT METHOD	ℓ	σ_{NOISE}	ACC (%)
A1	HE			71.7±9.0
B1	AUT + HE	1		90.0±3.0
B2		2		88.1±5.4
B3		3		88.2±4.9
B4		4		93.7±1.6
B5		5		92.5±1.8
C1	NOISY		0.1	90.5±3.3
C2			0.2	78.7±10.8
C3			0.5	50.7±8.7
C4			1.0	46.5±9.4

Pure He initialization (A1) fails to converge to perfect weights, achieving only 71.7% accuracy despite extensive training. The hybrid approach (B1-B5) improves performance to 88-94% by initializing most layers with perfect weights, but still becomes trapped in local minima after 100 steps. The choice of randomly initialized layer has minimal impact on performance.

Noisy perfect initialization (C1-C4) demonstrates clear degradation with increasing noise: small perturbations ($\sigma_{\text{noise}} = 0.1$) maintain 90.5% accuracy, while larger noise severely degrades performance. Importantly, noisy runs did not fully converge within 240 steps, suggesting longer training might improve results.

Rather than extending training duration, we analyze the loss landscape around the theoretical optimum to understand the geometric constraints preventing optimization success.

4.3. Loss Landscape Characterization

To investigate the initialization sensitivity observed in Table 1, we conduct a systematic analysis of the loss landscape

structure. Figure 5 presents a radial perturbation analysis around the theoretical optimum, evaluating loss values across 64 random noise vectors with varying magnitudes.

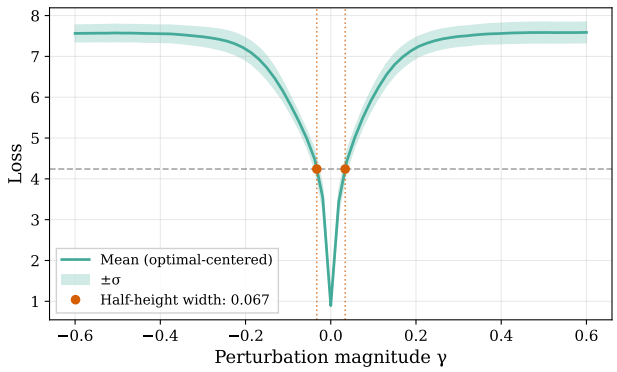


Figure 5. Loss landscape structure around theoretical optimal weights. The plot shows mean loss and standard deviation against perturbation magnitude γ, revealing characteristic V-shaped behavior. Orange markers indicate half-height width measurements, quantifying the sharpness of the optimum. Results averaged over 64 random perturbation directions demonstrate the narrow basin of attraction surrounding perfect weights.

The analysis reveals a characteristic V-shaped loss profile with steep gradients in the immediate vicinity of optimal weights. As perturbation magnitude increases, gradients diminish rapidly, creating a narrow basin of attraction. This geometry explains why optimization from distant initializations fails, the diminishing gradient signal provides insufficient direction toward the global minimum, leading to entrapment in local minima.

When we examine the 2D plane spanned by the initialized weights, optimal weights, and trained weights with He initialization, Figure 6 reveals that the model converges to a nearby local minimum rather than the global optimum. Notably, the gradient slope toward the perfect weights is relatively shallow at initialization, creating a challenging optimization landscape. This behavior is consistent across different random seeds, where models consistently settle into local minima instead of reaching the theoretical optimum, explaining the accuracy gaps observed in Table 1, Row A1.

We analyzed the convergence properties of models initialized with 64 different random seeds, where weights were initialized with Gaussian noise ($\sigma = 0.4$) and trained until accuracy convergence. To assess whether trained models reached local minima, we applied first- and second-order optimality conditions: (1) $\|\nabla L(\theta^*)\| \approx 0$ and (2) $H(\theta^*) \succeq 0$.

Although all models achieve accuracy convergence, the gradient norms range from 0.0 to 5.1 (mean: 2.5 ± 0.3), significantly exceeding the first-order optimality threshold

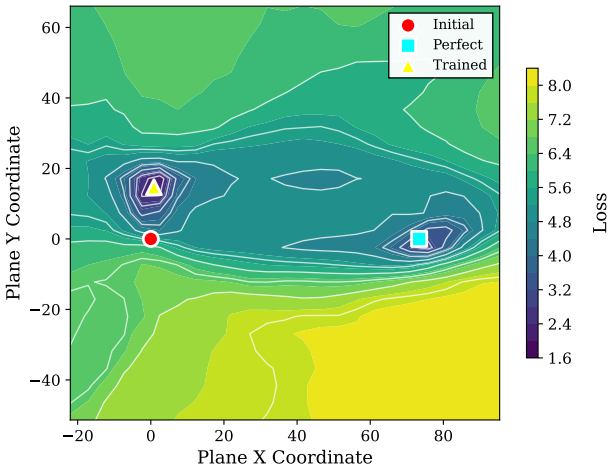


Figure 6. Loss landscape visualization in the 2D plane defined by initial, perfect, and trained weight configurations. The filled contour plot shows the loss topology with darker regions indicating lower loss values. Key points are marked: initial weights (red circle), perfect weights extracted from the target automaton (cyan square), and final trained weights (yellow triangle). The contour lines reveal the gradient structure, showing how the model gets trapped in local minima despite the proximity to the global optimum.

($\|\nabla L\| < 10^{-3}$). However, Hessian analysis shows exclusively positive eigenvalues with ranges $[513, 1026]$, indicating well-conditioned basins of attraction for local minima.

5. Conclusion

This work presents three key contributions: (1) a method for parallelizing p-stack machines through substack decomposition, (2) systematic translation of multistate p-stack machines into gradient-optimizable RNNs, and (3) identification of fundamental barriers preventing RNNs from learning exact mathematical operations with proposed solutions.

Our parallelization framework enables concurrent execution of stack-based computations, while our translation methodology provides a principled bridge between discrete computational models and continuous neural architectures. Most significantly, we reveal that RNNs struggle with exact mathematical operations due to specific geometric and representational challenges in the loss landscape. Our optimization and initialization strategies represent initial steps toward addressing these limitations.

5.1. Future Work

Several directions warrant investigation: **Solution Space Characterization** of optimal weight geometries could in-

form efficient optimization strategies. **Stack Encoding Optimization** is needed as our current approach may lose information during deep operations—developing uniform representational schemes across stack positions could improve robustness. **Automated Extraction** for complex operations like eigenvalue decomposition remains challenging; our Python-to-p-stack translation shows promise but requires optimization for parallelism and state minimality.

Optimization Enhancement represents a critical research direction given our findings on local minima convergence. The main focus should lie on developing new regularization techniques, loss functions, initialization strategies, and optimization algorithms specifically designed to overcome local minima traps. Additionally, architectural improvements to the neural network models could fundamentally alter the loss landscape topology, potentially reducing the prevalence of suboptimal local minima.

Advanced Training methods including curriculum learning and controlled stochasticity in intermediate computations could further improve convergence toward global optima.

Acknowledgment

I am deeply grateful to Valentin Abadie for his indispensable contributions and support throughout this work. Valentin's valuable insights and generation of key ideas were central to this paper's completion. His openness to novel approaches and constant readiness to offer support during challenges were immensely valuable and deeply appreciated.

Additionally, I acknowledge the use of large language models during the writing process. ChatGPT-4 (OpenAI, May 2024 version), Claude 3 Opus (Anthropic), and DeepSeek-R1 (DeepSeek AI) were employed for language refinement, technical paraphrasing, and readability improvement. The author has independently verified all content and takes full responsibility for the final work.

References

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015. URL https://arxiv.org/abs/1502.01852.

Neeser, J. Computation of turing machine transitions with neural networks. Master's thesis, Eidgenössische Technische Hochschule Zürich (ETH Zurich), Zurich, Switzerland, January 2023. Supervised by Valentin Abadie.

Siegelmann, H. and Sontag, E. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, 1995. ISSN 0022-0000. doi: https://doi.org/10.1006/jcss.1995.1013. URL https://www.sciencedirect.com/science/article/pii/S0022000085710136.

Siegelmann, H. T. and Sontag, E. D. Analog computation via neural networks. *Theoretical Computer Science*, 131(2):331–360, 1994. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(94)90178-3. URL https://www.sciencedirect.com/science/article/pii/0304397594901783.

Wang, Y., Liu, X., and Shi, S. Deep neural solver for math word problems. In Palmer, M., Hwa, R., and Riedel, S. (eds.), *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 845–854, Copenhagen, Denmark, September 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1088. URL https://aclanthology.org/D17-1088/.

Williams, R. J. and Zipser, D. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989. doi: 10.1162/neco.1989.1.2.270.

A. Hyperparameter Sensitivity Analysis

To evaluate the sensitivity of our RNN model described in Section 3.4, we conducted a systematic hyperparameter analysis around a carefully chosen base configuration.

A.1. Experimental Setup

The base model configuration implements a p-stack machine with split factor $s = 6$, bitstring length $b = 12$, and modified CReLU activation with saturation parameter $\vartheta = 0.9$. Training utilizes He weight initialization with 10,000 samples over 100 training steps and L1 regularization strength $\lambda = 0.005$. The Gaussian embedding uses a base standard deviation of $\sigma_{\text{base}} = 1.0$.

Table 2. Hyperparameter sensitivity analysis on model performance. Results show validation metrics (mean ± standard deviation) averaged across 3 random seeds on the add p-stack task. Unless otherwise specified, all hyperparameters match the base model configuration. Metrics: ACC (accuracy), MSE (mean squared error of $\hat{x}^{\langle T \rangle}$ scaled by 10^{-1}), PPL (perplexity).

CONFIG	s	b	ϑ	σ_{BASE}	INIT METHOD	N_{SAMPLES}	λ	ACC (%)	MSE ($\times 10$)	PPL
BASE	6	12	0.9	1	HE	10K	0.005	59.88±1.56	0.97±0.04	71±3
A1	1							38.42±13.05	0.89±0.61	112±67
A2	2							55.08±4.59	1.37±0.06	122±58
A3	4							60.42±2.24	1.12±0.04	79±9
A4	6							62.20±1.10	0.96±0.04	73±7
A5	8							65.38±1.85	0.92±0.01	79±17
A6	12							70.90±0.82	0.79±0.04	73±5
B1					GAUSSIAN			42.01±0.27	1.43±0.02	120±3
B2					GLOROT			52.87±0.19	1.36±0.03	122±6
B3					HE			61.37±0.74	0.97±0.04	72±4
B4					UNIFORM			43.21±4.88	2.00±0.33	274±38
C1		2						92.65±1.16	0.62±0.02	18±4
C2		4						87.27±3.71	0.68±0.06	26±5
C3		8						70.92±0.86	1.01±0.06	71±14
C4		12						61.53±0.72	0.96±0.04	75±13
C5		16						54.40±2.75	0.98±0.05	85±13
C6		32						41.30±2.85	0.95±0.04	66±2
D1				0.1				39.25±4.66	1.60±0.15	27±14
D2				0.5				54.77±3.32	1.23±0.04	54±2
D3				1.0				61.31±1.48	0.97±0.02	74±5
D4				2.0				51.40±0.87	0.94±0.02	14±4
D5				5.0				46.52±0.68	1.03±0.04	3±0
E1							0.0	68.52±0.91	0.56±0.04	45±4
E2							1E-05	67.28±0.74	0.57±0.05	48±9
E3							0.0001	68.18±1.09	0.58±0.04	43±1
E4							0.001	69.18±0.82	0.75±0.03	52±9
E5							0.01	53.31±0.69	1.14±0.02	89±5
E6							0.1	42.66±0.39	1.57±0.03	124±6
F1			0.0					51.53±2.11	1.19±0.05	123±12
F2			0.2					52.86±1.76	1.15±0.07	115±21
F3			0.4					53.96±0.66	1.03±0.05	94±15
F4			0.6					55.91±2.18	1.02±0.05	84±11
F5			0.8					59.83±1.38	0.97±0.02	79±8
F6			1.0					61.84±0.74	0.97±0.05	79±13

A.2. Split Factor Analysis (s)

The split factor parameter s determines how many parallel stacks are used in the p-stack architecture. Results in Table 2 rows A1-A6 demonstrate a clear performance improvement with increasing split factor values. The model achieves 38.4% accuracy with $s = 1$ but reaches 70.9% accuracy with $s = 12$. This improvement is intuitive: larger split factors prevent the

model from requiring excessively deep stacks, as deeper stacks store information in less significant precision bits that are more susceptible to numerical degradation. The consistent reduction in both MSE and perplexity with increasing split factor confirms this hypothesis.

A.3. Weight Initialization Method Analysis

The choice of weight initialization scheme significantly impacts model convergence and final performance. As shown in Table 2 rows B1-B4, He/Kaiming initialization substantially outperforms other methods, achieving 61.4% accuracy compared to 42.0% for Gaussian initialization, 52.9% for Glorot/Xavier, and only 43.2% for uniform initialization. The superior performance of He initialization stems from its ability to initialize weights within a range that places initial predictions inside the target distribution. Other initialization schemes may cause predictions to saturate at activation function boundaries, where gradients become vanishingly small, particularly problematic when the saturation parameter ϑ is large.

A.4. Bitstring Length Analysis (b)

The bitstring length b controls the precision of information storage within each stack element. Results in Table 2 rows C1-C6 reveal an inverse relationship between bitstring length and model performance. Shorter bitstrings achieve superior results: $b = 2$ yields 92.7% accuracy while $b = 32$ drops to 41.3% accuracy. This counterintuitive finding aligns with the split factor analysis—shorter bitstrings reduce the risk of information loss in deeper stack positions. The dramatic improvement with shorter bitstrings suggests that the model benefits more from multiple shallow, low-precision stacks than from fewer deep, high-precision stacks.

A.5. Gaussian Embedding Standard Deviation Analysis (σ_{base})

The base standard deviation σ_{base} of the Gaussian embedding provides direct control over model output uncertainty, as measured by perplexity. Results in Table 2 rows D1-D5 show that $\sigma_{\text{base}} = 1.0$ achieves optimal accuracy (61.3%) while maintaining reasonable perplexity (74). Smaller values ($\sigma_{\text{base}} = 0.1$) produce very low perplexity (27) but sacrifice accuracy (39.3%), indicating overconfident predictions. Larger values ($\sigma_{\text{base}} = 5.0$) yield extremely low perplexity (3) but poor accuracy (46.5%), suggesting the model becomes overly certain about incorrect predictions. The optimal value of 1.0 represents a balanced trade-off between prediction confidence and accuracy.

A.6. L1 Regularization Strength Analysis (λ)

L1 regularization strength controls model complexity and overfitting. The analysis in Table 2 rows E1-E6 reveals that minimal regularization yields optimal performance. Without regularization ($\lambda = 0.0$), the model achieves 68.5% accuracy, while very light regularization ($\lambda = 10^{-5}$ to $\lambda = 10^{-3}$) maintains similar performance. However, stronger regularization significantly degrades performance: $\lambda = 0.01$ reduces accuracy to 53.3%, and $\lambda = 0.1$ further drops it to 42.7%. The increasing MSE and perplexity with stronger regularization indicate that the model benefits from retaining its full representational capacity for this task.

A.7. Saturation Parameter Analysis (ϑ)

The saturation parameter ϑ controls the interpolation between pure CReLU and sigmoid-like activation behaviors. Since the standard CReLU activation has zero gradient outside the interval $[0, 1]$, we employ a differentiable sigmoid-like variant that maintains gradient flow throughout the entire domain:

$$\text{CReLU}_\vartheta(x) = \vartheta \cdot \text{clip}(x, 0, 1) + (1 - \vartheta) \cdot \sigma(4(x - 0.5)) \tag{3}$$

where $\text{clip}(x, 0, 1) = \max(0, \min(x, 1))$ represents the standard CReLU activation, and $\sigma(\cdot)$ denotes the standard sigmoid function.

Figure 7 demonstrates how the saturation parameter ϑ modulates the activation function shape, transitioning from pure sigmoid behavior ($\vartheta = 0$) to pure CReLU behavior ($\vartheta = 1$). Our experimental results in Table 2 (rows F1-F6) reveal a clear performance trend: when using He initialization, higher values of ϑ consistently improve model performance across all metrics. Specifically, accuracy increases monotonically from 51.53% at $\vartheta = 0.0$ to 61.84% at $\vartheta = 1.0$, while MSE

decreases and remains stable at higher saturation values.

This performance degradation at lower ϑ values can be attributed to the mismatch between the He initialization scheme and the modified activation function. He initialization is specifically designed for ReLU-like activations, placing weights within an optimal range for the sharp, piecewise-linear characteristics of CReLU. When ϑ is reduced, the sigmoid component smooths the activation function, disrupting this carefully calibrated weight distribution and leading to suboptimal gradient flow during training. The results suggest that maintaining the sharp transition characteristics of CReLU ($\vartheta \geq 0.8$) is crucial for optimal performance when using standard initialization methods.

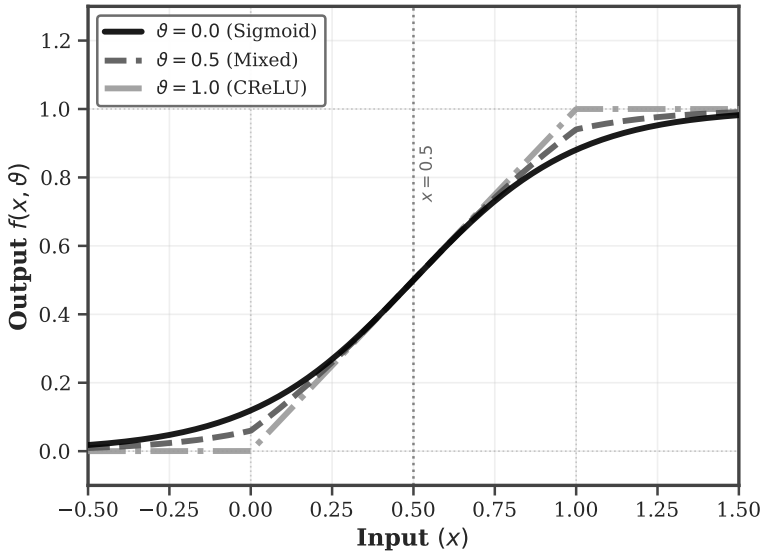


Figure 7. Effect of saturation parameter ϑ on activation function. The parameter interpolates between pure CReLU activation ($\vartheta = 0$) and sigmoid-like behavior ($\vartheta = 1$).

A.8. State Complexity vs Performance Analysis

We evaluated the relationship between automaton complexity and performance across different p-stack machines. Results in Table 3 show that complexity does not correlate directly with learning difficulty.

The most complex automaton (Binary Modulo: 11 states, 54 transitions) achieves high accuracy at 92.0%, while the simplest (Move: 3 states, 6 transitions) reaches 90.6%. Operations with identical complexity show varying performance: binary logical operations (AND, OR, XOR) with 5 states and 18 transitions range from 82.1% to 87.3% accuracy. The shift operations achieve the highest performance (92.7% and 92.5%).

The Const operation shows the poorest results (62.3%) despite moderate complexity. Unlike other operations, Const lacks computational logic and simply uses states as memory without exploiting any computational patterns, making it fundamentally different from pattern-based operations.

These findings suggest that computational pattern matters more than structural complexity, and that moderate complexity can provide beneficial representational capacity without hindering learnability.

Table 3. Performance of different p-stack machines with state complexity

p-STACK MACHINE	N_STATES	N_TRANSITIONS	ACC (%)
MOVE	3	6	90.6±1.2
BINARY AND	5	18	87.3±3.5
BINARY OR	5	18	82.1±1.8
BINARY XOR	5	18	86.7±3.4
NEGATION	5	10	84.4±1.2
BINARY ADDITION	6	27	87.1±1.1
LEFT SHIFT	6	13	92.7±0.4
RIGHT SHIFT	6	13	92.5±0.4
BINARY MULTIPLICATION	7	36	87.0±1.9
BINARY NAND	7	24	93.1±1.2
CONST	9	12	62.3±4.1
BINARY SUBTRACTION	10	37	86.4±4.1
BINARY MODULO	11	54	92.0±0.9

B. Example: CReLU Neural Network for Integer Addition

This section presents a concrete implementation of the theoretical construction described in Section 3.3 through a complete CReLU neural network that performs integer addition. The network implements a 3-stack pushdown automaton with split factor $s = 4$, resulting in 12 substacks total for binary arithmetic operations.

B.1. Problem Specification

The adder automaton processes two binary numbers through direct bitwise addition: (1) input digits onto stacks S^1 and S^2 (LSB first), (2) bitwise addition with carry propagation using states s_0 (no carry) and s_1 (carry), and (3) output result directly on stack S^3 without bit reversal. The automaton uses 3 control states $\{s_0, s_1, s_H\}$ and 18 transition rules handling all combinations of stack top symbols $\{0, 1, \$, X\}$ with carry state management.

B.2. Network Architecture

The resulting CReLU network has input/output dimension of 27 neurons ($3 + 2 \times 4 \times 3$), hidden layer dimensions of 51, 33, 48, 111 neurons respectively, totaling approximately 9,800 weights and 270 biases across 5 layers.

B.3. Weight and Bias Visualization

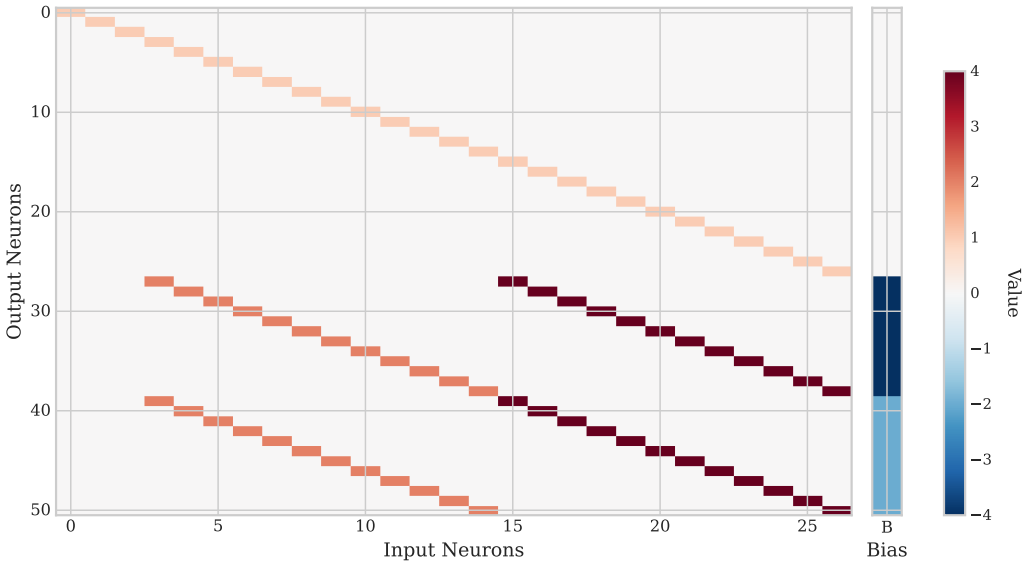


Figure 8. Layer 1: Conditional Top and Empty Bit Extraction. The weight matrix $W_1 \in \mathbb{R}^{51\times 27}$ implements identity passthrough for the upper block (3 states + 12 substack indices + 12 stack encodings) and conditional bit extraction with scaling factors 2 and 4 for the lower blocks. The bias vector contains threshold values -4 and -2 for top bit and non-empty detection respectively.

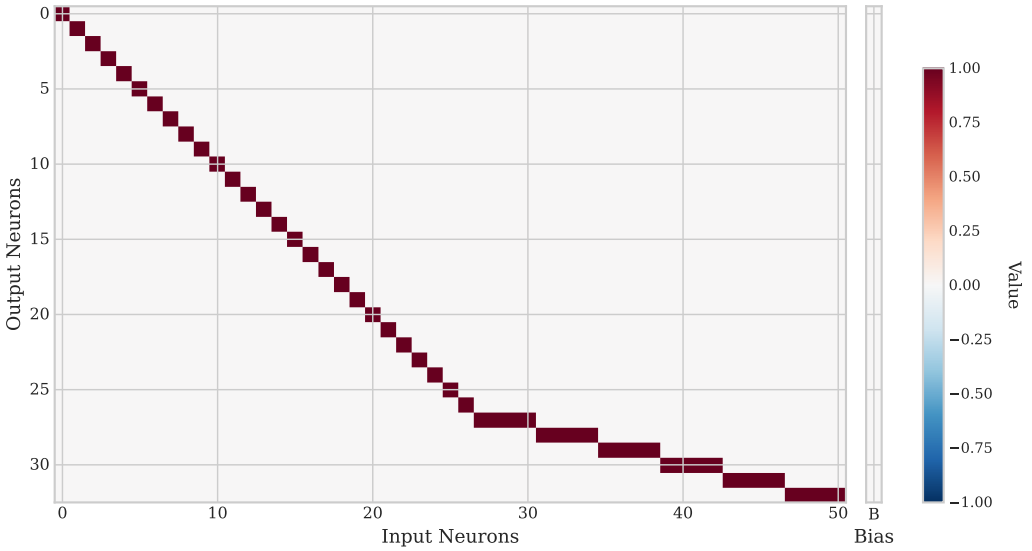


Figure 9. Layer 2: Substack Aggregation. The weight matrix $W_2 \in \mathbb{R}^{33\times 51}$ preserves state and stack information while aggregating conditional extractions across the 4 substacks for each of the 3 logical stacks. The horizontal stripe pattern in the lower block implements the substack-to-stack mapping with zero bias throughout.

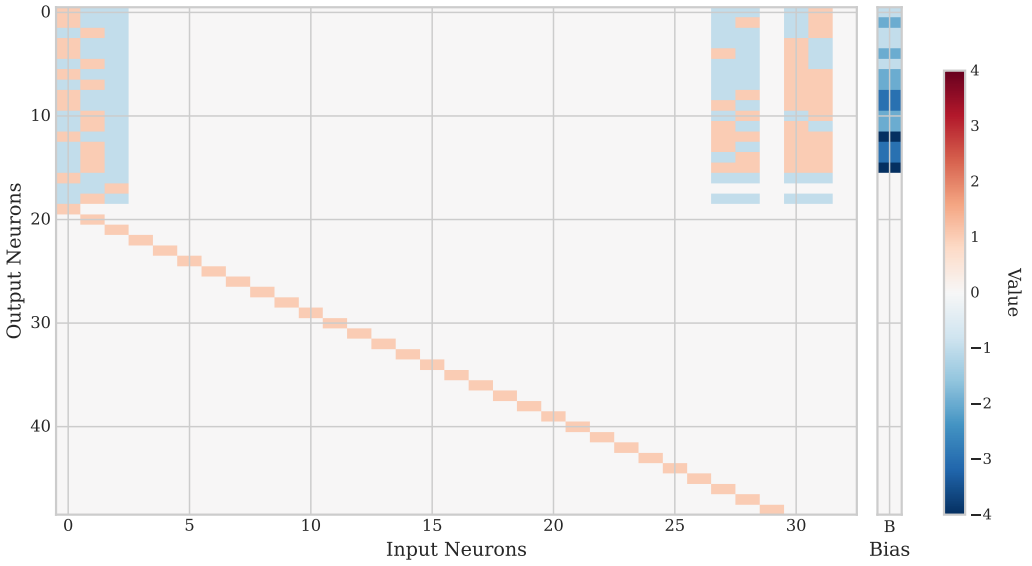


Figure 10. Layer 3: Pattern Matching. The weight matrix $W_3 \in \mathbb{R}^{48\times 33}$ implements transition rule recognition for the 18 distinct transition rules of the adder automaton. Each row in the upper block corresponds to a specific transition rule pattern matching current state and stack top configurations. The bias vector contains carefully calibrated thresholds for AND logic implementation.

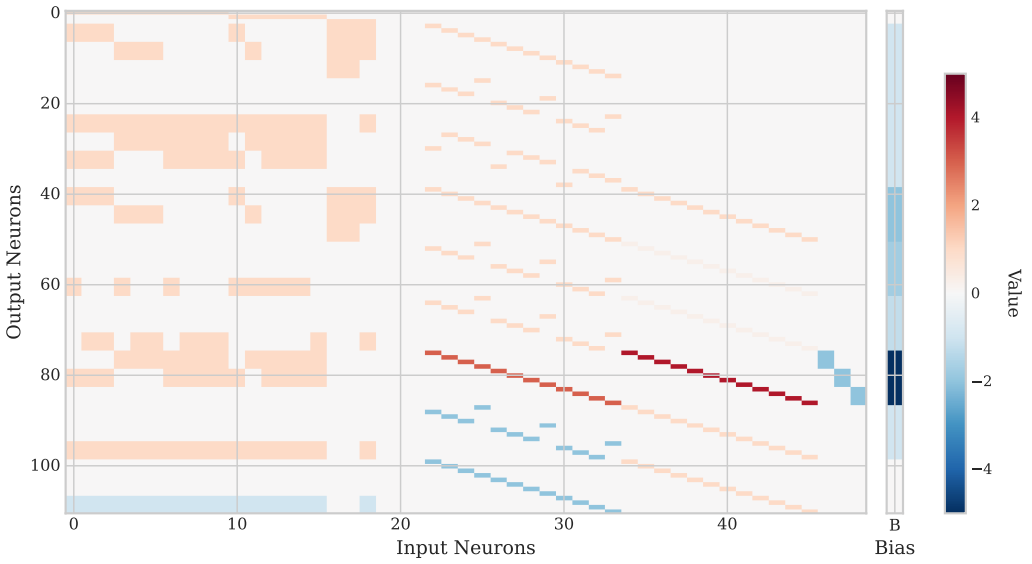


Figure 11. Layer 4: Operation Selection and Execution. The weight matrix $W_4 \in \mathbb{R}^{111\times 48}$ maps the 18 recognized patterns to concrete state transitions and stack operations across 3 stacks with 4 substacks each. The block structure implements state transitions (top), substack index updates (middle), and stack content modifications (bottom) with operation-specific scaling factors and biases.

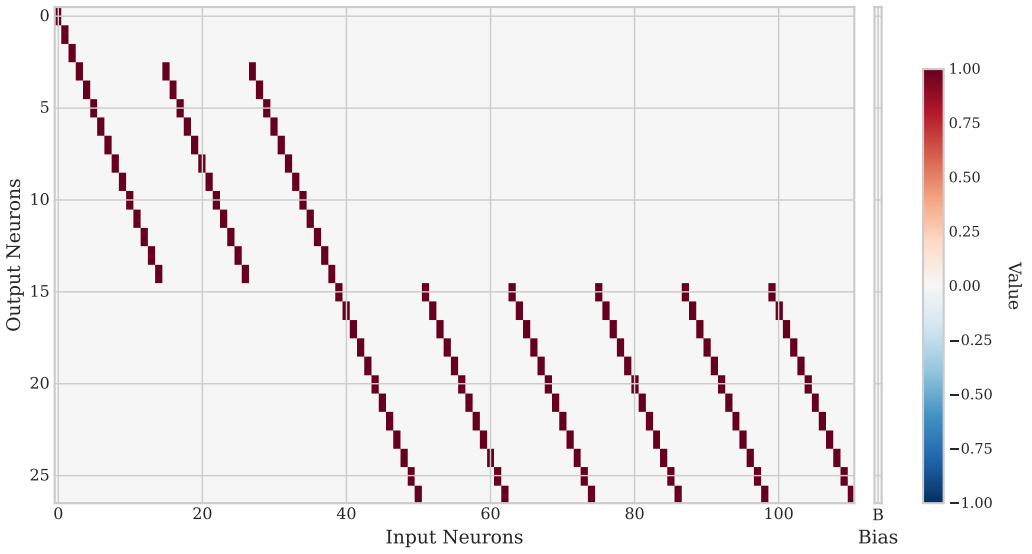


Figure 12. Layer 5: Output Formatting. The weight matrix $W_5 \in \mathbb{R}^{27\times 111}$ performs selective combination of operational pathways to reconstruct the final machine state. The block structure separates state passthrough (top 3 neurons) from substack index and stack content selection (remaining 24 neurons), with zero bias ensuring purely additive combination.

C. Extracting p-Stack Machines from Python Functions

This section details our systematic approach for translating mathematical operations implemented as Python functions into equivalent p-stack machines. The translation process operates through a multi-stage pipeline.

C.1. Translation Pipeline Architecture

Our extraction methodology follows a four-stage pipeline that. The process begins with Abstract Syntax Tree (AST) parsing to generate an intermediate instruction sequence, proceeds through structural analysis to determine resource requirements and control flow patterns, creates individual machines for each instruction, and concludes with machine composition that preserves execution semantics.

Algorithm 1 Python Function to p-Stack Machine Translation

Input: Python function source code
Output: Equivalent p-stack machine $\mathcal{P}$
tree $\leftarrow$ PARSEAST(source_code)
emitter $\leftarrow$ INSTREMITTER()
emitter.VISIT(tree)
$I \leftarrow$ emitter.GET_OPTIMIZED_CODE()
$p \leftarrow$ COMPUTEMAXSTACKS(I)
$P \leftarrow$ EXTRACTPARAMETERSTACKS(I)
$J \leftarrow$ ANALYZEJUMPSTRUCTURE(I)
$\mathcal{P} \leftarrow$ TRANSLATEFUNCTIONTOMACHINE(I, p, P, J) $\mathcal{P}$

C.2. AST-Based Instruction Generation

The first stage employs a custom `InstrEmitter` visitor that traverses the Python AST and generates a linear sequence of stack-based operations. This visitor maintains a comprehensive state tracking system that maps Python variables to numeric stack indices, manages temporary stack allocation for intermediate computations, and performs basic type inference to optimize subsequent processing stages.

The instruction generation process operates through recursive descent on the AST structure. When encountering binary operations, the visitor first processes both operands to determine their stack locations, allocates a temporary stack for the result, and emits the corresponding stack-based instruction. Similarly, variable references generate `LOAD_VAR` instructions that specify the stack index containing the variable's value, while function returns produce `RETURN` instructions indicating the stack containing the result value.

Algorithm 2 Binary Operation Processing

Function: PROCESSBINARYOPERATION(node)
VISIT(node.left)
left_idx $\leftarrow$ POP_STACK()
VISIT(node.right)
right_idx $\leftarrow$ POP_STACK()
result_idx $\leftarrow$ ALLOCATE_TEMP_STACK()
opcode $\leftarrow$ MAP_OPERATOR_TO_INSTRUCTION(node.op)
EMIT_INSTRUCTION(opcode, left_idx, right_idx, result_idx)
RELEASE_STACKS(left_idx, right_idx)
PUSH_STACK(result_idx)

Control flow constructs receive specialized treatment during instruction generation. Conditional statements trigger the creation of label-based jump instructions, where comparison operations generate boolean results that drive subsequent conditional jumps to appropriate code sections. The visitor maintains a label generation system that ensures unique identifiers for all jump targets while tracking which labels are actually referenced to enable dead code elimination.

C.3. Structural Analysis and Resource Computation

Following instruction generation, the system performs comprehensive analysis of the instruction sequence to extract structural information necessary for machine construction. The maximum stack requirement computation examines all instruction arguments to identify the highest stack index used, ensuring the final machine allocates sufficient stack resources for correct execution.

Parameter stack identification operates by scanning the instruction sequence for `LOAD_VAR` instructions, which indicate function parameters that must be loaded from specific stack locations. This analysis produces a set $P = \{i : (\texttt{LOAD\_VAR}, i) \in I\}$ representing all stack indices that contain input parameters, enabling proper machine initialization during execution.

Algorithm 3 Jump Structure Analysis

Input: Instruction sequence I
Output: Jump analysis structure J
J.labels_map $\leftarrow \emptyset$, J.conditional_jumps $\leftarrow \emptyset$
for each instruction $i_k \in I$ **do**
  if i_k.opcode $=$ `LABEL` **then**
    J.labels_map$[i_k$.target$] \leftarrow k$
  else if i_k.opcode contains `JUMP` **then**
    J.has_jumps $\leftarrow$ true
    Classify and record jump instruction
  end if
end for$J$

Jump structure analysis identifies control flow patterns by performing a two-pass examination of the instruction sequence. The first pass locates all label definitions and records their positions, while the second pass identifies jump instructions and classifies them as either conditional or unconditional transfers. This analysis constructs a mapping between symbolic labels and instruction indices, enabling subsequent control flow graph construction.

C.4. Individual Machine Creation

The third stage translates each instruction into a corresponding p-stack machine using specialized construction functions. These *maker* functions implement the low-level automaton logic required for each operation type, generating state machines that manipulate binary representations stored across the available stacks.

Arithmetic operations such as addition and multiplication require sophisticated state machines that implement bit-wise computation with proper carry propagation. The `BINARY_ADD` instruction, for instance, generates a machine containing states for each bit position during the addition process, with transitions encoding the complete addition truth table including carry handling between adjacent bit positions.

Algorithm 4 Instruction Machine Creation

Function: CREATEINSTRUCTIONMACHINE(instruction, p)
(opcode, args) $\leftarrow$ instruction
if opcode $\in$ ARITHMETIC_OPS **then**
  CREATEARITHMETICMACHINE(opcode, args, p)
else if opcode $\in$ CONTROL_FLOW_OPS **then**
  CREATEIDENTITYMACHINE(p) {Handled during merging}
else if opcode $\in$ DATA_MOVEMENT_OPS **then**
  null {No machine required}
else
  CREATEIDENTITYMACHINE(p) {Default fallback}
end if

Comparison operations generate machines that implement relational logic, producing boolean results stored in designated stack locations. These machines examine binary representations of their operands and generate appropriate true or false

values based on the comparison semantics. The resulting boolean values serve as inputs to subsequent conditional jump instructions during control flow evaluation.

Special instruction types receive customized handling during machine creation. Instructions such as `LOAD_VAR` and `RETURN` do not generate individual machines since they represent parameter initialization and result extraction respectively, operations handled at the machine composition level rather than through discrete automaton states.

C.5. Machine Composition and Control Flow Integration

The final stage combines individual instruction machines while preserving the execution semantics specified by the original Python function. For sequential execution without control flow, machines are composed through state connection where the halting state of machine M_i becomes the initial state of machine M_{i+1}, creating a unified execution path through all operations.

Functions containing conditional logic require sophisticated control flow handling that extends beyond simple sequential composition. The system constructs a control flow graph representing all possible execution paths, including conditional branches and loop structures. This graph drives the creation of specialized condition-checking states that evaluate boolean values stored in stacks and direct execution to appropriate machine sections based on the evaluation results.

Algorithm 5 Control Flow Aware Machine Merging

Input: Machine sequence M, jump information J, stack count p
Output: Unified machine $\mathcal{P}$
$G \leftarrow \textsc{BuildControlFlowGraph}(M, J)$
$Q \leftarrow \textsc{CreateStateSpace}(M, J)$
$\delta \leftarrow \emptyset$
$\textsc{AddSequentialTransitions}(\delta, G, Q)$
$\textsc{AddConditionalTransitions}(\delta, G, J, Q)$
$\textsc{AddUnconditionalJumps}(\delta, G, J, Q)$
$(Q, p, \{0, 1\}, \delta, q_0)$

Conditional jump handling requires the creation of specialized transition functions that examine stack contents and branch accordingly. For boolean conditions, transitions are generated for each possible stack state, with the machine following different execution paths based on whether the condition evaluates to true or false. The condition evaluation process consumes the boolean value from the stack through pop operations, ensuring proper stack management throughout execution.

The composition process maintains exact computational equivalence between the original Python function and the resulting p-stack machine through careful preservation of all execution paths and semantic relationships. Parameter stack information is integrated into the final machine specification, enabling proper initialization with input values during subsequent neural network compilation stages.

This systematic extraction methodology ensures that mathematical operations implemented in Python can be reliably translated into equivalent p-stack machine representations suitable for neural network compilation while maintaining precise computational semantics throughout the transformation process.