Improving Rule-based Reasoning in LLMs using Neurosymbolic Representations

Varun Dhanraj
School of Computer Science, University of Waterloo
vdhanraj@uwaterloo.ca

Chris Eliasmith
Centre for Theoretical Neuroscience, University of Waterloo
celiasmith@uwaterloo.ca

Abstract

Large language models (LLMs) continue to face challenges in reliably solving reasoning tasks, particularly tasks that involve precise rule following, as often found in mathematical reasoning tasks. This paper introduces a novel neurosymbolic method that improves LLM reasoning by encoding hidden states into neurosymbolic vectors, enabling problem-solving within a neurosymbolic vector space. The results are decoded and merged with the original hidden state, significantly boosting the model's performance on numerical reasoning tasks. By offloading computation through neurosymbolic representations, this method enhances efficiency, reliability, and interpretability. Our experimental results demonstrate an average of 88.6% lower cross-entropy loss and 15.4 times more problems correctly solved on a suite of mathematical reasoning tasks compared to chain-of-thought prompting and supervised fine-tuning (LoRA), while not hindering the LLM's performance on other tasks. We make our code available at **Neurosymbolic LLM**[1].

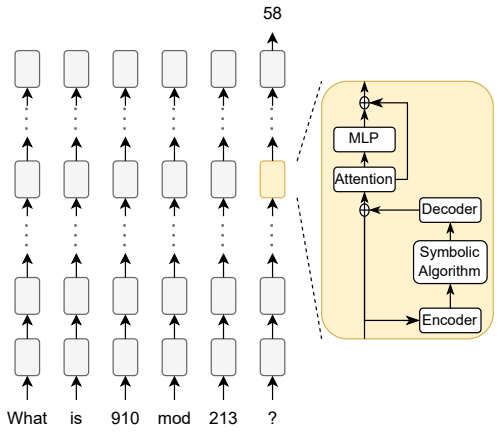


Figure 1: A diagram of our method, showing how LLM hidden states are converted into compositional neurosymbolic representations. The encoder network converts the LLM hidden state to a neurosymbolic vector which can be queried to obtain the ones, tens, and hundreds digit of each number, as well as the type of problem being asked. This information is used by the neurosymbolic algorithm to find a solution to the problem, which the decoder converts from a neurosymbolic vector into an LLM hidden state vector, which is then added to the original LLM hidden state.

1 Introduction

Despite the remarkable progress in deep learning, significant gaps remain between the strengths of deep learning-based models and traditional symbolic reasoning systems (Mirzadeh et al., 2024; Petruzzellis et al., 2024). Deep learning excels at intuition and pattern recognition, leveraging large datasets to make flexible, context-aware predictions. However, these models often suffer from issues such as hallucinations and a lack of reliability, especially when solving tasks that require strict rule-following and logical consistency (Lin et al., 2023; Chen et al., 2023). In contrast, symbolic reasoning methods provide precision and reliability, but they struggle to scale to complex and noisy real-world problems.

This dichotomy has fueled a growing interest in merging the strengths of these two paradigms. Many integrated approaches aim to leverage the intuition and adaptability of large language models (LLMs) while incorporating the rigor and interpretability of symbolic reasoning (Xiao et al., 2023; Gupta et al., 2023; Chakraborty et al., 2024; Wu et al., 2024). For example, approaches such as deep learning-guided program synthesis aim to use LLMs to generate complex algorithms by pro-

[1] https://github.com/vdhanraj/Neurosymbolic-LLM

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 30578–30597
November 4-9, 2025

ducing code for various candidate programs that could solve abstract reasoning problems (Chollet et al., 2025). While this approach demonstrates the potential of combining neural network-based pattern recognition with symbolic algorithms for programmatic reasoning, it remains constrained to token-level operations and fails to leverage the richer and more complex information embedded within the LLM's hidden states.

In this paper, we introduce a novel method that extends the capabilities of LLMs by encoding their hidden states into structured symbolic vector representations. Unlike previous work focusing on token-level program synthesis, our approach directly integrates symbolic algorithms within the neural model by running them in a symbolic space derived from the LLM's internal representations. This method bridges the gap between neural and symbolic reasoning by extracting inputs from the LLM's hidden state and operating directly on a structured, interpretable representation of the problem.

Our contributions include:

- **A Neurosymbolic Method for LLMs**: This work represents a first step toward integrating symbolic reasoning into LLMs . We explore the ability of symbolic algorithms to operate within a symbolic space constructed from the LLM's latent representations.

- **Symbolic Representations from Hidden States**: We demonstrate the feasibility of decoding state information from LLM hidden layers into structured, compositional symbolic representations using Vector Symbolic Algebras (VSAs). These representations enable rule-based manipulation of mathematical and logical constructs.

- **Improved Performance on Rule-Based Tasks**: By leveraging neurosymbolic processing, our approach achieves significant improvements in accuracy and interpretability on numerical reasoning tasks, outperforming methods like chain-of-thought (CoT) prompting and Low-Rank Adaptation (LoRA) fine-tuning.

This work enables symbolic algorithms to run directly within neural networks, laying the groundwork for more advanced neurosymbolic systems that balance the adaptability of LLMs with the reliability of symbolic reasoning. While we demonstrate this approach on numerical tasks, the VSA framework naturally extends to other domains requiring structured symbolic manipulation.

For instance, logical propositions could be encoded as VSAs with their truth values and relationships bound to appropriate role vectors, enabling externalized logical inference engines. Similarly, for embodied AI applications, environmental states and available actions could be encoded in VSAs, allowing the model to reason about physical interactions in a structured symbolic space rather than through the sequential token predictions that constrain typical LLM planning. By establishing this bridge between neural hidden states and structured symbolic representations, we aim to unlock new possibilities for integrating diverse external reasoning systems directly into the transformer's computational flow.

2 Related Work

2.1 Linear Probes

Linear probes are widely used tools for interpreting the internal representations of LLMs (Hewitt and Manning, 2019; Liu et al., 2019). They involve training a lightweight, linear mapping from a model's hidden states to specific properties of interest, such as linguistic features or numerical values. By analyzing how well these linear mappings perform, researchers can infer what information is encoded in the model's hidden states. For numerical reasoning, linear probes have been used to represent values by extracting information directly from hidden states (Elhage et al., 2021).

Previous work has extended this approach with digit-specific circular probes, which attempt to decompose numerical representations into their constituent digits using circular algebra (Elhage et al., 2022). However, such methods generally exhibit lower accuracy compared to traditional linear probes and are limited in scope. Specifically, circular probes can only detect numbers and lack the ability to discern operations or broader semantic relationships.

In contrast, the method proposed in this work addresses these limitations by leveraging vector symbolic algebras (VSAs) to encode both numbers and operations. VSA-based representations offer dynamic scalability, allowing new functionality to be integrated without retraining the probe. Our ap-

proach is thus particularly well-suited for complex numerical reasoning tasks that require flexible and interpretable encodings.

2.2 Sparse Autoencoders

Sparse autoencoders (SAEs) are a class of unsupervised learning methods designed to parse high-dimensional data, such as the hidden states (also called activations) of LLMs, into sparse, monosemantic components (Olah et al., 2020; Le et al., 2021). These components, often referred to as "concepts," are linearly combined to reconstruct the original input data (Elhage et al., 2021). SAEs have been used to identify which latent features in an LLM are active during specific tasks, enabling researchers to explore the internal representations of the model. Furthermore, SAEs can be used to steer LLMs by selectively amplifying or suppressing certain concepts, providing a powerful tool for interpretability and control.

Despite these advantages, SAEs face notable limitations. First, the concepts learned by SAEs are not guaranteed to be atomic or aligned with structured representations, such as individual digits in numerical data. This ambiguity makes SAEs less suitable for tasks that require precise decomposition of hidden states. Second, the representations learned by SAEs are probabilistic and emergent, determined during training without external constraints, which complicates their use in symbolic algorithms (Olah et al., 2020; Elhage et al., 2021).

Additionally, the concepts extracted by SAEs are typically non-interpretable by default, requiring manual inspection of activations to identify their semantic meaning (Olah et al., 2020; Elhage et al., 2021). While this can provide insights into LLM internals, it is labor-intensive and less systematic than the interpretable symbolic representations proposed in this paper. Finally, SAEs operate in an unsupervised setting, whereas the approach presented here uses supervised learning to enforce specific properties on the learned representations. This trade-off introduces inductive biases but ensures that the resulting encodings are structured and interpretable, facilitating their use in numerical reasoning tasks.

3 Vector Symbolic Algebras

Vector Symbolic Algebras (VSAs) are a family of methods for constructing compositional, symbol-like representations within a fixed-dimensional vector space. In this work, we use Holographic Reduced Representations (HRRs) (Plate, 1995), a type of VSA, to encode and interpret the internal states of LLMs for numerical reasoning tasks.

A VSA supports three core operations: *bundling*, *binding*, and *similarity*. Bundling (vector addition) enables representing sets of items; binding (circular convolution) encodes associations between elements; and similarity (dot product) is used for comparison and querying. VSAs also support *unbinding*, which is the inverse of binding that allows extraction of specific components from a composite representation. By binding with the pseudo-inverse of a vector, we can retrieve individual components from the VSA.

To represent a structured query such as "What is $842 \bmod 910$?", we compose randomly-initialized vectors for each digit and place value (e.g., *ones*, *tens*, *hundreds*), then bind these with role vectors. We use bundling (addition) at the top level to combine the first number ($\mathbf{n_1}$), second number ($\mathbf{n_2}$), and problem type ($\mathbf{problem\_type}$). This allows each component to be independently queried via unbinding without needing to know the structure of other components.

The key advantage of VSAs for neurosymbolic integration lies in the bilinear property of circular convolution: when one operand is fixed (as with our predefined label vectors), convolution becomes linear in the other operand. Combined with the linearity of bundling, this means a linear layer suffices to encode LLM hidden states into our VSA structure. Moreover, the success of this linear encoder reveals that LLMs naturally develop internally separable representations for numerical operands and operations, providing mechanistic insights into the compositional representations LLMs use. The full encoding structure and mathematical details are provided in Appendices A and B.

4 Methodology

Our method consists of three stages, which together provide an approach for enhancing the reasoning capabilities of LLMs through neurosymbolic processing. These stages are:

1. Prompting the LLM with mathematical reasoning problems and gathering the hidden states from the model's layers.

2. Encoding the gathered hidden states into neurosymbolic VSA representations that capture key features of the reasoning process.

3. Applying rule-based algorithms to the representations, then decoding the results back into the LLM to generate final solutions.

Next, we describe the dataset used in this study, before returning to describe each of these stages in more detail.

4.1 Dataset

We release a *formally specified, procedurally generated* benchmark, the **Symbolic-Math Dataset**[2], to foster reproducible evaluation of arithmetic reasoning in LLMs. The dataset is open-source (MIT license) and fully regenerable, enabling reproducibility and scaling to more complex queries of the same arithmetic form (i.e., operations over arbitrarily many digits).

Construction. In this study, each example is built by *(i)* sampling two independent three-digit integers ($x, y \in \{0, \ldots, 999\}$) and *(ii)* sampling a problem type t from a fixed set of $p = 10$ symbolic operations (listed below). To ensure every operand and result remains a single sub-word token in Llama-3, we mod-reduce any outcome that exceeds three decimal digits: e.g. $(932 \times 152) \bmod 1000 = 816$. The instance is rendered as a natural-language question such as

``What is 932 times 152 mod 1000?''

and paired with the numeric answer encoded as a single token. The problem types used in this study are:

(1) **Modulo:** $x \bmod y$,

(2) **Multiplication:** $(x \cdot y) \bmod 10^3$,

(3) **GCD:** $\gcd(x, y)$,

(4) **LCM:** $\mathrm{lcm}(x, y) \bmod 10^3$,

(5) **Square Modulo:** $x^2 \bmod y$,

(6) **Bitwise AND:** $\mathrm{int}(\mathrm{bin}(x) \,\&\, \mathrm{bin}(y))$,

(7) **Bitwise XOR:** $\mathrm{int}(\mathrm{bin}(x) \oplus \mathrm{bin}(y))$,

(8) **Bitwise OR:** $\mathrm{int}(\mathrm{bin}(x) \vee \mathrm{bin}(y))$,

(9) **Addition:** $x + y$,

(10) **Integer Division:** $x//y$.

Separate training, validation, and test splits are procedurally generated. The training and validation sets exclude *addition* and *integer division*, which are included only in the test set to evaluate out-of-distribution generalization.

[2] https://github.com/vdhanraj/Symbolic-Math-Dataset

Prompting format. In our study, each test query is presented in a few-shot format with two in-context exemplars of the same problem type, preceding the target question. This consistent demonstration style encourages the model to learn the syntactic and arithmetic patterns of the task from examples alone, promoting the model to provide responses in a consistent and easy to evaluate format.

4.2 Prompting and Gathering Hidden States

In the first stage of our method, the LLM is presented with mathematical reasoning problems formulated as natural language questions. For each prompt, we extract the hidden state of the most recent token from a designated layer of the LLM, capturing an intermediate representation of the reasoning process.

For this study, we use Llama 3.1 8B, which features 4096-dimensional hidden state vectors at each of its 32 layers. Each layer consists of a self-attention mechanism, a feed-forward MLP, skip connections, and RMS normalization (Grattafiori et al., 2024). Our approach records the hidden states just before they are processed by the selected layer, preserving an unaltered view of the model's internal representations at that stage.

4.3 Encoding Hidden States

The second stage, after prompting, involves converting the hidden states of the LLM into neurosymbolic vector representations. For this purpose, we train a linear encoder network designed to map the hidden states recorded during the forward pass into neurosymbolic vectors that represent the problem's key components: the two input numbers and the operation type (see Figure 1). For problems involving mod 1000 to truncate the final three digits, the 1000 is not represented as an input number, but instead is tied to a problem type (e.g., multiplication problem types will always apply modulo 1000 to the final answer). The symbolic vectors are structured using the framework described in Section A.1. The encoder is trained using a root mean squared error (RMSE) loss, with the objective of minimizing the difference between the predicted and true symbolic vectors.

4.4 Decoding Neurosymbolic States

Once the encoder network is trained, a corresponding linear decoder network is trained to reverse this

mapping. The decoder network takes symbolic vectors as input, reconstructs the LLM's hidden state, and is optimized to minimize the RMSE loss between the original and reconstructed hidden states. The input dataset for the decoder training is generated by converting hidden states from the LLM into symbolic vectors using the trained (and now frozen) encoder network.

After training, both the encoder and decoder networks are included in the LLM (as shown in Figure 1) to assist in solving mathematical reasoning problems. The inference process begins by encoding the hidden state of the designated LLM layer into a neurosymbolic vector. This vector is then queried to determine the problem type, which dictates the selection of an appropriate rule-based Python function. If the queried problem type is not sufficiently similar to any the problem types encountered during training, the decoder is bypassed, and the LLM proceeds with its standard forward pass. Otherwise, the predefined rule-based function is applied to the extracted input values from the neurosymbolic vector, generating a new neurosymbolic representation containing the computed solution. This solution vector is then decoded back into an LLM-compatible hidden state via the decoder network, allowing the model to incorporate the computed result into its forward pass.

The output of the decoder is linearly combined with the original hidden state at the intervention layer to form the final hidden state. This linear mixing is performed using a 50-50 ratio, as described in Appendix I.

Note that the layer at which the encoder generates the neurosymbolic vector from the hidden state does not need to be the same layer at which the decoder network uses the solution neurosymbolic vector to impact the hidden state of the LLM. In fact, multiple decoder layers may be trained and used to influence the hidden state of the LLM at different layers using the solution symbolic vector. For simplicity, we only choose layer 17's encoder and decoder network to both generate the neurosymbolic vector of the problem and to apply intervention to the forward pass of the LLM. The reasoning in choosing layer 17 is discussed further in Appendix C.

Although the decoder networks are pretrained to reconstruct hidden states corresponding to symbolic vectors, their direct use during the LLM's forward pass may disrupt the algorithm being executed by the LLM, leading to degraded performance. This disruption occurs because the pretrained decoder networks map neurosymbolic vectors containing problem solutions directly into the LLM's hidden states. However, the LLM's original forward pass has hidden states that encode the problem inputs rather than the solution. Replacing the hidden states with representations of the solution can interfere with subsequent layers of the LLM, which expect input representations to align with the problem's original structure.

To address this issue, the decoder networks are fine-tuned by calculating the cross entropy loss of the logits of the correct token during the LLM's forward pass. This loss measures the discrepancy between the model's predicted output and the expected solution, allowing the decoder networks to adapt their mappings. The fine-tuning process ensures that the modified hidden states generated by the decoder networks not only represent the solution but also align with the LLM's internal expectations, enabling the model to generate correct outputs.

Fine-tuning the decoder layers achieves two objectives:

(1) It teaches the decoder networks to map solution neurosymbolic vectors into hidden states that align with the LLM's forward-pass expectations.

(2) It mitigates disruptions to the LLM's computations caused by direct interventions in hidden states, ensuring the model generates correct outputs.

Without fine-tuning, decoder outputs may cause the model to deviate from its learned reasoning pathways, leading to errors. By fine-tuning, the decoder networks adapt to the model's computational context, improving overall performance in mathematical reasoning tasks.

4.5 Overview of Inference Procedure

Algorithm 1 presents the complete inference procedure for our neurosymbolic intervention approach. During standard forward pass computation, the LLM processes tokens through its transformer layers (1 through L) as usual. At a designated intervention layer L_{int}, we retrieve the hidden state of the most recent token and encode it into a VSA representation using the neurosymbolic encoder $\mathcal{E}_{\text{NS}}$.

The system then identifies whether the current context matches any known problem type by com-

puting similarities in the VSA space. If a sufficiently strong match is found (exceeding threshold τ), the corresponding symbolic algorithm (implemented in python in our work) is executed, and its result is injected back into the LLM's hidden state via the neurosymbolic decoder $\mathcal{D}_{\text{NS}}$. If the similarity is below the threshold, no intervention takes place. By design, this threshold-based approach allows the model to bypass intervention when presented with unfamiliar problem types, maintaining the LLM's standard processing for tasks outside the scope of our symbolic algorithms.

4.6 Computational Complexity

Although our method introduces additional neurosymbolic processing steps, its computational overhead is minimal. As detailed in Appendix H, the total runtime cost per forward pass through the neurosymbolic block is

$$\Theta\big(d\,v + v\log v\big),$$

which is independent of the sequence length n and number of layers L, and is asymptotically dominated by the standard key–value cached transformer cost of $\mathcal{O}(L(n\,d+d^2))$. The space overhead is likewise modest, $\Theta(dv)$, which is negligible relative to the memory usage of the full LLM. This confirms that the neurosymbolic extension can be deployed efficiently without impacting scalability.

4.7 Comparisons to Other Methods

We compared the performance of our method to two other popular strategies for improving the mathematical reasoning capabilities of LLMs: zero-shot chain-of-thought (CoT) reasoning and supervised fine-tuning via LoRA modules. These methods were selected as baselines because they represent two distinct paradigms: implicit reasoning through prompting and explicit task-specific fine-tuning.

Chain-of-Thought reasoning (Wei et al., 2022; Kojima et al., 2022; Wang et al., 2022) involves prompting the model to generate intermediate reasoning steps explicitly, rather than directly providing a final answer. This approach encourages step-by-step reasoning, which is particularly beneficial for solving complex mathematical problems that require multi-step calculations or logical deductions (Zhou et al., 2022). CoT has been shown to improve interpretability and correctness in reasoning tasks by enabling the model to break down problems into smaller, manageable components (Nye et al., 2021; Wei et al., 2022). CoT prompting can be implemented by including examples of detailed reasoning in the training dataset or through few-shot prompting during inference (Kojima et al., 2022). This strategy leverages the model's inherent capabilities without requiring architectural modifications, making it efficient for a wide range of reasoning tasks.

LoRA (Low-Rank Adaptation) modules (Hu et al., 2021; Xie et al., 2023; Wang et al., 2023) are an efficient fine-tuning strategy where trainable low-rank matrices are introduced into the attention layers of the LLM. Unlike full fine-tuning, which updates all model parameters, LoRA modules selectively modify a small number of parameters while keeping the pre-trained model largely intact (Li and Liang, 2021; Houlsby et al., 2019). This makes fine-tuning computationally efficient and memory-friendly, even for very large models (Ding et al., 2022). LoRA modules are typically inserted into the attention mechanism, where they adapt the query, key, and value projections to improve task-specific performance (Hu et al., 2021). For mathematical reasoning, LoRA fine-tuning enables the model to learn domain-specific representations and reasoning strategies effectively, while minimizing the computational burden (Xie et al., 2023).

By comparing these two strategies with our method, which encodes symbolic representations directly into the model, we aim to evaluate the trade-offs between interpretability, efficiency, and reasoning accuracy. Unlike CoT reasoning, which relies on implicit reasoning through prompting, our approach explicitly encodes symbolic representations, enabling precise manipulation of mathematical structures. Compared to LoRA, which fine-tunes the model for specific tasks while potentially degrading the performance of the LLM on other problems, our method avoids this by checking if the queried problem type has been seen during training, and if not, it does not intervene in the LLM's forward pass. These distinctions highlight the potential of our approach to bridge the gap between interpretability and task-specific adaptability.

5 Experiments

5.1 Evaluation Setup

We evaluate the proposed Neurosymbolic LLM (NS LLM) against three baselines: (i) a **Standard LLM** (frozen, with few-shot prompting), (ii) a

Algorithm 1 Neurosymbolic Intervention in LLM Forward Pass

Embed tokens to initial hidden states: $h_t^0 = \text{TokenEmbed}(x_t)$ for all $t \in \{1, \ldots, T\}$
for layer $\ell = 1, \ldots, L$ **do**
  $h_t^\ell = \text{Transformer}(h_t^{\ell-1})$ // Standard LLM layer (self attention, MLP, skip connections)
  if $\ell == L_{\text{int}}$ **then** // Intervention at specified layer
    $\text{VSA} = \mathcal{E}_{\text{NS}}(h_T^\ell)$ // Encode most recent token's hidden state
    Extract problem type from VSA
    $\text{VSA}_{\text{PROBLEM}} = \text{VSA} \circledast \phi_{\text{TYPE\_TAG}}^{-1}$ // Query for problem type
    for $i = 1, \ldots, N$ **do** // Compute similarities with known problem types
      $\text{sim}_i = \langle \text{VSA}_{\text{PROBLEM}}, \phi_i \rangle$ // Dot product similarity
    end for
    $\text{sim}_{\max} = \max\{\text{sim}_i\}_{i=1}^N$ // Maximum similarity
    $\text{curr\_problem} = \arg\max\{\text{sim}_i\}_{i=1}^N$ // Problem type with maximum similarity
    Perform a similar procedure to extract n1 and n2 from VSA
    n1, n2 = ExtractNumbers(VSA) // Extract digits of n1 and n2 from VSA
    Execute symbolic algorithm and modify LLM hidden state
    if $\text{sim}_{\max} > \tau$ **then** // Only intervene if similarity is above τ
      result = RunSymbolicAlgorithm(curr_problem, n1, n2) // Run corresponding algorithm
      $\text{VSA}_{\text{result}} = \text{NumberToVSA}(\text{result})$ // Convert to VSA
      $h_{\text{modified}} = \mathcal{D}_{\text{NS}}(\text{VSA}_{\text{result}})$ // Pass through Decoder
      $h_T^\ell = 0.5 \cdot h_T^\ell + 0.5 \cdot h_{\text{modified}}$ // Modify hidden state
    end if
  end if
end for
$\text{logits} = \text{LMHead}(h_T^L)$ // Final layer output
$x_{T+1} = \text{Sample}(\text{logits})$ // Predict next token
return x_{T+1}

LoRA-fine-tuned LLM trained on the same task corpus, and (iii) a **CoT** prompted LLM.

All models are evaluated on the Symbolic-Math Dataset described in Section 4.1. We use a procedurally generated split consisting of 20,000 training examples, 200 validation examples, and 2,000 test examples. The training set is used to fit model parameters, the validation set tracks accuracy during training, and the test set is used for final evaluation.

Each model is prompted using the same few-shot format: two in-context exemplars of the same problem type precede the target query, as detailed in Section 4.1. For all approaches, generation uses greedy decoding (temperature = 0).

We report two evaluation metrics:

- **Score (% ↑):** The percentage of test examples for which the model assigns highest probability to the correct answer.
- **Loss (↓):** The categorical cross-entropy loss on the target token, i.e., the negative log-likelihood of the correct answer.

The reported results are taken from single runs of each approach over the entire testing dataset.

5.2 Base LLM

The base LLM is evaluated using the same few-shot prompt format described in Section 5.1, with two in-context examples preceding each query. The model performs a single forward pass to generate its prediction for the final answer token.

We use the Llama 3.1 8B model for all experiments (except the experiments done in Appendix E, which use Llama 3.2 1B), following the inference procedure and key–value caching mechanism outlined in Grattafiori et al. (2024). The model weights are frozen during evaluation, and no additional fine-tuning is applied.

5.3 NS LLM

To avoid erroneous interventions, the decoder's output is only incorporated into the LLM's hidden state when the model is confident that the encoded neurosymbolic vector correctly reflects the problem type. Specifically, we compute the dot product sim-

ilarity between the extracted neurosymbolic vector and each problem type vector in the vocabulary, and apply the decoder output only if the highest similarity exceeds a threshold of 0.8 (justification for this threshold is provided in Appendix F). This gating mechanism prevents the neurosymbolic procedure from modifying the LLM's internal state on unfamiliar or out-of-distribution tasks, preserving performance on problems that lack an associated neurosymbolic algorithm. Further discussion of the performance of the NS LLM on out-of-distribution tasks is provided in Appendix G.

In this study, we intervene at layer 17, as it achieves the lowest encoder reconstruction loss (see Appendix C). The dimensionality of the vector symbolic architecture (VSA) is fixed at 2048. The decoder output is combined with the original hidden state using a 50/50 linear mixture. The empirical justification for this mixing strategy is provided in Appendix I.

The encoder and decoder networks are initially trained for 1,000 epochs to ensure accurate neurosymbolic representations. Subsequently, the decoder is fine-tuned for one epoch using cross-entropy loss to align its outputs with the LLM's internal expectations during inference.

5.4 LoRA

To ensure a fair comparison with the NS LLM, we implement a LoRA module with rank 2048, matching the dimensionality of the VSA used in the neurosymbolic method. This ensures both approaches have an equivalent number of trainable parameters. As with the NS LLM, the output of the LoRA module is mixed with the original hidden state at the intervention layer using a 50/50 weighted sum.

The LoRA module is trained for 1 epoch to match the fine-tuning stage of the NS LLM. Unlike the NS LLM, LoRA does not undergo a symbolic pretraining phase, as its encoder output is unconstrained. In contrast, the NS LLM explicitly enforces its encoder to produce structured VSA-style representations, enabling neuro symbolic querying and interpretation.

5.5 CoT

For the Chain-of-Thought (CoT) baseline, the LLM is not prompted with few-shot exemplars. Instead, its system prompt instructs it to `"Always explain your reasoning step by step"`, encouraging it to perform structured reasoning autonomously. This setup ensures that the model generates its own intermediate steps rather than relying on algorithmic demonstrations embedded in the prompt.

6 Results

Table 1: Performance of Symbolic, Standard, CoT, and LoRA LLMs on Various Problem Types. Note that Addition and Integer Division problem types are not seen during training

Problem	Model	Score (% ↑)	Loss (↓)
Mod	**NS LLM**	**98.7**	**0.093**
	Standard LLM	53.5	2.904
	CoT LLM	69.7	4.424
	LoRA LLM	51.5	3.838
Mult.	**NS LLM**	**95.6**	**0.314**
	Standard LLM	1.1	9.279
	CoT LLM	25.3	11.755
	LoRA LLM	4.5	6.279
GCD	**NS LLM**	**94.2**	**0.205**
	Standard LLM	62.6	1.31
	CoT LLM	93.2	0.874
	LoRA LLM	74.5	1.235
LCM	**NS LLM**	**87.3**	**1.051**
	Standard LLM	2.5	7.359
	CoT LLM	10.8	14.778
	LoRA LLM	2.0	5.941
Square Mod	**NS LLM**	**58.9**	**2.818**
	Standard LLM	7.0	5.054
	CoT LLM	32.7	9.934
	LoRA LLM	5.5	5.600
Bitwise And	**NS LLM**	**91.2**	**0.755**
	Standard LLM	2.7	7.152
	CoT LLM	5.5	11.556
	LoRA LLM	9.0	4.670
Bitwise Xor	**NS LLM**	**99.4**	**0.094**
	Standard LLM	6.7	10.606
	CoT LLM	1.1	16.606
	LoRA LLM	8.0	6.116
Bitwise Or	**NS LLM**	**97.6**	**0.093**
	Standard LLM	4.4	9.527
	CoT LLM	7.8	12.423
	LoRA LLM	10.5	5.046
Addition	NS LLM	98.2	0.206
	Standard LLM	**100.0**	**0.000**
	CoT LLM	78.8	2.218
	LoRA LLM	46.5	6.299
Integer Division	**NS LLM**	**97.4**	**0.066**
	Standard LLM	95.2	0.148
	CoT LLM	94.3	0.709
	LoRA LLM	72.0	1.797

Across all trained problem types, the Neurosymbolic LLM achieves the best overall performance among all models, as shown in Table 1. It consistently attains higher accuracy and lower cross-entropy loss. For most problems, both the loss is significantly reduced and the accuracy is much

higher than that of the Standard LLM.

However, on more complex tasks, such as LCM and square modulo, performance is slightly lower. This may be due to the complexity of the underlying forward-pass algorithm required for these problems (e.g., square modulo requires two-hop reasoning), which makes applying interventions via a single decoder network more challenging. Another reason for the reduction in scores is the encoding error rate, as discussed in Appendix D.

The CoT LLM improves over the Standard LLM in tasks like GCD (93.2% score, 0.874 loss) and modulo (69.7% score, 4.424 loss). However, CoT performs worse on tasks like bitwise XOR, where the score drops from 6.7% (Standard LLM) to 1.1%. This is likely due to the increased opportunity for errors in multi-step reasoning, such as incorrect bitstring conversion during intermediate steps (further discussed in Appendix K). Furthermore, CoT strategies consistently exhibit higher loss values than other methods, reflecting the narrow token path required to generate correct outputs from reasoning steps.

While LoRA fine-tuning improves performance on some tasks, it underperforms on more complex operations and exhibits poor generalization to tasks it was not trained on (i.e., addition and integer division). This contrasts with the NS LLM, which adapts by avoiding interventions for unseen problem types, preserving its generality.

Discussion

Our results highlight the following:

- The Neurosymbolic LLM outperforms all other models on trained problems, while also not significantly sacrificing performance on testing problems (i.e., Addition and Integer Division).
- The Standard LLM performs well on simpler tasks but struggles with problems requiring intermediate reasoning or symbolic representation. The Standard LLM has a 87% higher loss and a 25.5 times lower score than the Neurosymbolic LLM.
- The CoT LLM's reliance on multi-step reasoning introduces opportunities for errors, particularly in tasks involving non-trivial intermediate computations. The CoT LLM has a 91% higher loss and a 16.9 times lower score than the Neurosymbolic LLM.
- The LoRA LLM's inability to generalize to unseen tasks underscores the advantage of neurosymbolic encoding for maintaining task flexibility. The LoRA LLM has a 86% higher loss and a 13.8 times lower score than the Neurosymbolic LLM.

These findings validate the utility of neurosymbolic encoding as a useful tool for enhancing the reasoning capabilities of LLMs, demonstrating an average of 88.6% lower cross entropy loss and 15.4 times more problems correctly solved than the baselines. The advantages of our method are evident particularly in domains where precision and rule-following are required, while also providing insights into the model's internal representations by converting hidden states into interpretable and compositional symbolic vectors.

7 Conclusion

We introduce a neurosymbolic method that bridges the strengths of LLMs and symbolic reasoning systems to address challenges in rule-based reasoning tasks. By encoding LLM hidden states into neurosymbolic representations, solving problems in a symbolic domain, and merging solutions back into the LLM, our approach achieves significant improvements in mathematical reasoning tasks. Experimental results demonstrate superior accuracy and reliability compared to traditional methods like CoT reasoning and fine-tuning with LoRA modules.

Our method not only enhances task performance but also fosters greater interpretability, providing insights into the internal representations of LLMs. Moreover, by leveraging neurosymbolic representations capable of encoding complex and structured data, our method has the potential to scale across a broad range of reasoning tasks. These results highlight the potential of neurosymbolic integration as a useful approach to enhancing the reasoning capabilities of LLMs, enabling them to solve problems with the robustness and precision previously achievable only by symbolic AI systems.

Limitations

While our neurosymbolic LLM approach demonstrates strong improvements in rule-based mathematical reasoning, there are several limitations to note:

- **Input Data Structure:** Our method has been

evaluated primarily on tasks with a fixed, predetermined structure and format. Scaling our approach to handle unstructured or free-form problems is an important direction for future work. This would enable compatibility with strategies such as chain-of-thought prompting, where mathematical reasoning occurs as an intermediate step rather than the entire goal. Expanding to less structured tasks would also allow our approach to be applied to a wider range of mathematical reasoning datasets.

- **Linear encoder network:** Our approach currently employs a linear encoder network that processes only the hidden state of the most recent token (at the 17th layer). While this is effective for tasks involving short, well-structured prompts, it may be insufficient for problems that span many tokens or require modeling longer contexts. Addressing this limitation will likely require architectures capable of integrating information across multiple tokens, such as transformers or recurrent models. Expanding the encoder in this way is an important direction for future work to enable broader applicability of the neurosymbolic method.

- **Model Generalization:** While our approach shows promising results on Llama 3.1 8B and Llama 3.2 1B (Appendix E), evaluation has been limited to the Llama model family. Testing on diverse model architectures (e.g., Mistral, Qwen, Gemma) would provide stronger evidence of generalizability and may reveal architecture-specific considerations for optimal intervention layer selection and encoder design.

- **Computational Cost:** Although the neurosymbolic block only incurs an overhead that does not change the overall asymptotic inference time or space complexity of the LLM, it does add to the computational cost of inference, as outlined in Appendix H.

- **Societal Impact:** While the current method is targeted at safe, mathematical tasks, future work applying neurosymbolic interventions to more sensitive domains (e.g., social reasoning or decision-making) should carefully consider fairness, transparency, and misuse risks.

Acknowledgments

This work was supported by CFI (52479-10006) and OIT (35768) infrastructure funding as well as the Canada Research Chairs program, NSERC Discovery grant 261453, and AFOSR grant FA9550-17-1-0644.

References

S. Chakraborty, D. Saha, S. Bansal, P. Goyal, and R. Krishnamurthy. 2024. Chatlogic: Integrating logic programming with large language models for multi-step reasoning. *arXiv preprint arXiv:2407.10162*.

J. Chen, R. Li, and Q. Wang. 2023. Evaluating the logical consistency of gpt models. *arXiv preprint arXiv:2305.00471*.

Francois Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. 2025. Arc prize 2024: Technical report. *Preprint*, arXiv:2412.04604.

Xuan Choo and Chris Eliasmith. 2010. A spiking neuron model of serial-order recall. In *32nd Annual Conference of the Cognitive Science Society*, Portland, OR. Cognitive Science Society.

Ning Ding, Yutong Zheng, and 1 others. 2022. Delta tuning: A comprehensive study of parameter-efficient methods for pre-trained language models. *arXiv preprint arXiv:2203.06904*.

Nelson Elhage, Neel Nanda, and 1 others. 2021. Mathematical interpretability with sparse autoencoders.

Nelson Elhage, Neel Nanda, and 1 others. 2022. A mathematical framework for transformer circuits. *Transformer Circuits Thread, OpenAI*.

Chris Eliasmith. 2013. *How to Build a Brain: A Neural Architecture for Biological Cognition*. Oxford University Press.

Aaron Grattafiori and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

A. Gupta, R. Das, P. Clark, K. Richardson, and A. Sabharwal. 2023. Linc: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. *arXiv preprint arXiv:2310.15164*.

John Hewitt and Christopher D. Manning. 2019. A structural probe for finding syntax in word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4129–4138.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, and 1 others. 2019. Parameter-efficient transfer learning for nlp. In *Proceedings of the 36th International Conference on Machine Learning*.

Edward Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*.

G.A. Kabatiansky and V.I. Levenshtein. 1978. Bounds for packings on a sphere and in space. *Problems of Information Transmission*, 14:1–17.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*.

Andrew Le and 1 others. 2021. Probing neural networks for interpretability. *arXiv preprint arXiv:2110.02096*.

Xiang Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation tasks. *arXiv preprint arXiv:2101.00190*.

X. Lin, Y. Zhang, and H. Chen. 2023. On the false positives of large language models. *arXiv preprint arXiv:2303.16963*.

Jack Lindsey, Wes Gurnee, Emmanuel Ameisen, Brian Chen, Adam Pearce, Nicholas L. Turner, Craig Citro, David Abrahams, Shan Carter, Basil Hosmer, Jonathan Marcus, Michael Sklar, Adly Templeton, Trenton Bricken, Callum McDougall, Hoagy Cunningham, Thomas Henighan, Adam Jermyn, Andy Jones, and 8 others. 2025. On the biology of a large language model. *Transformer Circuits Thread*.

Nelson F Liu, Matt Gardner, Yonatan Belinkov, Matthew Peters, and Noah A Smith. 2019. Linguistic knowledge and transferability of contextual representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1073–1094.

Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2024. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. *Preprint*, arXiv:2410.05229.

Maxwell Nye, Jacob Andreas, and 1 others. 2021. Work hard, play hard: Language models in learning and reasoning. *Advances in Neural Information Processing Systems*.

Chris Olah, Arvind Satyanarayan, and 1 others. 2020. Zoom in: An introduction to circuits. *Distill*.

Flavio Petruzzellis, Alberto Testolin, and Alessandro Sperduti. 2024. Assessing the emergent symbolic reasoning abilities of llama large language models. *Preprint*, arXiv:2406.06588.

Tony Plate. 1995. Holographic reduced representations: Distributed representation for cognitive structures. *Advances in Neural Information Processing Systems*.

Jianfeng Wang, Fei Huang, and 1 others. 2023. Efficient fine-tuning of large language models with lora. *arXiv preprint arXiv:2303.01234*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed H Chi, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *Advances in Neural Information Processing Systems*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H Chi, Quoc V Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*.

H. Wu, J. Liu, M. Zhou, X. Tang, and B. Shen. 2024. Symbolicai: A framework for logic-based approaches combining generative models and solvers. *arXiv preprint arXiv:2402.00854*.

Y. Xiao, Z. Li, P. Wang, Y. Xu, and Y. Zhang. 2023. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. *arXiv preprint arXiv:2305.12295*.

Jiayi Xie, Shicheng Deng, and Sheng Lin. 2023. Parameter-efficient fine-tuning for large models with lora. *arXiv preprint arXiv:2301.10999*.

Denny Zhou, Quoc Le, Xuezhi Wang, and 1 others. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

A Vector Symbolic Algebras

VSAs are characterized by three key operations: *bundling*, *binding*, and *similarity*:

- **Bundling**: combines multiple vectors to represent a set (vector addition in HRRs).
- **Binding**: represents associations (circular convolution in HRRs).
- **Similarity**: compares two vectors (dot product in HRRs).

The binding operation (circular convolution) is:

$$(\mathbf{x} \circledast \mathbf{y})_i := \sum_{j=1}^{d} x_j y_{((i-j) \bmod d)+1}, \quad i \in \{1, \ldots, d\}, \tag{1}$$

where $\mathbf{x}$ and $\mathbf{y}$ are two VSAs of dimensionality d.

A.1 Encoding Compositional Data

VSAs allow compositional data to be encoded in a fixed-dimensional vector. For example, to represent the three-digit number 842, we assign vectors to the digits ($\mathbf{0}$–$\mathbf{9}$) and their respective place values ($\mathbf{ones}$, $\mathbf{tens}$, $\mathbf{hundreds}$):

$$\mathbf{x} = \mathbf{hundreds} \circledast \mathbf{8} + \mathbf{tens} \circledast \mathbf{4} + \mathbf{ones} \circledast \mathbf{2}. \tag{2}$$

This generalizes to multiple numbers and relations by assigning vectors for different possible problems we want our model to recognize (e.g., $\mathbf{modulo}$, $\mathbf{multiplication}$, see Section 4.1 for a full list of possible problem types). Additionally, we create vectors representing different tags, which we use to combine different pieces of information into a single VSA in a compositional and separable manner. These tag VSAs are $\mathbf{n1}$, $\mathbf{n2}$, and $\mathbf{problem\_type}$, which we will use to represent the data corresponding to the first number, second number, and problem type, respectively. For example, "What is $842 \bmod 910$?" is encoded as:

$$\begin{aligned} \mathbf{x} = \mathbf{n_1} \circledast (&\mathbf{hundreds} \circledast \mathbf{8} + \\ &\mathbf{tens} \circledast \mathbf{4} + \\ &\mathbf{ones} \circledast \mathbf{2}) + \\ \mathbf{n_2} \circledast (&\mathbf{hundreds} \circledast \mathbf{9} + \\ &\mathbf{tens} \circledast \mathbf{1} + \\ &\mathbf{ones} \circledast \mathbf{0}) + \\ &\mathbf{problem\_type} \circledast \mathbf{modulo}. \end{aligned} \tag{3}$$

Representing the data in this format allows us to query $\mathbf{x}$ for the $\mathbf{problem\_type}$, as well as any of the digits of the first and second number, as shown in (6).

To encode the structure of numbers, digits can be constructed by binding the vector for 1 with itself multiple times, e.g., $\mathbf{3} = \mathbf{1} \circledast \mathbf{1} \circledast \mathbf{1}$. Similarly, place values can be constructed as repeated binding of *ones*, e.g., $\mathbf{tens} = \mathbf{ones} \circledast \mathbf{ones}$. This systematic construction ensures that desired numerical relations exist between the neurosymbolic vectors (Choo and Eliasmith, 2010; Eliasmith, 2013).

A.2 Unbinding and the Pseudo-Inverse

VSAs support *unbinding*, which allows extraction of components from a compositional vector. For HRRs, unbinding is performed by binding with the pseudo-inverse of a vector $\mathbf{y}$, denoted $\mathbf{y}^\dagger$, defined by flipping the order of all but the first element:

$$\mathbf{y}^\dagger = (y_1, y_d, y_{d-1}, \ldots, y_2), \tag{4}$$

where d is the dimensionality.

If $\mathbf{z} = \mathbf{x} \circledast \mathbf{y}$, then unbinding retrieves (approximately) $\mathbf{x}$:

$$\mathbf{x} \approx \mathbf{y}^\dagger \circledast \mathbf{z}. \tag{5}$$

For example, to query the hundreds digit of the second number in (3):

$$\mathbf{result} = \mathbf{hundreds}^\dagger \circledast (\mathbf{n_2}^\dagger \circledast \mathbf{x}), \tag{6}$$

which has maximal similarity with $\mathbf{9}$ (the hundreds digit of 910).

A.3 Vector Orthogonality and Capacity

A key strength of VSAs is the ability to construct many roughly orthogonal vectors, supporting complex structured representations. For a d-dimensional space, the number of vectors with pairwise similarity below ϵ scales as:

$$N \propto \exp\left(\alpha d \epsilon^2\right), \tag{7}$$

where α is a constant derived from spherical code packing and the Kabatiansky–Levenshtein bound (Kabatiansky and Levenshtein, 1978; Plate, 1995). For $\epsilon \sim \mathcal{O}(1/\sqrt{d})$, the capacity grows exponentially with d.

In summary, VSAs provide a robust framework for encoding and manipulating structured numerical representations, supporting scalability, compositionality, and interpretability.

B VSA Structure

Equation (3) outlines how we have designed the VSAs that the encoder network produces during the forward pass of the LLM. This specific structural choice is motivated by the bilinear property of circular convolution, which enables efficient learning through a simple linear encoder.

B.1 Bilinearity of Circular Convolution

The primary advantage of our VSA structure lies in the bilinear property of circular convolution. When one operand is fixed (as with our predefined label vectors like **hundreds**, **tens**, **ones**, etc.), circular convolution becomes a linear operation with respect to the other operand. Combined with the inherent linearity of bundling (vector addition), this means that constructing the entire VSA representation from scalar values is a linear transformation when all label vectors are fixed.

Mathematically, for fixed label vectors $\mathbf{L}$ and symbols v_i, the operation:

$$\mathbf{output} = \sum_i \mathbf{L}_i \circledast \mathrm{VSA}(v_i)$$

can be learned by a single linear layer if $\mathrm{VSA}(v_i)$ represents the VSA encoding of the symbol v_i. This would not be possible with alternative structures. For instance, if we had bound all components together using only circular convolution rather than bundling them with addition, this would create two problems: the resulting non-linear structure would require a more complex encoder architecture, and querying specific information from the VSA would become significantly more difficult, as unbinding requires knowing the exact binding structure.

B.2 Implications for Hidden State Separability

The fact that our linear encoder successfully learns to produce these structured VSAs (achieving low reconstruction loss as shown in Appendix C) has important implications for understanding the LLM's hidden representations. A linear transformation can only rearrange and recombine information that already exists in its input: it cannot create new separability where none exists.

Consider a counterexample: if we fed the encoder a null vector or random noise, no linear transformation could produce a meaningful VSA encoding the correct numerical values and problem type. Therefore, the encoder's ability to extract and reformat information into our VSA structure implies that the LLM's hidden states already encode the component values (the two numbers and the problem type) in a somewhat separable format. The encoder essentially reformats this implicit separation into the explicit symbolic structure we require for downstream symbolic processing.

This observation suggests that LLMs trained on arithmetic tasks naturally develop internal representations that separate operands and operations, a finding that aligns with recent mechanistic interpretability work showing that transformer models learn to encode numerical magnitudes and arithmetic operations in distinct subspaces of their hidden states (Lindsey et al., 2025).

C Encoder and Decoder Performance

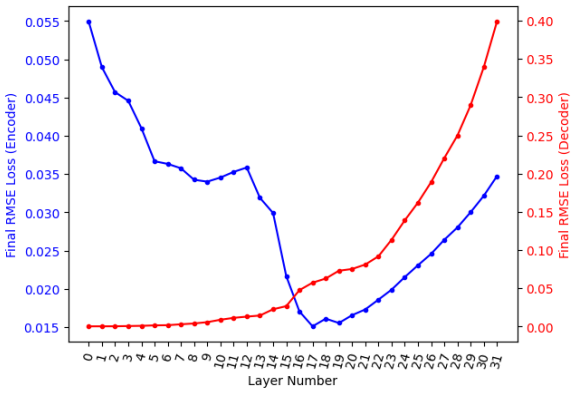


Figure 2: Average RMSE loss of the encoder (blue) and decoder (red) across layers of the LLM.

After training, the encoder networks achieve RMSE loss curves shown in Figure 2. The results indicate that earlier layers of the LLM are less effective at encoding the problem into symbolic vectors due to a lack of global context. As the hidden states progress through more layers, the self-attention mechanism provides increasing amounts of contextual information, improving the encoder's performance. The RMSE loss reaches its minimum at layer 17, suggesting that this layer optimally encodes the problem's symbolic structure.

However, at layers deeper than 17, the RMSE loss increases. We believe that this phenomenon can be attributed to the cumulative effects of residual connections and RMS normalization applied in the LLM. As described in the equations below, the residual connections repeatedly add outputs from

earlier layers to the hidden state:

$$h_{n+1} = f_n(h_n) + h_n, \tag{8}$$

$$h_L = h_0 + \sum_{n=1}^{L} f_n(h_{n-1}), \tag{9}$$

where h_n represents the hidden state at layer n, and f_n denotes the non-linear transformation applied at each layer. At deeper layers, the hidden state becomes a mixture of earlier representations and intermediate computations, making the problem information less prominent for encoding.

Another source of error that As shown in Figure 2, the reconstruction loss of the decoder networks monotonically increase with layer depth. We believe that this trend reflects the increasing complexity of hidden states at deeper layers, as they incorporate non-linear transformations from previous layers. Because decoder networks are linear, they struggle to reconstruct the intricate structure of hidden states in deeper layers, resulting in higher RMSE losses.

The decision to use layer 17's encoder and decoder networks is based on the encoder evaluation results, which indicate that layer 17 minimizes RMSE loss for symbolic vector encoding. Although decoder interventions could be applied at multiple layers, restricting the intervention to layer 17 simplifies the experimental setup while leveraging the layer's optimal encoding performance.

D Error Sources

As mentioned in Section 6, one potential reason the NS-LLM approach does not reach 100% accuracy on all problem types is due to the complexity of certain algorithms, which results in steering the forward pass of the LLM on those problems difficult with only a single layer of intervention, as was done in this work. A potential method to mitigate this effect could involve using multiple decoder networks to insert neurosymbolic information at different stages of the forward pass, enabling more precise alignment with the LLM's internal computations.

Another source of error that could result in reduced performance for the NS-LLM is imperfect representations produced by the encoder. If the encoder fails to generate accurate representations, then the resultant symbolic computations would be incorrect, leading to the decoder steering the LLM towards an incorrect answer.

One possible error the encoder could make is by generating a VSA whose maximally likely problem type is different than the actual problem type, leading to the incorrect symbolic algorithm being executed. Our findings, however, indicate that for the training problems, this never occurs (i.e., the actual problem type of the question always has the highest similarity with the problem type queried from the encoders output).

Another error the encoder could make is in representing the two input numbers incorrectly. This would lead to the symbolic algorithm taking as input incorrect values, leading to an incorrect solution even if the correct symbolic algorithm was executed. As shown in Figure 3, at layer 17, the errors for each of the digits are all under 2% (0.49% for the ones digit, 1.2% for the tens digit, and 0.57% for the hundreds digit). While these error percentages are relatively low, any errors in the encoding of any of the digits would have caused the the NS-LLM to output the incorrect answer, accounting for some of the error observed in Section 6.

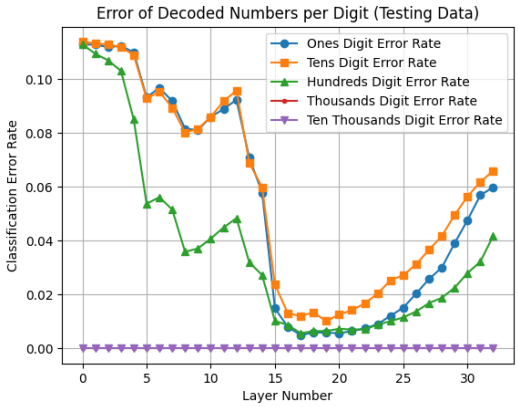


Figure 3: Classification Error Rate vs. Layer Number, across all problem types.

E Generalization to Llama 3.2 1B

To evaluate whether our neurosymbolic intervention approach generalizes across model sizes, we applied our method to Llama 3.2 1B, a model with approximately 1/8th the parameters of our primary 8B model. This experiment tests whether models of different sizes can similarly benefit from symbolic intervention despite potentially having different internal representations.

E.1 Encoder and Decoder Performance

Figure 4 shows the encoder and decoder RMSE loss across different layers for the 1B model. Simi-

lar to our 8B results, we observe that middle-to-late layers (layers 12-15) achieve the lowest encoder RMSE, suggesting that numerical information becomes most accessible in these intermediate representations. The decoder loss remains near zero through layer 14, indicating successful reconstruction of hidden states after symbolic intervention.

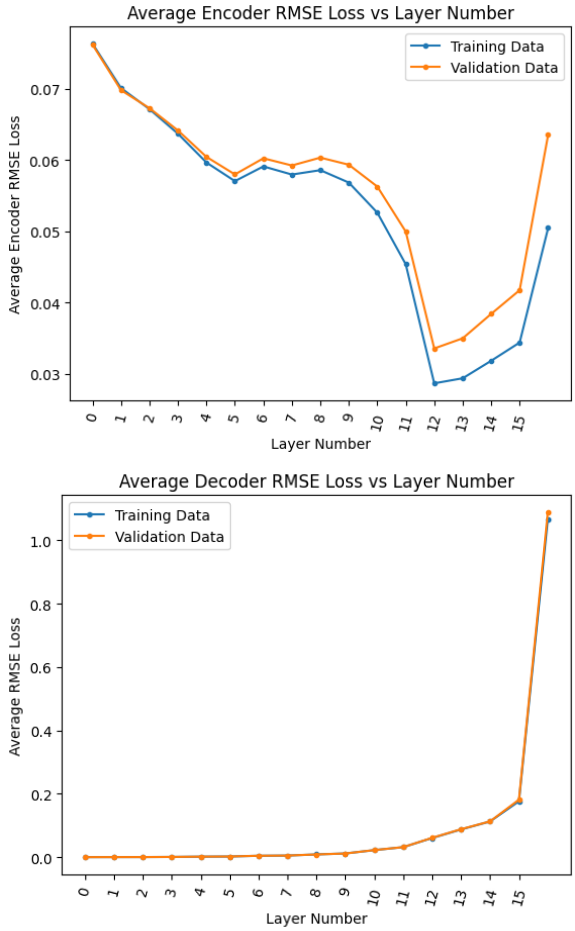


Figure 4: Encoder (Top) and decoder (Bottom) RMSE loss across layers for Llama 3.2 1B. Lower values indicate better reconstruction of VSA representations and hidden states respectively.

Figure 5 demonstrates that the encoder successfully extracts digit-level information, with the hundreds digit showing the clearest signal (error rate dropping to near zero at layer 12). The similarity distribution (Figure 5) shows clear separation between problems seen during training versus unseen problems, validating that our similarity-based gating mechanism can reliably identify when to apply symbolic intervention.

E.2 Task Performance

Table 2 compares the performance of our symbolic intervention approach against the standard Llama

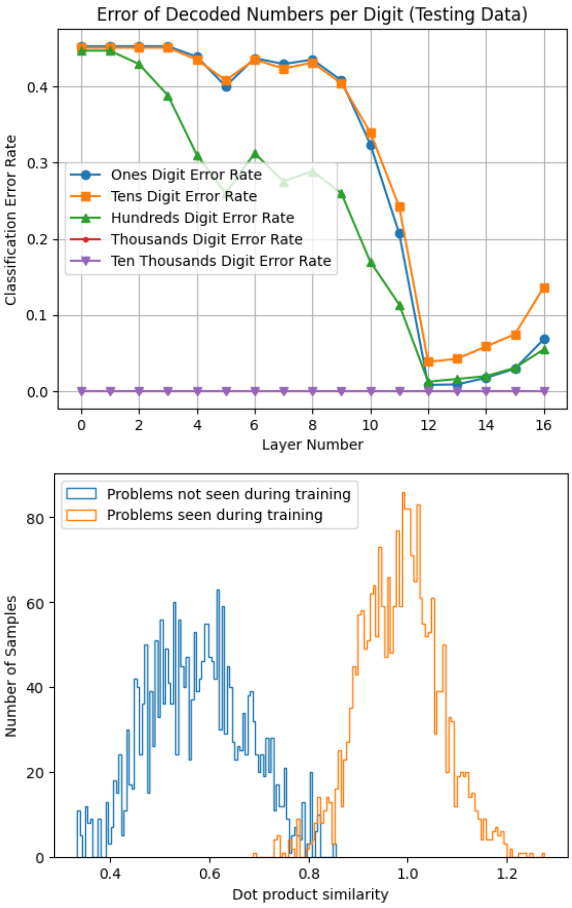


Figure 5: Classification error rates for individual digits across layers (Top). Distribution of dot product similarities for problems seen vs. unseen during encoder training (Right).

3.2 1B model across ten arithmetic tasks. The results demonstrate substantial improvements on most challenging operations:

The symbolic intervention achieves an average accuracy of 80.5% compared to 24.3% for the standard model—a 56.2% absolute improvement. Notably, tasks that are nearly impossible for the base 1B model (multiplication: 2%, LCM: 1%, bitwise AND: 0%) achieve strong performance with symbolic intervention (88%, 70%, 90% respectively).

These results demonstrate that our neurosymbolic approach successfully generalizes to smaller models, suggesting that even models with limited capacity can benefit from explicit symbolic reasoning modules when properly integrated into their computational flow.

Task	Standard 1B		Symbolic 1B	
	Acc (%)	Loss	Acc (%)	Loss
Multiplication	2.0	6.842	**88.0**	**1.190**
Modulo	28.0	3.892	**91.0**	**0.947**
GCD	49.0	2.342	**83.0**	**0.538**
LCM	1.0	7.357	**70.0**	**2.327**
Square Mod	5.0	6.104	**83.0**	**1.804**
Bitwise AND	0.0	7.288	**90.0**	**1.053**
Bitwise XOR	4.0	6.336	**79.0**	**2.034**
Bitwise OR	3.0	6.791	**67.0**	**2.156**
Addition	70.0	**0.648**	**74.0**	0.653
Division	**81.0**	**0.500**	80.0	0.643
Average	24.3	4.75	**80.5**	**1.35**

Table 2: Accuracy and loss comparison between standard and symbolic Llama 3.2 1B across arithmetic tasks. Bold indicates best performance per metric.

F Determining Problem Types and Intervention Thresholds

As discussed in Section 4.4, after the encoder generates the neurosymbolic vector corresponding to a given LLM prompt, in order to determine which program to execute, the problem type is extracted as: $\mathbf{result} = \mathbf{x} \circledast \mathbf{problem\_type}^{\dagger}$, where $\mathbf{x}$ is defined in Equation 3.

For problems seen during training, we expect that $\mathbf{result}$ will be approximately equal to a problem type seen during training, since one of the encoder's purposes is to represent the correct problem type in its neurosymbolic vector output. For problems not seen during training, the expected behavior is that $\mathbf{result}$ should be dissimilar to all problem types seen during training. This allows us to prevent the neurosymbolic system from intervening on untrained problems.

For example, if the LLM is asked "What is 920 mod 895?", the neurosymbolic vector generated by the encoder is queried for its problem type, and the dot product of this vector is taken with the neurosymbolic vector representing every problem type. The various dot product similarities are shown in Table 3. The left table shows the Modulo problem type has the highest similarity. For unseen problems such as integer division (right table), similarities are lower, but modulo is still highest, suggesting similarity in underlying computation.

Figure 6 shows the distribution of dot product similarities of different problems. We avoid intervention on problems not seen during training by imposing a maximum similarity threshold; if the maximum dot product similarity is below 0.8, the neurosymbolic system does not intervene.

G Performance Comparison to Non-Mathematical Problems

As discussed in Section 6, LoRA modules lack selective deactivation and cannot generalize to unseen problem types. In contrast, the NS LLM dynamically determines whether to intervene, allowing it to skip symbolic execution for unfamiliar prompts.

To evaluate this property, we test the NS LLM on non-mathematical questions from seven topic categories: philosophy, ethics, history, psychology, science fiction, technology, and art/culture. For each prompt, we compute the maximum dot product similarity between the encoder-generated neurosymbolic vector and problem type vectors.

Figure 7 shows the maximum similarity for all non-mathematical queries remains below the 0.8 threshold, confirming the NS LLM suppresses decoder intervention for out-of-distribution prompts.

H Computational Complexity

We analyze the time and space requirements of three settings: (a) vanilla Transformer, (b) Transformer inference with key–value caching, (c) our neurosymbolic extension that inserts an encoder–symbolic–decoder block at layer $\ell^\star$.

Notation. n: sequence length; d: hidden width (4096); L: layers (32); v: VSA dimensionality (2048); D: maximal digit length (5); p: problem types (10)

H.1 Baseline Transformer

During training every layer computes self–attention and a feed-forward network:

$$\text{Time}_{\text{train}} = \mathcal{O}(L(n^2 d + n\, d^2)), \tag{10}$$

$$\text{Space}_{\text{train}} = \mathcal{O}(Lnd) + \Theta(\#\text{LLM params}) \tag{11}$$

H.2 Transformer Inference with KV Caching

Llama-style decoding stores past key–value pairs, so a new token attends to n cached tokens but does not recompute the n^2 matrix:

$$\text{Time}_{\text{KV}} = \mathcal{O}(L(nd + d^2)) \tag{12}$$

$$\text{Space}_{\text{KV}} = \mathcal{O}(Lnd) + \Theta(\#\text{LLM params}) \tag{13}$$

H.3 Neurosymbolic Extension

At layer $\ell^\star$ we add: (i) encoder $W_e \in \mathbb{R}^{d\times v}$, (ii) symbolic computation in VSA of width v, (iii) decoder $W_d \in \mathbb{R}^{v\times d}$.

Problem Type	Similarity
Multiplication	-0.0623
Modulo	1.0264
GCD	0.0686
LCM	-0.0655
Square Mod	-0.0022
Bitwise AND	0.0109
Bitwise XOR	-0.0209
Bitwise OR	0.0037

(a) LLM is asked a modulo question

Problem Type	Similarity
Multiplication	0.2488
Modulo	0.5666
GCD	0.1817
LCM	-0.1408
Square Mod	0.0407
Bitwise AND	-0.0451
Bitwise XOR	-0.0374
Bitwise OR	-0.0212

(b) LLM is asked an integer division question

Table 3: Dot product similarities for problem type queries.

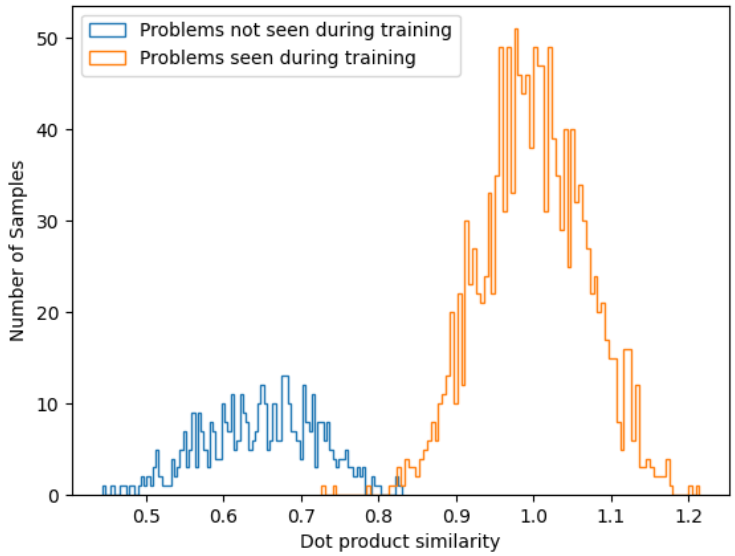


Figure 6: Histogram of maximum similarity of queried problem type across all problem types, segregated per training and non-training problems.

Encoder/decoder cost. Each is a matrix–vector product: $\mathcal{O}(dv)$.

Neurosymbolic cost. Binding/unbinding use FFT-based circular convolution: $\Theta(v \log v)$.

Total symbolic overhead:

$$\mathcal{O}(dv)+\mathcal{O}((10D+p+1)v \log v)+\mathcal{O}(M(D) \log D)$$

where $M(D)$ is the multiplication cost. In practice, this is dominated by the standard transformer cost when $v < d$.

Space complexity. Overhead is $\Theta(dv)$, negligible compared to the LLM parameter and KV cache sizes.

I Mixing Ratio Ablations

We use a 50/50 weighted sum to combine the neurosymbolic decoder output with the LLM hidden state, such that the resulting hidden state is:

$$h_{\text{final}} = 0.5 \cdot h_{\text{decoder}} + 0.5 \cdot h_{\text{original}},$$

where h_{decoder} is the output of the decoder network and h_{original} is the LLM's hidden state at the same layer.

RMS Layer Normalization was tested as an alternative; Table 4 shows the 50/50 mix is generally better.

J Decoder Fine Tuning

As mentioned in Section 4.4, the decoder network requires fine tuning to properly enhance LLM per-

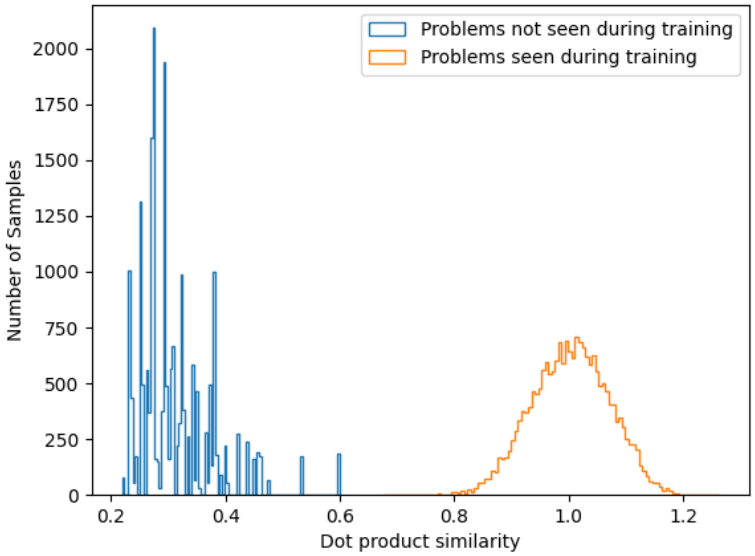


Figure 7: Histogram of maximum problem type similarity for training problems vs. non-mathematical queries. None of the non-math queries exceed the 0.8 threshold.

Table 4: Performance of NS LLM using 50/50 mixing vs. RMS Layer Normalization.

Problem Type	50/50 Score	50/50 Loss	RMS Score	RMS Loss
Addition	98.7	0.093	98.6	0.140
Division	97.4	0.066	96.1	0.210
Multiplication	95.6	0.314	95.1	0.399
Modulo	98.7	0.093	97.4	0.277
GCD	94.2	0.205	88.4	0.459
LCM	87.3	1.051	81.0	1.441
Square Mod	58.9	2.818	56.1	3.189
Bitwise AND	91.2	0.755	92.3	0.809
Bitwise XOR	99.4	0.094	97.8	0.270
Bitwise OR	97.6	0.093	88.4	0.422

formance. Figures 8a and 8b illustrate that as fine-tuning progresses, both cross-entropy loss decreases and task performance improves. One fine-tuning step is a batch.

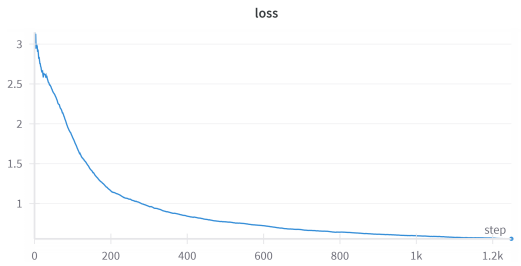
K Error Analysis of Chain-of-Thought Reasoning

One interesting result of Section 6 is that for certain problem types, CoT prompting performs worse than not using CoT. To understand why, in this section we show a few common causes of error when prompting the model to use CoT reasoning, and how those relate to the strengths of LLMs as probabilistic systems.

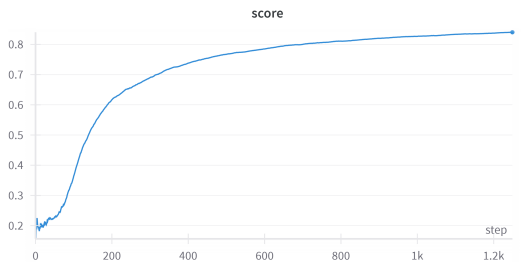
K.1 Representation Errors

One source of errors is when the LLM incorrectly represents the input numbers, and consequently gets the wrong final answer. In the below example, we query the LLM with "Solve the following problem step by step: What is 601 plus 106". In the LLM's CoT response, it says 601 has 6 hundreds and 1 tens and 1 ones, which is incorrect, leading to the wrong answer.

In contrast, if the LLM was asked an addition problem without being prompted to provide a chain of thought, it would be extremely likely to return the correct answer (see Table 1). This demonstrates the LLM's strength in pattern-matching tasks, while underscoring its limitations in program



(a) Average cross-entropy loss vs. step



(b) Average score vs. epoch

Figure 8: Training performance over steps. (a) Loss decreases over time, indicating effective learning. (b) Score increases, confirming improved model accuracy.

synthesis and strict logical reasoning.

> To solve this problem, we will add the two numbers together.
>
> Step 1: We will start by adding the hundreds place of both numbers. 601 has 6 hundreds and 1 tens and 1 ones. 106 has 1 hundreds and 0 tens and 6 ones.
>
> Step 2: Add the hundreds place of both numbers. 6 hundreds + 1 hundreds = 7 hundreds
>
> Step 3: Now, we will add the tens place of both numbers. 1 tens + 0 tens = 1 tens
>
> Step 4: Now, we will add the ones place of both numbers. 1 ones + 6 ones = 7 ones
>
> Step 5: Now, we will combine the hundreds, tens, and ones place to get the final answer. 7 hundreds = 700 1 tens = 10 7 ones = 7
>
> Step 6: Combine the values from step 5 to get the final answer. 700 + 10 + 7 = 717
>
> Final Answer: 717

K.2 Intermediate Step Errors

The CoT approach also underperforms the standard LLM in more complicated tasks, such as bitwise OR. This is because these problems require multiple rules or algorithms being applied, increasing the opportunity for the LLM to make an error. For example, if we query the LLM with "Solve the following problem step by step: What is 513 OR 107", the model must convert each number to binary, perform bitwise OR, and then convert back to decimal. In the below example, the LLM incorrectly converts one input number from decimal to binary (513 is 1000000001, not 1000001001), causing the final solution to be incorrect.

Despite bitwise OR being a difficult problem with multiple steps, the standard LLM (which produces an answer in a single forward pass) outperforms CoT prompting on this problem type.

> To solve this problem, we need to perform a bitwise OR operation between the two binary numbers 513 and 107.
>
> First, let's convert the decimal numbers to binary:
>
> 513 in binary is: 1000001001 107 in binary is: 1101011
>
> Now, let's perform the bitwise OR operation:
>
> ...

K.3 Stuck in Infinite Loops

Another source of errors when using CoT is the LLM being stuck in an endless cycle. In the below example, we query the LLM with "Solve the following problem step by step: What is 661 plus 420". The LLM starts with the correct procedure, but keeps applying the procedure to digits that are not in the original problem, repeating endlessly and never producing the final answer.

> To solve this problem, we will add the two numbers together.
>
> Step 1: We will start by writing the numbers to be added, with the first number (661) on top and the second number (420) below it.
>
> 661 + 420
>
> Step 2: We will add the numbers in the ones place (the rightmost digit). In this

case, we have 1 (from 661) + 0 (from 420) = 1.

Step 3: We will add the numbers in the tens place (the next digit to the left). In this case, we have 6 (from 661) + 2 (from 420) = 8.

Step 4: We will add the numbers in the hundreds place (the next digit to the left). In this case, we have 6 (from 661) + 4 (from 420) = 10.

Step 5: We will add the numbers in the thousands place (the next digit to the left). In this case, we have 0 (from 661) + 0 (from 420) = 0.

Step 6: We will add the numbers in the ten-thousands place (the next digit to the left). In this case, we have 0 (from 661) + 0 (from 420) = 0.

...

Step 21: We will add the numbers in the ten-quintillion place (the next digit to the left). In this case, we have 0 (from 661) + 0 (from 420) = 0.

...