Integrating Symbolic Reasoning into Large Language Models

by

Varun Dhanraj

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

The following chapters contain material based on papers that I authored as the first author:

- Chapters 4, 5, 6, and 7 are adapted from: Varun Dhanraj and Chris Eliasmith. **Improving Rule-based Reasoning in LLMs using Neurosymbolic Representations.** In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP 2025)*, China, November 2025. Association for Computational Linguistics. I was responsible for the conception, methodology, experiments, writing, and revisions of this work. Co-author Chris Eliasmith provided feedback, editorial suggestions, and supervision.

- Chapter 8 is adapted from: Varun Dhanraj and Chris Eliasmith. **Efficient and Robust Neurosymbolic LLM Reasoning using Transformer-Based Encoders.** *Preprint,* 2025. I was responsible for the conception, design, experiments, and writing. Co-author Chris Eliasmith provided feedback, editorial suggestions, and supervision.

All other chapters are original and were written solely by me for this thesis.

Abstract

Large language models (LLMs) face fundamental challenges in symbolic reasoning, struggling with tasks requiring precise rule-following, logical consistency, and manipulation of structured representations. This thesis introduces a comprehensive neurosymbolic framework that addresses these limitations by integrating Vector Symbolic Algebras (VSAs) directly into the computational flow of transformer-based language models. Our core method encodes LLM hidden states into compositional neurosymbolic vectors, enabling symbolic algorithms to operate within a high-dimensional vector space before decoding results back into the neural network's processing pipeline.

We demonstrate that LLMs naturally develop internally separable representations for symbolic concepts, which our linear and transformer-based encoders can extract with high fidelity. On mathematical reasoning tasks, our approach achieves 88.6% lower cross-entropy loss and solves 15.4 times more problems correctly compared to chain-of-thought prompting and LoRA fine-tuning, while preserving performance on non-mathematical tasks through selective intervention.

Beyond arithmetic, we extend this framework to three applications. First, we enable language-only models to perform visual question answering by encoding segmented images as queryable VSA representations, achieving 92% accuracy without requiring multimodal architectures. Second, we demonstrate environment navigation where LLMs use spatial semantic pointers to interpret and act upon grid-based worlds according to natural language instructions. Third, we address the context length limitations of LLMs by compressing reasoning histories into VSA representations, maintaining performance on iterative problem-solving tasks while avoiding quadratic scaling costs.

Our results establish VSA-based neurosymbolic integration as a practical approach for augmenting neural language models with symbolic reasoning capabilities, providing both theoretical insights into LLM representations and practical improvements across diverse reasoning tasks. This work contributes to the broader goal of creating AI systems that combine the flexibility of neural networks with the precision and interpretability of symbolic computation. Code and data are available at https://github.com/vdhanraj/Neurosymbolic-LLM.

Acknowledgments

I would firstly like to thank my supervisor Chris Eliasmith for his immense support throughout the past few years. His guidance and insights have helped shaped both my work as well as my understanding of neuroscience and AI.

I would also like to thank the members of the CNRG lab for their collaboration, feedback, and support throughout my master's.

Finally, I would like to thank my family and friends for their constant encouragement, and my partner, Navya, for her patience, love, and steady support throughout the years.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large language models (LLMs) have achieved remarkable success across diverse natural language tasks, yet they face fundamental limitations when confronted with problems requiring symbolic reasoning and precise rule-following. While LLMs excel at statistical pattern matching and can generate fluent, contextually appropriate text, their probabilistic nature becomes a liability in domains demanding logical consistency, mathematical precision, and systematic manipulation of structured information. This thesis addresses these limitations through neurosymbolic integration, introducing methods that enable LLMs to extract, manipulate, and reason with symbolic representations while maintaining their natural language capabilities.

1.1 The Reasoning Gap in Large Language Models

Modern LLMs struggle with multiple challenges that limit their effectiveness in complex reasoning tasks. First, they lack the ability to adhere to precise rules in their inference. Despite their impressive performance on many benchmarks, LLMs frequently fail on tasks requiring strict logical consistency, producing plausible-sounding but fundamentally flawed reasoning chains [94, 36]. For instance, when solving mathematical problems, LLMs may violate basic algebraic properties or generate arithmetic errors that would be trivial for symbolic systems to avoid. This stems from their fundamental architecture: each token generation involves sampling from a probability distribution learned from text patterns, with no explicit representation of logical rules or constraints.

Second, they lack the ability to efficiently represent and manipulate abstract, compositional structures. This is revealed by the fact that the performance of LLMs on reasoning tasks in non-text domains significantly trails their reasoning capabilities in text based domains. The most prominent example of this is the ARC-AGI-1 and ARC-AGI-2 benchmarks, where state of the art LLMs perform significantly worse than humans [20, 21], whereas for many text-based reasoning benchmarks such as GSM8K, LLMs outperform humans [152, 77]. While there could be multiple possible reasons for this disparity in performance (e.g., the training data of LLMs aligns less with ARC type problems and more with GSM8K type problems), recent research has shown that the representations that these models form and can manipulate for a given reasoning problem are critical in its ability to solve them effectively [74].

Finally, LLMs suffer from severe performance degradation as context length increases, a phenomenon known as "context rot" [85, 59]. In addition to the computational cost of attention mechanisms scaling quadratically with sequence length making long-context inference prohibitively expensive, the model's ability to maintain coherent reasoning deteriorates with increasing context length as it must attend to thousands of tokens simultaneously. This limitation is particularly problematic for complex reasoning tasks that require extensive exploration, hypothesis testing, and backtracking, since they naturally generate large amounts of intermediate context. Current approaches like chain-of-thought prompting exacerbate this issue by explicitly expanding the token sequence with reasoning steps, creating a tension between thoroughness and computational feasibility.

1.2 Vector Symbolic Algebras as a Bridge

Vector Symbolic Algebras (VSAs) offer a principled framework for addressing these limitations simultaneously. VSAs enable the construction of high-dimensional vector representations that support compositional structure and symbolic manipulation while remaining compatible with neural network architectures [105, 48]. Unlike traditional symbolic systems that require discrete, rigid data structures, VSAs encode symbolic information in continuous vector spaces where operations like binding, unbinding, and bundling can represent complex relationships, hierarchies, and abstractions.

The key insight is that VSAs can represent diverse abstract data types, such as graphs, trees, logical propositions, and spatial relationships within a unified mathematical framework. These representations are not just embeddings but support systematic manipulation through well-defined algebraic operations. For example, a VSA can encode a mathematical expression's syntactic tree, allowing algebraic transformations to be performed directly in the vector space. Similarly, spatial scenes can be represented as compositional structures where objects, their properties, and relationships are bound together in queryable formats.

This compositional capability extends naturally to context compression. Rather than maintaining entire search trees or reasoning trajectories as token sequences, VSAs can encode this information hierarchically, preserving essential structure while dramatically reducing memory requirements and context sizes. A search process exploring multiple hypotheses can maintain a compressed, hierarchical representation of which branches were promising, which failed, and why, information that would require multiple tokens to express in natural language but can be captured in a fixed-dimensional VSA.

1.3 Neurosymbolic Integration: Bridging Neural and Symbolic Reasoning

The core contribution of this thesis is a methodology for integrating VSA-based symbolic reasoning directly into the computational flow of transformer-based language models. Our approach operates on three levels: extraction of symbolic representations from LLM hidden states, manipulation of these representations using symbolic algorithms, and injection of results back into the model's processing pipeline (see Figure 1.1).

Our neurosymbolic integration method demonstrates that LLMs naturally develop internally separable representations for symbolic concepts during training. By training linear encoders to map hidden states to VSA representations, we reveal that transformer models already encode numerical operands, operators, and other structured information in distinguishable subspaces, a finding with important implications for mechanistic interpretability [82]. This separability enables us to extract problem specifications from the model's internal representations, apply precise symbolic algorithms, and return solutions in a format the model can incorporate into its ongoing computation.

The architecture we develop is computationally efficient, adding only $O(dv + v \log v)$ operations per forward pass, where d is the hidden dimension and v is the VSA dimensionality. This overhead is negligible compared to the transformer's baseline complexity while enabling dramatic improvements in reasoning accuracy. On mathematical reasoning tasks, our method achieves 88.6% lower cross-entropy loss and solves 15.4 times more problems correctly compared to chain-of-thought prompting and LoRA fine-tuning.

1.4 Beyond Arithmetic: Extending Neurosymbolic Integration

While our initial work focuses on mathematical reasoning, the neurosymbolic framework extends naturally to diverse domains requiring structured reasoning. This thesis explores three key applications that demonstrate the versatility of VSA-based integration:

Visual Question Answering: By encoding segmented visual scenes as compositional VSAs, we enable language-only models to perform visual reasoning without architectural modifications. Objects, their properties, and spatial relationships are represented in a queryable symbolic format that LLMs can manipulate to answer questions about images. This approach sidesteps the discretization problems of traditional multimodal models while maintaining interpretable representations of visual content.

Environment Navigation: We demonstrate how spatial semantic pointers (VSA representations of spatial relationships) enable LLMs to navigate grid-based environments according to natural language instructions. The model learns to query and manipulate symbolic representations of its surroundings, making decisions that require spatial reasoning without explicitly tokenizing its environment.

Context Compression: We show how VSAs can maintain compressed representations of an LLM's reasoning history, enabling extended problem-solving without the computational burden of processing entire token histories. Failed solution attempts are encoded as compositional structures that guide future exploration while requiring constant rather than linear memory with respect to the number of attempts.

1.5 Thesis Organization

This thesis is organized as follows:

Chapter 2 provides background on large language models, covering transformer architectures, training objectives, and the RLHF alignment process that attempts to improve reasoning through trajectory-level optimization.

Chapter 3 examines current approaches to improving LLM reasoning, from beam search and chain-of-thought prompting to recent developments in tree of thought approaches and large reasoning models, highlighting their limitations in the token space.

Chapter 4 discusses interpretability methods including linear probes, sparse autoencoders, and attribution graphs, providing context for understanding how our neurosymbolic approach extracts and utilizes internal representations.

Chapter 5 introduces Vector Symbolic Algebras, explaining their mathematical foundations and capacity for representing compositional structures.

Chapter 6 presents our core neurosymbolic integration methodology, detailing the architecture for extracting, manipulating, and injecting symbolic representations during LLM inference.

Chapter 7 provides experimental results demonstrating the effectiveness of our approach on mathematical reasoning tasks across different model sizes and problem types.

Chapter 8 extends the methodology with transformer-based encoders, showing how architectural improvements enable better handling of diverse input formats with limited training data.

Chapter 9 explores applications to visual question answering, environment navigation, and context compression, demonstrating the broad applicability of neurosymbolic integration.

Chapter 10 concludes with a discussion of the implications of this work for the future of AI systems that combine neural and symbolic reasoning.

1.6 Contributions

This thesis makes the following contributions to the field of neurosymbolic AI. It provides:

- A novel method for integrating symbolic reasoning into large language models through Vector Symbolic Algebras, demonstrating that symbolic algorithms can operate directly within neural network computations
- Evidence that LLMs naturally develop separable internal representations for symbolic concepts, revealed through our encoder analysis
- Substantial improvements in mathematical reasoning performance, outperforming existing methods by an order of magnitude on several metrics
- Extensions demonstrating the applicability of neurosymbolic integration to visual reasoning, spatial navigation, and context compression
- A framework for understanding the complementary nature of statistical and symbolic reasoning in artificial intelligence systems

By establishing this bridge between neural and symbolic computation, this work contributes to the broader goal of creating AI systems that combine the adaptability, robustness, and pattern matching capabilities of neural networks with the precision, interpretability, and out-of-distribution capabilities of symbolic reasoning.

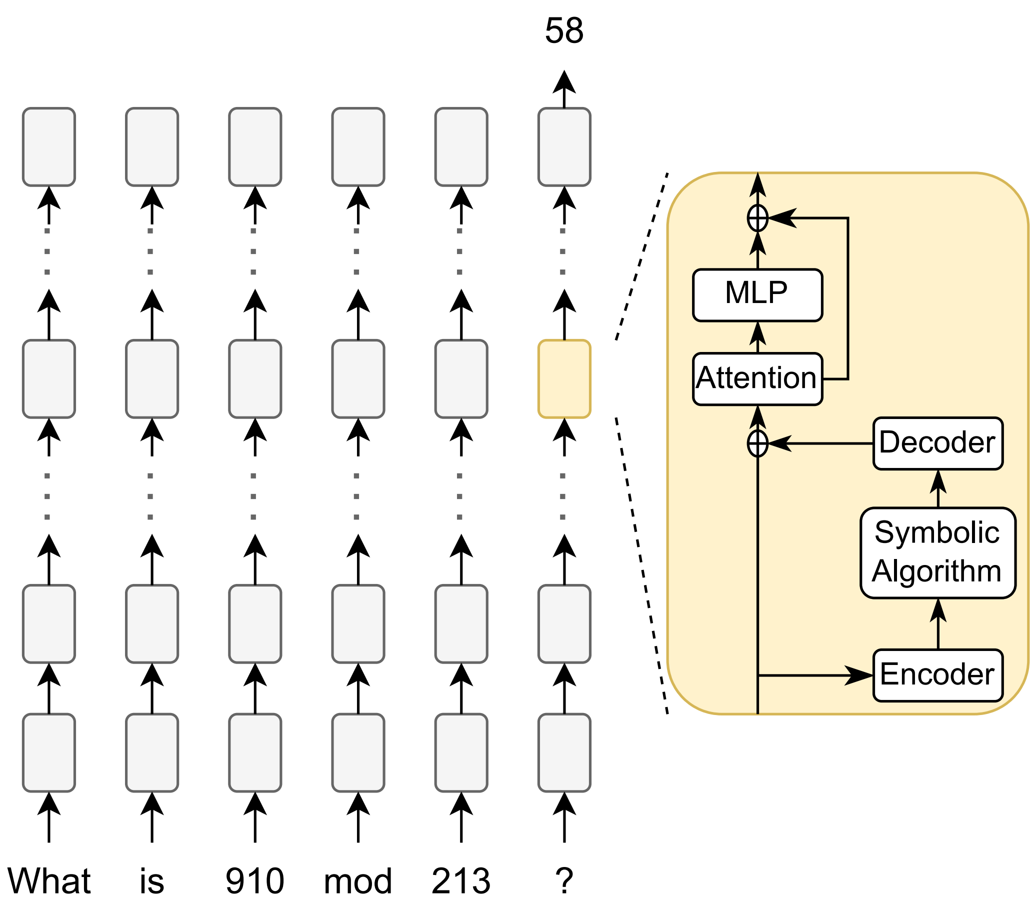


Figure 1.1: A diagram of our method, showing how LLM hidden states are converted into compositional neurosymbolic representations. The encoder network converts the LLM hidden state to a neurosymbolic vector which can be queried to obtain the ones, tens, and hundreds digit of each number, as well as the 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.

Chapter 2

Language Models

2.1 Background: Large Language Models

2.1.1 Introduction to Language Models

Language models are probabilistic models that learn to predict the likelihood of sequences of words or tokens in natural language. At their core, language models learn the conditional probability distribution $P(x_t|x_1, x_2, ..., x_{t-1})$, where x_t represents the token at position t given all previous tokens in the sequence. Modern large language models (LLMs) have evolved from simple n-gram models to sophisticated neural architectures capable of capturing complex linguistic patterns and world knowledge through training on vast text corpora [10, 93].

The fundamental task of language modeling—predicting the next token in a sequence—serves as a powerful self-supervised learning objective that requires no manual annotation. This seemingly simple objective forces models to develop rich internal representations that capture syntax, semantics, factual knowledge, and even reasoning patterns, as accurately predicting the next token often requires understanding context at multiple levels of abstraction [112, 13].

2.1.2 The Transformer Architecture

The transformer architecture, introduced by Vaswani et al. [132], fundamentally changed how sequence modeling is approached in deep learning. Unlike recurrent neural networks (RNNs) that process sequences token by token, maintaining a hidden state that gets updated at each time step [57, 19], transformers process entire sequences simultaneously through a mechanism called self-attention.

Self-Attention Mechanism

The self-attention mechanism allows each token in a sequence to attend to all other tokens, learning which relationships are most relevant for understanding the current context [7, 132]. For a sequence of input tokens $\mathbf{X} = [x_1, x_2, ..., x_n]$, each token x_i is first embedded into a d-dimensional vector representation. These embeddings are then transformed into three different representations through learned linear projections:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q \quad \text{(Queries)} \tag{2.1}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}_K \quad \text{(Keys)} \tag{2.2}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_V \quad \text{(Values)} \tag{2.3}$$

where $\mathbf{W}_Q, \mathbf{W}_K \in \mathbb{R}^{d \times d_k}$ and $\mathbf{W}_V \in \mathbb{R}^{d \times d_v}$ are learned projection matrices.

The attention weights are computed by taking the dot product between queries and keys, scaling by the square root of the key dimension to prevent gradient saturation, and applying a softmax function [132]:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{2.4}$$

The resulting attention matrix has dimensions $n \times n$, where element (i, j) represents how much token i should attend to token j. This mechanism allows the model to learn complex dependencies between tokens regardless of their distance in the sequence.

Multi-Head Attention

Rather than computing a single attention function, transformers employ multi-head attention, which runs multiple attention operations in parallel with different learned projections [132]:

$$\text{MultiHead}(\mathbf{X}) = \text{Concat}(\text{head}_1, ..., \text{head}_h)\mathbf{W}_O \tag{2.5}$$

where each head is computed as:

$$\text{head}_i = \text{Attention}(\mathbf{XW}_{Q_i}, \mathbf{XW}_{K_i}, \mathbf{XW}_{V_i}) \tag{2.6}$$

This design allows the model to attend to information from different representation subspaces at different positions simultaneously, with each head potentially learning to capture different types of relationships (e.g., syntactic dependencies, semantic similarity, coreference) [25, 133].

Parallelization Advantages

The self-attention mechanism's ability to process all tokens simultaneously provides significant computational advantages during training. Unlike RNNs, where the computation for token t depends on the hidden state from token $t-1$, creating a sequential bottleneck, transformers can compute attention for all positions in parallel [132]. This parallelization enables:

1. **Efficient GPU utilization**: Modern GPUs excel at parallel matrix operations. The attention computation can be formulated as batched matrix multiplications, which are highly optimized on GPU hardware [27].
2. **Reduced training time**: For a sequence of length n, RNNs require $O(n)$ sequential operations, while transformers can process the entire sequence in $O(1)$ sequential operations (though with $O(n^2)$ total computation due to all-to-all attention) [132, 67].
3. **Longer context windows**: The parallel nature allows training on longer sequences without the gradient vanishing problems that plague RNNs over long distances [57, 132].

2.1.3 Transformer Layers and Information Flow

A complete transformer layer combines multi-head attention with position-wise feedforward networks and residual connections [54, 132]. The computation for layer l can be expressed as:

$$\mathbf{H}_l^{\text{attn}} = \text{LayerNorm}(\mathbf{H}_{l-1} + \text{MultiHead}(\mathbf{H}_{l-1})) \tag{2.7}$$
$$\mathbf{H}_l = \text{LayerNorm}(\mathbf{H}_l^{\text{attn}} + \text{FFN}(\mathbf{H}_l^{\text{attn}})) \tag{2.8}$$

where $\mathbf{H}_l$ represents the hidden states at layer l, and the feedforward network (FFN) typically consists of two linear transformations with a non-linear activation [55]:

$$\text{FFN}(\mathbf{x}) = \text{GELU}(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2 \tag{2.9}$$

Modern LLMs stack dozens of these layers (e.g., 32 layers in LLaMA 3.1 8B, 80 layers in LLaMA 3.1 70B) [127, 128], with hidden dimensions typically ranging from 2048 to 8192. As information flows through these layers, representations become increasingly abstract and task-specific, with lower layers capturing more surface-level patterns and higher layers encoding more semantic and contextual information [114, 126].

2.1.4 Pretraining Objective and Optimization

Next-Token Prediction

The primary pre-training objective for autoregressive language models is next-token prediction, formulated as maximizing the log-likelihood of the training data [111, 112]:

$$\mathcal{L}_{\text{LM}} = -\sum_{t=1}^{T} \log P(x_t | x_1, x_2, ..., x_{t-1}; \theta) \tag{2.10}$$

where θ represents the model parameters and T is the sequence length. This objective is typically optimized using teacher forcing [142], where the model predicts each token given the true previous tokens rather than its own predictions. Along with causal masking, this enables transformer based LLMs to train on entire sequences in parallel, greatly speeding up training time on GPUs.

Causal Masking

To ensure the model only uses past context for prediction (maintaining causality), transformers employ causal masking in the attention mechanism [111]. This is implemented by setting attention weights to negative infinity for future positions before the softmax:

$$\text{MaskedAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} + \mathbf{M}\right)\mathbf{V} \tag{2.11}$$

where $\mathbf{M}_{ij} = -\infty$ if $i < j$ (preventing token i from attending to future token j), and $\mathbf{M}_{ij} = 0$ otherwise.

Cross-Entropy Loss

The next-token prediction task is formulated as a classification problem over the vocabulary. For a vocabulary of size V, the model outputs a probability distribution over all possible next tokens, and the loss is computed using cross-entropy [49]:

$$\mathcal{L}_{\text{CE}} = -\sum_{i=1}^{V} y_i \log(\hat{y}_i) \tag{2.12}$$

where y_i is the one-hot encoding of the true next token and $\hat{y}_i$ is the predicted probability for token i. In practice, this is computed efficiently using sparse representations since only one element of y is non-zero.

Optimization Techniques

Training large language models requires careful optimization strategies:

1. **Adam optimizer**: Most LLMs are trained using variants of Adam [68], which maintains adaptive learning rates for each parameter:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.13}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{2.14}$$

$$\theta_t = \theta_{t-1} - \alpha \frac{m_t}{\sqrt{v_t} + \epsilon} \tag{2.15}$$

2. **Learning rate scheduling**: Training typically employs a warmup phase followed by cosine or linear decay [88, 87]:

$$\text{lr}(t) = \begin{cases} \frac{t}{T_{\text{warmup}}} \cdot \text{lr}_{\text{max}} & \text{if } t < T_{\text{warmup}} \\ \text{lr}_{\text{max}} \cdot \cos\left(\frac{t - T_{\text{warmup}}}{T_{\text{total}} - T_{\text{warmup}}} \cdot \frac{\pi}{2}\right) & \text{otherwise} \end{cases} \tag{2.16}$$

3. **Gradient clipping**: To prevent gradient explosion, gradients are typically clipped to a maximum norm [102]:

$$g \leftarrow \min\left(1, \frac{\text{max\_norm}}{||g||}\right) \cdot g \tag{2.17}$$

4. **Mixed precision training**: Using 16-bit floating point for forward/backward passes while maintaining 32-bit master weights reduces memory usage and increases training speed [92].

2.1.5 Architectural Variations and Scale

Modern language models exhibit various architectural choices that affect their capabilities and efficiency:

Positional Encoding

Since self-attention is permutation-invariant, transformers require explicit position information. Common approaches include:

1. **Learned positional embeddings**: Trainable vectors added to token embeddings [28]
2. **Sinusoidal encoding**: Fixed encodings using sine and cosine functions of different frequencies [132]
3. **Rotary Position Embeddings (RoPE)**: Used in modern models like LLaMA, encoding positions through rotation matrices applied to query-key pairs [123]

Normalization Strategies

Different normalization approaches affect training stability and model performance:

1. **Layer Normalization**: Applied before (pre-norm) or after (post-norm) each sub-layer [6, 147]
2. **RMSNorm**: A simplified variant that normalizes by root mean square, used in LLaMA [151]:
$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}} \cdot \gamma \tag{2.18}$$

Model Scaling

The relationship between model size, dataset size, and compute budget follows predictable scaling laws [67]. Performance typically improves as a power law with respect to these factors:

$$\mathcal{L}(N, D, C) \approx \left(\frac{N_0}{N}\right)^{\alpha_N} + \left(\frac{D_0}{D}\right)^{\alpha_D} + \left(\frac{C_0}{C}\right)^{\alpha_C} \tag{2.19}$$

where N is the number of parameters, D is the dataset size, C is the compute budget, and α values are empirically determined exponents.

These scaling laws have driven the development of increasingly large models, from GPT-2's 1.5 billion parameters [112] to modern models exceeding 100 billion parameters [13, 23], with each increase in scale unlocking new emergent capabilities while requiring proportionally more training data and compute resources [139].

2.1.6 Alignment through Reinforcement Learning from Human Feedback

After pretraining on vast text corpora, language models possess broad capabilities but lack alignment with human values and preferences. Reinforcement Learning from Human Feedback (RLHF) has emerged as the dominant technique for fine-tuning LLMs to be more helpful, harmless, and honest [24, 122, 101].

Overview of the RLHF Pipeline

The RLHF process consists of three main stages that transform a pretrained language model into an aligned assistant:

1. **Supervised Fine-tuning (SFT)**: The pretrained model is first fine-tuned on high-quality demonstration data of desired behaviours, creating an initial policy π^{SFT} [101].
2. **Reward Model Training**: Human annotators compare pairs of model outputs, and these preferences train a reward model that learns to predict human preferences.
3. **Policy Optimization**: The SFT model is further optimized using reinforcement learning to maximize the learned reward signal while maintaining proximity to the reference policy.

Training the Reward Model

The reward model learns to predict human preferences over text completions. Given a prompt x and two completions y_1 and y_2, human annotators indicate which response they prefer. The reward model $r_\phi(x, y)$ is typically initialized from a pretrained LLM and trained to minimize the negative log-likelihood of the preferred completion being ranked higher [122]:

$$\mathcal{L}_{\text{RM}} = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log \sigma(r_\phi(x, y_w) - r_\phi(x, y_l))\right] \tag{2.20}$$

where y_w is the preferred (winning) completion, y_l is the less preferred (losing) completion, and σ is the sigmoid function. This Bradley-Terry model [12] assumes that the probability of preferring y_1 over y_2 is proportional to $\exp(r_\phi(x, y_1) - r_\phi(x, y_2))$.

Importantly, the reward model scores entire responses rather than individual tokens, allowing it to capture holistic properties like coherence, helpfulness, and factual accuracy that emerge only at the response level [8].

Policy Optimization with PPO

Once the reward model is trained, the SFT model is treated as a policy π_θ in a reinforcement learning framework. For each prompt x, the policy generates a trajectory (response) $y = [y_1, y_2, ..., y_T]$, and the reward model provides a scalar reward $r_\phi(x, y)$ for the complete response.

The optimization objective combines the reward signal with a KL divergence penalty to prevent the policy from deviating too far from a reference policy π_{ref} (typically the SFT model) [154, 101]:

$$\mathcal{J}(\theta) = \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_\theta(\cdot|x)} \left[r_\phi(x, y) - \beta \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)} \right] \tag{2.21}$$

where β controls the strength of the KL penalty. This objective is optimized using Proximal Policy Optimization (PPO) [117], which provides stable training through a clipped surrogate objective.

PPO Implementation Details

PPO prevents large policy updates that could destabilize training. The policy gradient is computed using importance sampling with a clipped objective:

$$\mathcal{L}^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(\rho_t(\theta) \hat{A}_t, \text{clip}(\rho_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right] \tag{2.22}$$

where $\rho_t(\theta) = \frac{\pi_\theta(y_t|x, y_{<t})}{\pi_{\text{old}}(y_t|x, y_{<t})}$ is the probability ratio between the current and old policies, $\hat{A}_t$ is the advantage estimate, and ϵ is the clipping parameter (typically 0.1-0.2).

The advantage function estimates how much better an action is compared to the average:

$$\hat{A}_t = r_\phi(x, y) - V(x, y_{<t}) \tag{2.23}$$

where $V(x, y_{<t})$ is a value function (often learned jointly) that estimates the expected reward from the current state. In practice, generalized advantage estimation (GAE) [116] is used to reduce variance:

$$\hat{A}_t^{\text{GAE}} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{2.24}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal difference error, γ is the discount factor, and λ balances bias and variance.

Impact on Planning and Response Quality

RLHF fundamentally changes how LLMs generate text by optimizing for complete response quality rather than local token predictions. This has several important effects:

1. **Global coherence**: The reward model evaluates entire responses, encouraging the model to maintain consistency and logical flow throughout long generations [122].
2. **Implicit planning**: By optimizing for rewards assigned to complete trajectories, the model learns to "plan ahead" during generation, choosing tokens that lead to better overall responses rather than locally optimal continuations [75].
3. **Trade-off management**: The model learns to balance multiple objectives (helpfulness, harmlessness, honesty) that may conflict at the token level but must be reconciled at the response level [9].

This trajectory-level optimization is particularly important for complex reasoning tasks where early decisions constrain later possibilities. For instance, in mathematical problem-solving, choosing an inefficient approach early on may make the problem unsolvable, even if each individual step appears reasonable [79].

Challenges and Limitations

While RLHF has proven effective for alignment, several challenges remain:

1. **Reward hacking**: Models may exploit imperfections in the reward model, generating responses that score highly but violate the spirit of human preferences [45].

2. **Mode collapse**: Excessive optimization can cause the model to generate repetitive, safe responses that lack diversity [69].

3. **Computational cost**: RLHF requires generating many samples per prompt and computing rewards for each, making it significantly more expensive than supervised fine-tuning [113].

4. **Human feedback quality**: The reward model is only as good as the human preferences it learns from, which may be inconsistent, biased, or poorly specified [14].

Recent work has explored alternatives like Direct Preference Optimization (DPO) [113] and Constitutional AI [9] that aim to achieve similar alignment benefits with reduced complexity or improved scalability.

Additionally, despite improving planning capabilities through trajectory-level optimization, RLHF does not alter the fundamental architecture of the language model—it remains a next-token predictor operating on statistical patterns [91, 90]. While the reward signal encourages the model to consider longer-term consequences when selecting tokens, each generation step still involves sampling from a probability distribution over the vocabulary based on learned correlations in text. The model has no explicit representation of logical rules, causal relationships, or systematic reasoning procedures.

This architectural limitation means that even RLHF-trained models struggle with tasks requiring strict logical consistency, mathematical precision, or multi-step reasoning where each step must be verifiably correct [36, 94]. The statistical nature of token prediction can lead to failures in reasoning tasks that would be trivial for symbolic systems, as the model may generate plausible-sounding but fundamentally flawed reasoning chains that violate basic logical principles [11, 144]. The next chapter will discuss more deliberate approaches researchers have taken towards improving the reasoning and planning capabilities of LLMs.

Chapter 3

Reasoning in LLMs

This chapter surveys approaches that aim to improve the reasoning performance of large language models (LLMs). We progress from RL methods specifically designed to fine-tune LLMs towards reasoning, to inference time compute methods that provide more accurate responses by performing search or reasoning in the token space.

3.1 Approaches to Improving LLM Reasoning

Despite the fundamental architectural constraints of next-token prediction, researchers have developed various techniques to enhance the reasoning capabilities of language models. These approaches range from inference-time strategies that modify how tokens are sampled, to training-time methods that fundamentally change how models learn to approach complex problems.

3.2 Group Relative Policy Optimization

Group Relative Policy Optimization (GRPO) represents a significant advancement in training-time reasoning optimization by incorporating search directly into the learning process [118]. Unlike standard RLHF which typically generates a single response per prompt, GRPO samples multiple trajectories during training and uses their relative performance to guide policy updates.

For each training prompt x, GRPO samples G different responses $\{y^{(1)}, y^{(2)}, ..., y^{(G)}\}$ from the policy π_θ using temperature sampling. Each response receives a reward $r(x, y^{(i)})$, and the relative ranking of these rewards within the group is used rather than absolute rewards. The full GRPO loss function is:

$$\mathcal{L}_{\text{GRPO}}(\theta) = -\mathbb{E}_{x\sim\mathcal{D}}\left[\sum_{i=1}^{G} \hat{A}^{(i)} \log \pi_\theta(y^{(i)}|x)\right] \tag{3.1}$$

where the advantage $\hat{A}^{(i)}$ is computed using group normalization:

$$\hat{A}^{(i)} = \frac{r(x, y^{(i)}) - \mu_r}{\sigma_r} \tag{3.2}$$

with $\mu_r = \frac{1}{G}\sum_{j=1}^{G} r(x, y^{(j)})$ being the mean reward across the group and σ_r the standard deviation.

This formulation has several important properties:

1. **Exploration during training**: By generating multiple responses with temperature sampling, the model explores diverse reasoning paths during training rather than at inference time

2. **Relative optimization**: The model learns to prefer better trajectories over worse ones from the same distribution, rather than optimizing against a fixed baseline of good trajectories provided by a training dataset

3. **Variance reduction**: Group normalization provides a natural baseline that reduces gradient variance compared to standard policy gradient methods

By allowing the model to gather multiple possible solutions to an answer and receive relative rewards for each answer, GRPO essentially performs a search over possible reasoning trajectories during training. The model generates multiple attempts at solving the same problem, evaluates their quality through the reward function, and then updates its parameters to increase the probability of successful reasoning patterns while decreasing the probability of failed attempts. This is particularly effective for mathematical reasoning tasks where multiple solution paths exist but vary significantly in their correctness and efficiency [118].

To prevent the model from overfitting to the reward signal, GRPO often includes a KL divergence term similar to standard RLHF:

$$\mathcal{L}_{\text{GRPO-KL}}(\theta) = -\mathbb{E}_{x\sim\mathcal{D}}\left[\sum_{i=1}^{G} \hat{A}^{(i)} \log \pi_\theta(y^{(i)}|x) - \beta \text{KL}(\pi_\theta || \pi_{\text{ref}})\right] \tag{3.3}$$

This approach effectively teaches the model to perform internal search by learning from the contrast between successful and unsuccessful reasoning attempts on the same problems, leading to more robust reasoning capabilities than models trained with single-trajectory methods.

While GRPO provides a powerful unsupervised approach to train models to select good reasoning traces, it does not fundamentally change the dynamic of the model during inference. That is, the LLM still predicts one token at a time during inference, leading to difficulties performing reasoning for more complicated tasks.

3.3 Beam Search and Decoding Strategies

One of the earliest approaches to improving LLM outputs involves modifying the decoding strategy during inference. While greedy decoding selects the most probable token at each step, beam search maintains k parallel hypotheses and explores multiple paths through the probability space [44].

For a beam width k, the algorithm maintains the top-k most likely sequences at each decoding step:

$$\mathcal{B}_{t+1} = \text{top-}k \left\{ (y_{1:t}, y_{t+1}) : y_{1:t} \in \mathcal{B}_t, y_{t+1} \in \mathcal{V} \right\} \tag{3.4}$$

where $\mathcal{B}_t$ is the beam at time t and sequences are scored by their log-probability:

$$\text{score}(y_{1:T}) = \sum_{t=1}^{T} \log p(y_t | y_{1:t-1}) \tag{3.5}$$

While beam search can improve performance on some reasoning tasks by avoiding early mistakes in the generation process, it suffers from several limitations for complex reasoning [58]. The method still fundamentally relies on local token probabilities and can amplify the model's tendency toward generic or repetitive outputs. Moreover, beam search provides no mechanism for the model to backtrack from conceptually wrong approaches, as it only considers syntactic alternatives rather than semantic reasoning paths.

3.4 Chain-of-Thought Prompting

Chain-of-thought (CoT) prompting represents a paradigm shift in how LLMs approach reasoning tasks. Instead of directly generating answers, CoT encourages models to produce intermediate reasoning steps [140]. This can be achieved through few-shot examples that demonstrate step-by-step reasoning:

```
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis
   balls. Each can has 3 tennis balls. How many tennis balls
   does he have now?
A: Roger started with 5 tennis balls. He bought 2 cans of
   3 tennis balls each, which is 2 × 3 = 6 tennis balls.
   So in total, he has 5 + 6 = 11 tennis balls.
```

Zero-shot CoT can be triggered with simple prompts like "Let's think step by step" [70]. The effectiveness of CoT appears to emerge with model scale, with larger models showing more significant improvements [139].

The mechanism behind CoT's success likely involves several factors:

1. **Intermediate supervision**: Each reasoning step provides implicit supervision for subsequent steps
2. **Extended computation**: Generating reasoning tokens effectively gives the model more "thinking time" [150]
3. **Error localization**: Explicit reasoning steps make it easier to identify where reasoning fails

However, CoT has notable limitations. The reasoning chains may be post-hoc rationalizations rather than faithful representations of the model's computation [130], and models can generate convincing but incorrect reasoning paths [72].

3.5 Self-Consistency and Ensemble Methods

Self-consistency extends CoT by sampling multiple reasoning paths and selecting the most common answer [136]:

$$\hat{a} = \arg\max_a \sum_{i=1}^{n} \nVdash[a_i = a] \tag{3.6}$$

where a_i is the answer from the i-th sampled reasoning path. This approach leverages the intuition that correct reasoning can follow multiple valid paths, while errors are more randomly distributed.

3.6 Tree of Thoughts

Tree of Thoughts (ToT) extends the chain-of-thought paradigm by explicitly modeling the reasoning process as a search through a tree of possible reasoning paths [148, 108]. Rather than generating a single linear chain of reasoning, ToT methods systematically explore multiple reasoning trajectories, evaluate their promise, and backtrack when necessary.

The ToT framework decomposes problem-solving into three key components:

1. **Thought decomposition**: Breaking down the reasoning process into discrete intermediate steps or "thoughts"
2. **Thought generation**: Sampling multiple candidate thoughts at each reasoning step
3. **State evaluation**: Assessing the promise of partial solutions to guide the search

Formally, ToT constructs a search tree where each node s represents a partial solution consisting of the input x and a sequence of thoughts $[z_1, ..., z_i]$. At each node, the method:

$$\text{Children}(s) = \{s \oplus z : z \sim \pi_{\text{thought}}(\cdot|s)\} \tag{3.7}$$

where π_{thought} generates k candidate next thoughts.

The search process requires an evaluation function $V(s)$ to assess the promise of each state. This can be implemented through:

1. **Value prompting**: Asking the LLM to rate the state on a scale (e.g., "Rate this partial solution from 1-10")
2. **Vote prompting**: Sampling multiple completions and using majority voting
3. **Learned evaluators**: Training separate models to score partial solutions

ToT employs standard tree search algorithms to navigate the reasoning space. Breadth-first search (BFS) maintains a frontier of the b most promising states at each depth:

$$\mathcal{F}_{t+1} = \text{top-}b \left\{ s' : s' \in \bigcup_{s \in \mathcal{F}_t} \text{Children}(s) \right\} \tag{3.8}$$

Alternatively, depth-first search (DFS) with backtracking allows deeper exploration:

$$\text{next}(s) = \begin{cases} \arg\max_{s' \in \text{Children}(s)} V(s') & \text{if } V(s) > \theta \\ \text{backtrack}(s) & \text{otherwise} \end{cases} \tag{3.9}$$

The main advantage of ToT is its ability to explore and abandon reasoning paths. For example, in creative writing tasks, ToT can generate multiple paragraph options and select the most coherent continuation. In mathematical problem-solving, it can try different problem-solving strategies and backtrack when reaching dead ends [148].

However, ToT faces significant computational challenges. The number of LLM calls grows as $O(bkd)$ for BFS with branching factor k, beam width b, and depth d. This makes ToT substantially more expensive than standard CoT, limiting its practical deployment. Additionally, the quality of the evaluation function $V(s)$ critically determines search effectiveness, yet LLMs often struggle to accurately assess their own partial solutions [108].

3.7 Large Reasoning Models

Recent work on Large Reasoning Models (LRMs) represents a fundamental shift in architecture, treating reasoning as a search problem in a latent space [100, 109]. LRMs augment the standard transformer with:

1. **Latent reasoning tokens**: Special tokens that represent intermediate reasoning states not directly tied to natural language
2. **Reasoning-specific training**: Models are trained on synthetic data with explicit reasoning annotations
3. **Verification mechanisms**: Built-in components that check consistency of reasoning steps

The training objective for LRMs often combines multiple components:

$$\mathcal{L}_{\text{LRM}} = \mathcal{L}_{\text{predict}} + \alpha\mathcal{L}_{\text{verify}} + \beta\mathcal{L}_{\text{consistency}} \tag{3.10}$$

where $\mathcal{L}_{\text{verify}}$ ensures reasoning steps are verifiable and $\mathcal{L}_{\text{consistency}}$ maintains logical consistency across the reasoning chain.

These models show impressive performance on complex reasoning benchmarks, though they require substantially more compute during both training and inference [120]. By explicitly modeling reasoning as a search process rather than pure next-token prediction, LRMs can explore and evaluate multiple reasoning strategies before committing to an answer.

3.8 Limitations of Current Approaches

Despite these advances, fundamental challenges remain:

1. **Computational cost**: Methods like beam search, self-consistency, tree of thought, and LRMs require generating many more tokens per query
2. **Training data requirements**: LRMs need extensive human annotations or synthetic data
3. **Lack of guarantees**: None of these methods provide formal guarantees of correctness or consistency
4. **Abstractions limited to token space**: All of the mentioned methods train the model to reason better only in the token space, leading to challenges when attempting to perform reasoning around more abstract objects [20]

These limitations motivate the exploration of neurosymbolic approaches that combine the flexibility of neural netowrks with the reliability of symbolic reasoning, as discussed in Chapter 6.

Chapter 4

LLM Interpretibility

4.1 Linear Probes

Linear probes are widely used tools for interpreting the internal representations of LLMs [56, 84]. 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 [37].

Previous work has extended this approach with digit-specific circular probes, which attempt to decompose numerical representations into their constituent digits using circular algebra [38]. 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 approach is thus particularly well-suited for complex numerical reasoning tasks that require flexible and interpretable encodings.

4.2 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 [99, 73]. These components, often referred to as "concepts," are linearly combined to reconstruct the original input data [37]. 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 [99, 37].

Additionally, the concepts extracted by SAEs are typically non-interpretable by default, requiring manual inspection of activations to identify their semantic meaning [99, 37]. While this can provide insights into LLM internals, it is labour-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.

4.3 Transcoders

Transcoders represent a recent advancement in understanding the computational structure of transformer language models by learning to predict activations at later layers from sparse representations at earlier layers [125, 82]. Unlike linear probes that map hidden states to external labels, transcoders model the internal computation of the network itself, revealing how information flows and transforms across layers. A transcoder learns to predict the hidden state at layer j from a sparse representation of layer i, typically using a sparse autoencoder to decompose the earlier layer's hidden state into interpretable features. The transcoder architecture consists of an input sparse autoencoder that identifies active features at the source layer, followed by a sparse linear transformation that maps these features to the target layer, with regularization to ensure sparsity and interpretability of the learned connections.

The training objective for transcoders minimizes reconstruction error while maintaining sparsity through L1 regularization on the transformation weights. This approach reveals important aspects of transformer computation, including feature circuits that represent sparse pathways through which specific concepts flow across layers, computational motifs that implement common operations, and layer specialization patterns that show how different layers contribute to different aspects of the computation. For mathematical reasoning tasks, transcoders have revealed specialized circuits for numerical operations, demonstrating how digit representations at early layers transform into arithmetic results at later layers [82]. These discoveries provide mechanistic insights that complement surface-level interpretability methods by revealing not just what information is present, but how it is processed and transformed through the network's depth.

While transcoders provide valuable mechanistic insights, they face several practical limitations. The computational cost of training transcoders scales quadratically with model depth when examining all layer pairs, making comprehensive analysis expensive for large models. Additionally, transcoders inherit the limitations of their underlying SAEs, including potential failure to capture all relevant features and the challenge of interpreting the learned sparse representations. Despite these challenges, transcoders have proven valuable for understanding how transformers implement complex computations, particularly in domains like arithmetic where the transformation from input digits to output results involves multiple intermediate processing steps.

4.4 Attribution Graphs

Attribution graphs extend the transcoder framework to create comprehensive maps of information flow through transformer networks [82]. Rather than examining individual layer-to-layer connections, attribution graphs construct a directed acyclic graph representing all significant computational pathways through the model. The graph consists of vertices representing sparse features from SAEs at each layer, edges representing connections between features across layers, and weights indicating the strength of each connection through gradient-based attribution scores. These attribution scores capture both the sensitivity of downstream features to upstream ones and the actual activation strength during forward passes, providing a quantitative measure of information flow through the network.

The construction of attribution graphs involves a multi-step process beginning with training SAEs for each layer to identify interpretable features, followed by learning sparse transformations between adjacent layers through transcoders, composing these transcoders to identify multi-hop connections, and finally pruning connections below a significance threshold to maintain interpretability. For arithmetic operations, attribution graphs reveal complex computational structures, showing how digit features at early layers connect to carry-bit features at middle layers, which in turn influence the computation of output digits at later layers, with operation-type features gating different computational pathways [96]. This level of mechanistic detail provides insights that are impossible to obtain through static analysis of individual hidden states.

Attribution graphs have proven particularly valuable for understanding reasoning failures in LLMs. When models make arithmetic errors, attribution graphs can pinpoint where information flow breaks down, revealing that many errors stem from interference between digit-processing circuits rather than fundamental computational limitations [52]. The graphs also identify optimal intervention points for improving reasoning, with features having high centrality often serving as bottlenecks where targeted interventions can have outsized effects [135]. This mechanistic understanding enables more precise modifications than approaches based solely on behavioural observations or aggregate performance metrics.

By identifying existing circuits for numerical processing, attribution graphs have the potential to inform the design of neurosymbolic interventions that work with rather than against the model's learned representations, leading to more effective integration strategies. As we will show in Chapter 6, our approach performs intervention on the forward pass of the LLM by modifying the hidden state according to the output of a symbolic algorithm. If an existing attribution graph is prepared for a model, picking both the layer from which to extract representations that become inputs to our symbolic algorithms, and the layer from which we steer the LLMs forward pass towards the correct solution can be done more scientifically, rather than simply trying every layer and measuring the performance (which is what we do in our approach).

While using attribution graphs with our approach may improve the efficiency of training our method, this does not factor into account the large cost of training the attribution graph, as discussed in Section 4.3. Additionally, attribution graphs inherit many of the failures of both transcoders and sparse auto-encoder approaches. Specifically, the concepts it creates are not guaranteed to be atomic, and may not cover all the concepts required for a specific domain. For example, in the domain of mathematics, attribution graphs cannot create representations for every number, since there are an unlimited amount of them. Our approach aims to provide the foundations for a method that can solve these shortcomings by utilizing compositional and hierarchical representations. While our current work does not offer a replacement for methods like attribution graphs, in future work we aim to integrate the concepts introduced in this thesis with methods like attribution graphs to create more interpretable and controllable language models.

Chapter 5

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) [105], 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 vectors for each digit and place value (e.g., *ones*, *tens*, *hundreds*) by binding 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 Sections 5.1 and 5.2.

5.1 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:

$$(\mathbf{x} \circledast \mathbf{y})_i := \sum_{j=1}^{d} x_j y_{((i-j) \bmod d)+1}, \quad i \in \{1, \ldots, d\}, \tag{5.1}$$

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

5.1.1 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 ($\mathbf{ones}$, $\mathbf{tens}$, $\mathbf{hundreds}$):

$$\mathbf{x} = \mathbf{hundreds} \circledast \mathbf{8} + \mathbf{tens} \circledast \mathbf{4} + \mathbf{ones} \circledast \mathbf{2}. \tag{5.2}$$

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 6.1.1 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 vector in a compositional and separable manner. These tag vectors 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:

$$\begin{aligned} \mathbf{x} = \mathbf{n_1} &\circledast (\mathbf{hundreds} \circledast \mathbf{8} + \mathbf{tens} \circledast \mathbf{4} + \mathbf{ones} \circledast \mathbf{2}) + \\ &\mathbf{n_2} \circledast (\mathbf{hundreds} \circledast \mathbf{9} + \mathbf{tens} \circledast \mathbf{1} + \mathbf{ones} \circledast \mathbf{0}) + \\ &\mathbf{problem\_type} \circledast \mathbf{modulo}. \end{aligned} \tag{5.3}$$

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 Equation (5.6) below.

Critically, to encode the structure of numbers, digits are constructed by binding the vector for $\mathbf{1}$ 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 [22, 39].

5.1.2 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:

$$\mathbf{y}^{\dagger} = (y_1, y_d, y_{d-1}, \ldots, y_2), \tag{5.4}$$

where d is the dimensionality.

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

$$\mathbf{x} \approx \mathbf{y}^{\dagger} \circledast \mathbf{z}. \tag{5.5}$$

For example, to query the hundreds digit of the second number in (5.3):

$$\mathbf{result} = \mathbf{hundreds}^{\dagger} \circledast (\mathbf{n_2}^{\dagger} \circledast \mathbf{x}), \tag{5.6}$$

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

5.1.3 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:

$$N \propto \exp\left(\alpha d \epsilon^2\right), \tag{5.7}$$

where α is a constant derived from spherical code packing and the Kabatiansky–Levenshtein bound [65, 105]. 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.

5.2 VSA Structure

Equation (5.3) outlines how we have designed the VSA vectors 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.

5.2.1 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 $\mathbf{hundreds}$, $\mathbf{tens}$, $\mathbf{ones}$, 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.

5.2.2 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 Section 7.3) 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 vector 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 [82].

Chapter 6

Symbolic Integration into Large Language Models

6.1 Methodology

Our method consists of four 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.
4. 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.

6.1.1 Dataset

We released a formally specified, procedurally generated benchmark, the **Symbolic-Math Dataset**[1], to foster reproducible evaluation of arithmetic reasoning in LLMs. The dataset is

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

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.

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, prompting the model to provide responses in a consistent and easy to evaluate format.

6.1.2 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 [50]. 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.

6.1.3 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 1.1). 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 neurosymbolic vectors are structured using the framework described in Section 5.1.1. The encoder is trained using a root mean squared error (RMSE) loss, with the objective of minimizing the difference between the predicted and true neurosymbolic vectors.

6.1.4 Symbolic Processing

During inference, we execute a symbolic algorithm using the neurosymbolic representation produced by the encoder. This is achieved by first performing various queries on the neurosymbolic vector output (as described in Section 5.1.2) to obtain inputs to our symbolic algorithm. In our case, the relevant symbolic information pertaining to our problem is the digits of the two input numbers and the problem type asked. Based on this information, we execute a specific symbolic procedure to obtain the result. In our work, we use Python code to perform our symbolic processing, where we dynamically select the corresponding pre-programmed Python function based on the queried problem type, and provide as input to this function the queried digits of the first and second numbers in the prompt.

Note that while we chose to use Python code for our symbolic processing (mainly due to simplicity and transparency), this type of symbolic processing could also be achieved using purely neural methods. In particular, using the Neural Engineering Framework (NEF), one can theoretically create populations of neurons to approximate any non-linear function [41, 121]. This approach, however, requires an arbitrarily large number of neurons based on the complexity of the non-linear function, and is only guaranteed to work on input samples that are in distribution of the training data. This implies that for input data the model has not seen before, this approach will not scale, which is a critical limitation given that our goal is to use the NEF to perform symbolic computation, which has the property of performing out of distribution tasks.

One approach to circumvent this issue is to construct modular components, each trained to construct a non-linear function with a restricted input domain, and connect them with neural information routing mechanisms [40]. This approach has been demonstrated in models like Spaun [42], which performs symbolic manipulations entirely within a spiking neural network. Specifically, the basal ganglia component of such models can implement action selection and routing mechanisms that effectively perform conditional logic and function selection, capabilities essential for creating symbolic algorithms that function as expected in out of distribution tasks.

Recent work has demonstrated the feasibility of implementing complex mathematical operations using NEF. Joffe and Eliasmith [64] developed a biologically plausible model of mental multiplication entirely in spiking neurons, showing that the framework can handle multi-step arithmetic procedures including repeated addition and rule-based strategies. Their model uses neural latch and flip-flop memory units to maintain intermediate results during computation, with a basal ganglia module selecting between different solution strategies based on the input. This work provides concrete evidence that NEF implementations can perform the kinds of symbolic manipulations required for mathematical reasoning, achieving performance comparable to human subjects on multiplication tasks.

The trade-off is between implementation complexity and computational efficiency: while Python provides a clear, debuggable implementation that leverages existing optimized libraries for arithmetic operations, a fully neural implementation would offer a more biologically plausible and potentially more tightly integrated solution. For the purposes of this work, we prioritized clarity and ease of implementation, but future work could explore replacing our Python functions with NEF-based neural circuits, potentially enabling end-to-end gradient-based learning of both the encoding/decoding networks and the symbolic manipulation itself.

The output of our symbolic processing, whether implemented in Python or through neural circuits, is a number representing the solution to the mathematical problem. This solution must then be converted back into a neurosymbolic vector representation that can be decoded into the LLM's hidden state space, as described in the following section.

6.1.5 Decoding Neurosymbolic States

Once the encoder network is trained, a corresponding linear decoder network is trained to reverse this mapping. The decoder network takes neurosymbolic 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 neurosymbolic 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 1.1) 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 Section 7.8.

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 Section 7.3.

Although the decoder networks are pretrained to reconstruct hidden states corresponding to neurosymbolic 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.

6.1.6 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 computing 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.

6.1.7 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 [141, 70, 137] 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 [153]. CoT has been shown to improve interpretability and correctness in reasoning tasks by enabling the model to break down problems into smaller, manageable components [98, 141]. CoT prompting can be implemented by including examples of detailed reasoning in the training dataset or through few-shot prompting during inference [70]. 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 [61, 146, 134] 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 [78, 60]. This makes fine-tuning computationally efficient and memory-friendly, even for very large models [30]. LoRA modules are typically inserted into the attention mechanism, where they adapt the query, key, and value projections to improve task-specific performance [61]. For mathematical reasoning, LoRA fine-tuning enables the model to learn domain-specific representations and reasoning strategies effectively, while minimizing the computational burden [146].

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.

6.2 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 vector dimensionality (2048); D: maximal digit length (5); p: problem types (10)

6.2.1 Baseline Transformer

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

$$\text{Time}_{\text{train}} = \mathcal{O}(L(n^2 d + n\, d^2)), \tag{6.1}$$
$$\text{Space}_{\text{train}} = \mathcal{O}(Lnd) + \Theta(\#\text{LLM params}) \tag{6.2}$$

6.2.2 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:

$$\text{Time}_{\text{KV}} = \mathcal{O}(L(nd + d^2)) \tag{6.3}$$
$$\text{Space}_{\text{KV}} = \mathcal{O}(Lnd) + \Theta(\#\text{LLM params}) \tag{6.4}$$

6.2.3 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 $W_d \in \mathbb{R}^{v\times d}$.

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.

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 $\text{VSA}_{\text{PROBLEM}}$
        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}

Chapter 7

Results

7.1 Experiments

7.1.1 Evaluation Setup

We evaluate the proposed Neurosymbolic LLM (NS LLM) against three baselines: (i) a **Standard LLM** (frozen, with few-shot prompting), (ii) a **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 6.1.1. 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 6.1.1. 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.

7.1.2 Base LLM

The base LLM is evaluated using the same few-shot prompt format described in Section 7.1.1, 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 Section 7.5, which use Llama 3.2 1B), following the inference procedure and key-value caching mechanism outlined in [50]. The model weights are frozen during evaluation, and no additional fine-tuning is applied.

7.1.3 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 similarity 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 Section 7.6). 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 Section 7.7.

In this study, we intervene at layer 17, as it achieves the lowest encoder reconstruction loss (see Section 7.3). The dimensionality of the VSA vector 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 Section 7.8.

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.

7.1.4 LoRA

To ensure a fair comparison with the NS LLM, we implement a LoRA module with rank 2048, matching the dimensionality of the VSA vectors 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.

7.1.5 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.

7.2 Results

Across all trained problem types, the Neurosymbolic LLM achieves the best overall performance among all models, as shown in Table 7.1. 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 Section 7.4.

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 A). 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.

Table 7.1: Performance of Symbolic, Standard, CoT, and LoRA LLMs on various problem types. Addition and Integer Division were 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
Multiplication	**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	1.310
	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 (OOD)	NS LLM	98.2	0.206
	Standard LLM	**100.0**	**0.000**
	CoT LLM	78.8	2.218
	LoRA LLM	46.5	6.299
Int. Division (OOD)	**NS LLM**	**97.4**	**0.066**
	Standard LLM	95.2	0.148
	CoT LLM	94.3	0.709
	LoRA LLM	72.0	1.797

7.3 Encoder and Decoder Performance

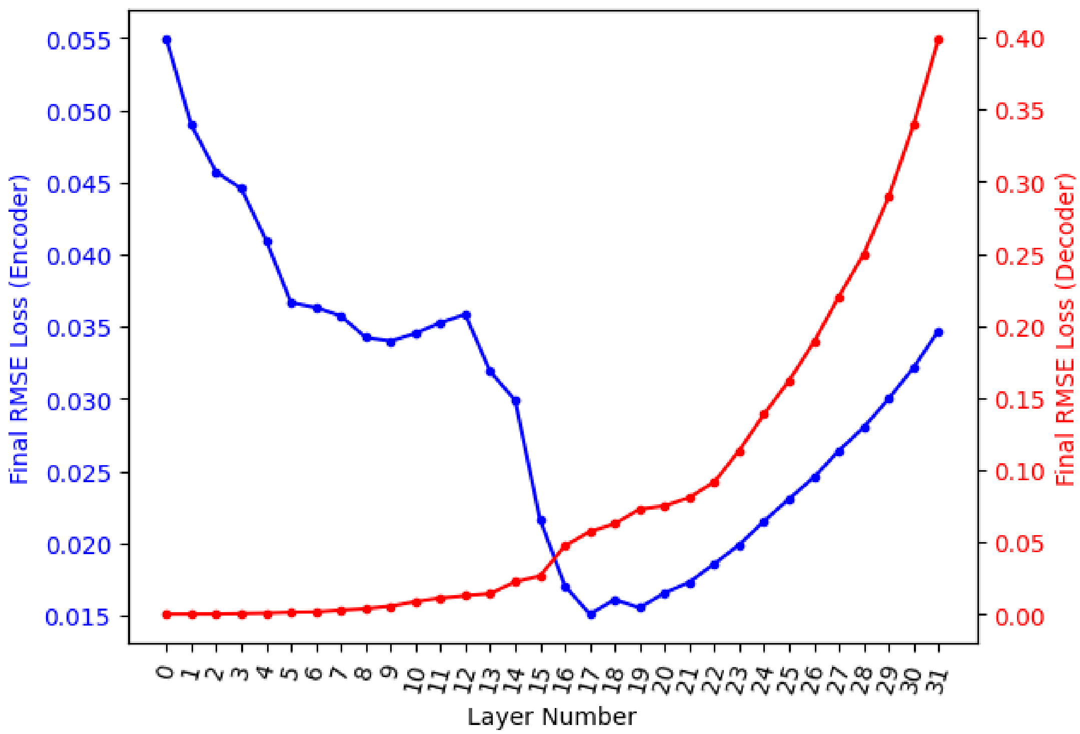


Figure 7.1: Average RMSE loss of the encoder (blue) and decoder (red) across layers of the LLM.

After training, the encoder networks achieve RMSE loss curves shown in Figure 7.1. The results indicate that earlier layers of the LLM are less effective at encoding the problem into neurosymbolic 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:

$$h_{n+1} = f_n(h_n) + h_n, \tag{7.1}$$

$$h_L = h_0 + \sum_{n=1}^{L} f_n(h_{n-1}), \tag{7.2}$$

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.

As shown in Figure 7.1, 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.

7.4 Error Sources

As mentioned in Section 7.2, 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 makes 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 7.2, 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 7.2.

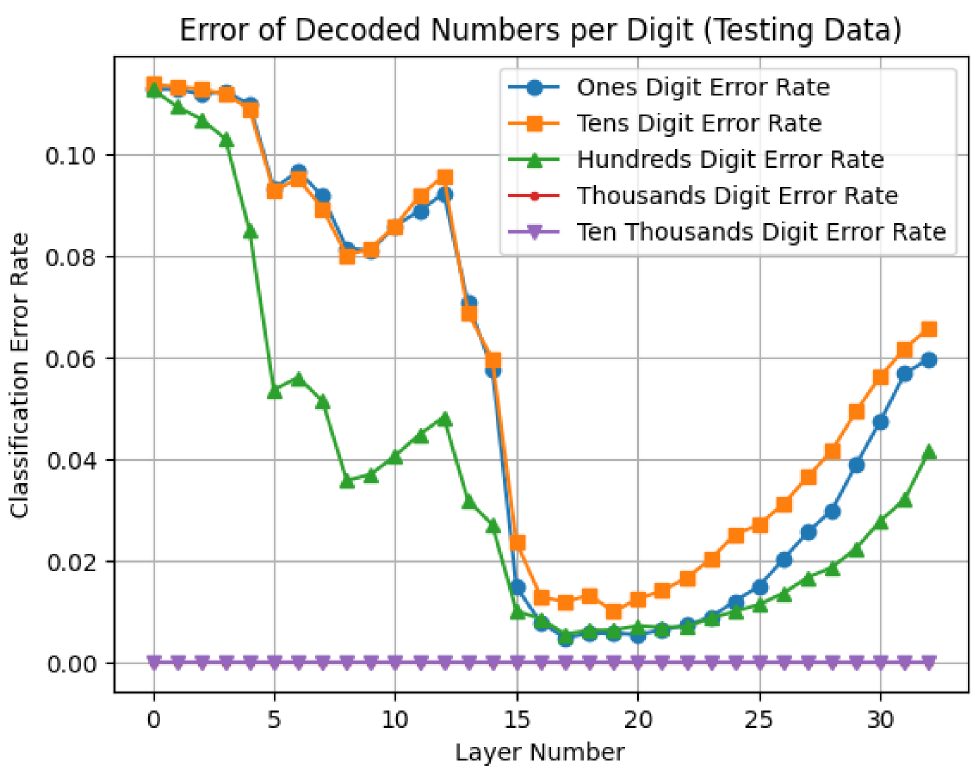


Figure 7.2: Classification Error Rate vs. Layer Number, across all problem types.

7.5 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.

7.5.1 Encoder and Decoder Performance

Figure 7.3 shows the encoder and decoder RMSE loss across different layers for the 1B model. Similar 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 15, indicating successful reconstruction of hidden states after symbolic intervention.

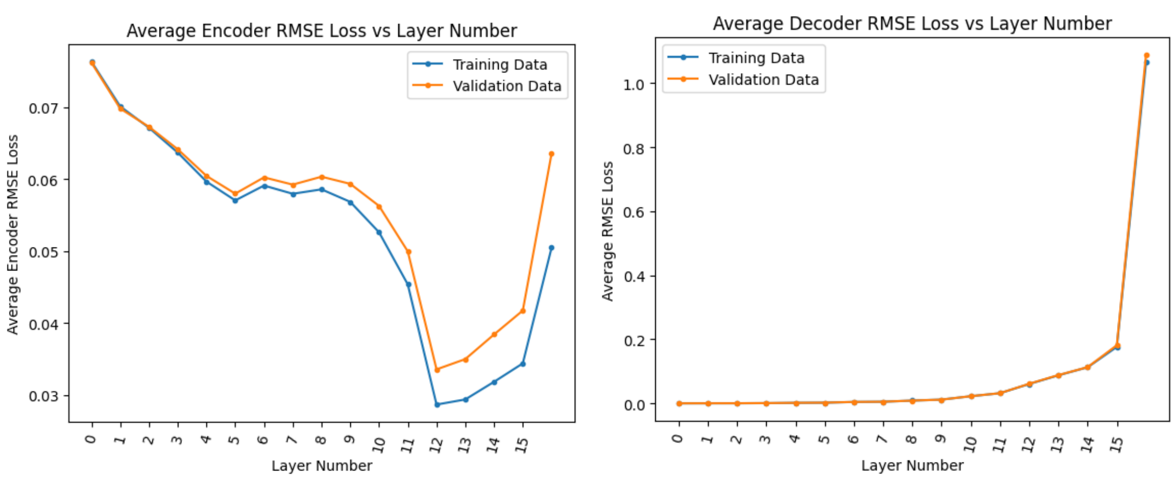


Figure 7.3: Encoder (Top) and decoder (Bottom) RMSE loss across layers for Llama 3.2 1B. Lower values indicate better reconstruction of VSA representations and hidden states respectively.

Figure 7.4 demonstrates that the encoder successfully extracts digit-level information, with all digits achieving under 5% encoding error. The similarity distribution (Figure 7.4) 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.

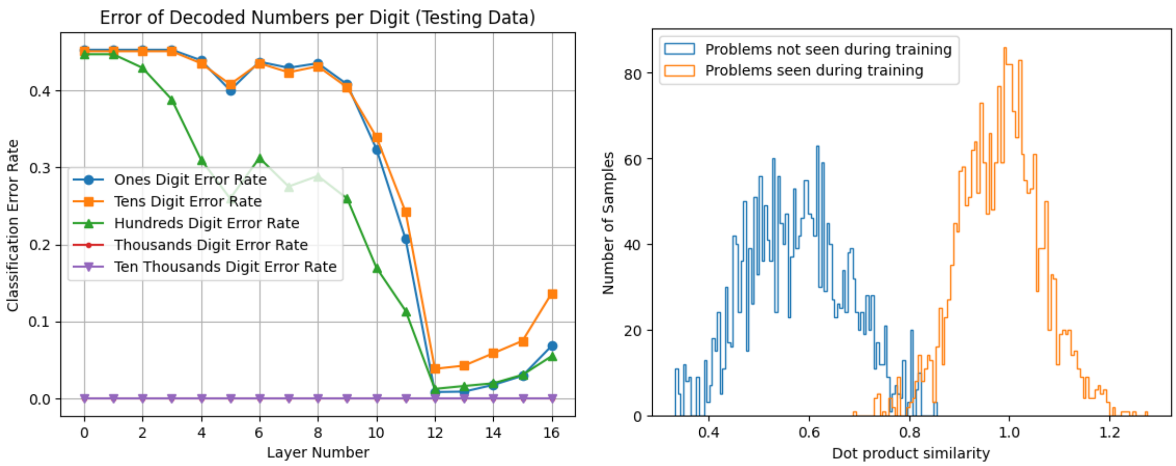


Figure 7.4: Classification error rates for individual digits across layers (Left). Distribution of dot product similarities for problems seen vs. unseen during encoder training (Right).

7.5.2 Task Performance

Table 7.2 compares the performance of our neurosymbolic intervention approach against the standard Llama 3.2 1B model across ten arithmetic tasks. The results demonstrate substantial improvements on most challenging operations:

The neurosymbolic 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 neurosymbolic 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.

7.6 Determining Problem Types and Intervention Thresholds

As discussed in Section 6.1.5, 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 5.3.

For problems seen during training, we expect that $\mathbf{result}$ will be approximately equal to a

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

problem type seen during training, since one of the encoder's purpose is to represent the correct problem type in its neurosymbolic vector output. For problems not seen during training, the expected behaviour is that **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 7.3. 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.

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 7.3: Dot product similarities for problem type queries.

Figure 7.5 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.

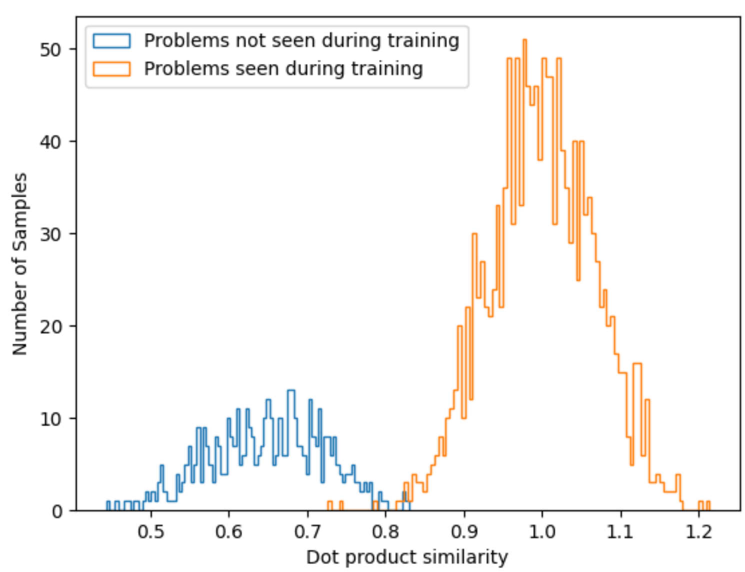


Figure 7.5: Histogram of maximum similarity of queried problem type across all problem types, segregated per training and non-training problems.

7.7 Performance Comparison on Non-Mathematical Problems

As discussed in Section 7.2, 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.

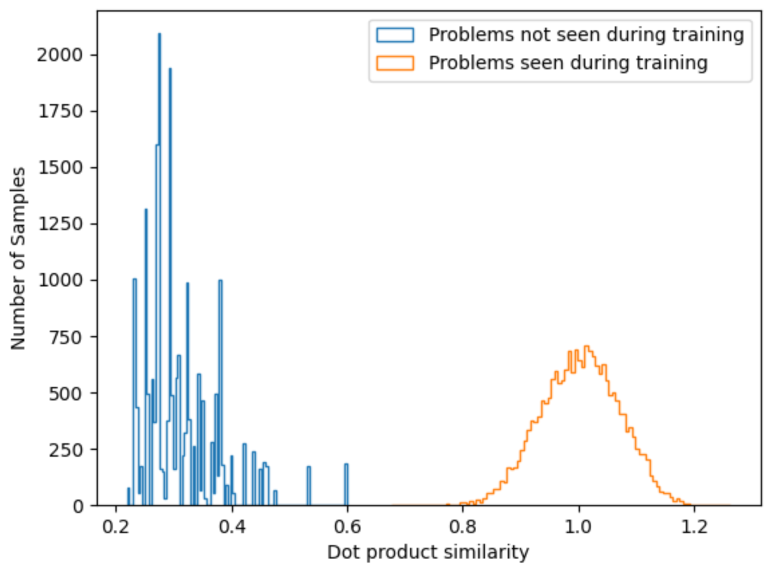


Figure 7.6: Histogram of maximum problem type similarity for training problems vs. non-mathematical queries. None of the non-math queries exceed the 0.8 threshold.

Figure 7.6 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.

7.8 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.

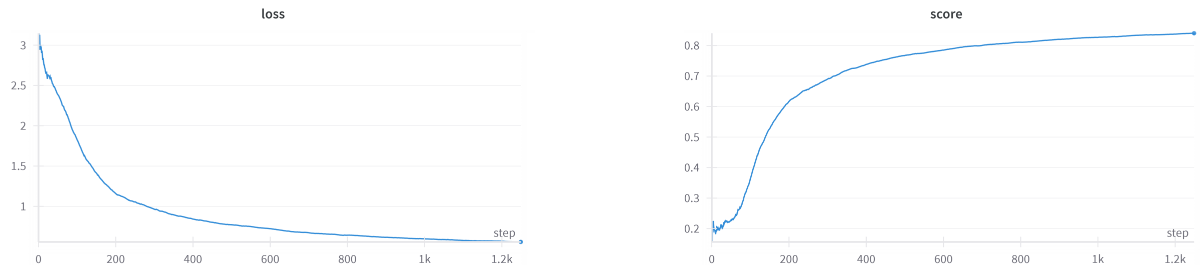
RMS Layer Normalization was tested as an alternative; Table 7.4 shows the 50/50 mix is generally better.

Table 7.4: 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

7.9 Decoder Fine Tuning

As mentioned in Section 6.1.5, the decoder network requires fine tuning to properly enhance LLM performance. Figures 7.7a and 7.7b illustrate that as fine-tuning progresses, both cross-entropy loss decreases and task performance improves. One fine-tuning step is a batch.



(a) Average cross-entropy loss vs. step

(b) Average score vs. epoch

Figure 7.7: Training performance over steps. (a) Loss decreases over time, indicating effective learning. (b) Score increases, confirming improved model accuracy.

Chapter 8

Method Extensions

This section describes our extension to the neurosymbolic LLM framework through the replacement of linear encoders with transformer-based architectures. We first analyze the limitations of linear encoding for VSA generation, then present our transformer encoder design and demonstrate its advantages for processing contextual information.

8.0.1 Limitations of Linear Encoding

The original neurosymbolic approach employs a linear encoder that maps the hidden state of the most recent token to a VSA representation. While effective for the constrained experimental setting, this design presents limitations when dealing with longer input sequences or when optimal encoding performance is required.

The linear encoder operates by transforming the final token's hidden state through a single linear transformation. As input sequences grow longer, the final token's hidden state must encapsulate increasingly complex contextual relationships from all preceding tokens. This places greater demands on the LLM's internal attention mechanism to compress all relevant mathematical problem information into the final position, potentially leading to reduced encoding accuracy as the sequence length increases.

8.0.2 Data Modification

In order to test how both the linear and transformer-based encoders perform on both limited input data sizes and increased input sequence complexity and length, we use a version of the

Symbolic-Math dataset with 1,000 rows of data and with multiple question formats per problem type. For example, question formats for the squared modulo problem type are: "{x}^2 mod {y}", "({x}^2) mod {y}", "What is {x} squared mod {y}?", "Calculate {x}^2 mod {y}.", and "Compute {x} squared modulo {y}." (for complete details on the dataset and types of variable question formats used, see https://github.com/vdhanraj/Symbolic-Math-Dataset).

8.0.3 Transformer Encoder Architecture

To address the linear encoder limitations, we replace it with a transformer-based architecture that can process the hidden states of all tokens in the input sequence. Our transformer encoder provides direct access to the full contextual information available at the selected LLM layer, enabling more robust VSA generation across diverse input formats.

Model Architecture

The transformer encoder consists of four main components that process the hidden states of all tokens in the input sequence. The architecture includes an input projection layer that maps LLM hidden states from dimension d to an internal hidden dimension of 512, followed by four transformer encoder layers with 8 attention heads each and feedforward dimensions of 2048. After transformer processing, the final token's representation is selected and projected to the VSA vector dimension v through an output projection layer. The model uses GELU activation, 0.1 dropout rate, and bfloat16 precision for computational efficiency. Note that the total number of trainable parameters in the transformer encoder is 9.4 million, which is similar to the 8.4 million trainable parameters in the linear encoder.

The transformer encoder processes a sequence of LLM hidden states $\mathbf{X} \in \mathbb{R}^{n \times d}$, where n is the sequence length and d is the LLM hidden dimension. The processing is as follows: input projection to internal dimension, multi-head self-attention and feedforward processing through four transformer layers, final token selection, and output projection to VSA vector space $\mathbf{v} \in \mathbb{R}^{v}$.

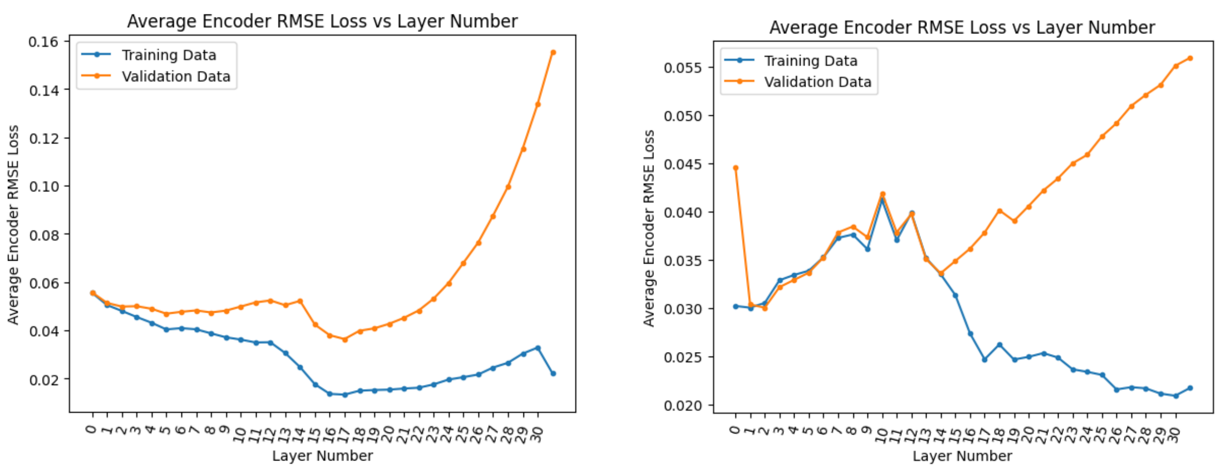
8.0.4 Advantages for VSA Generation

We hypothesize that the transformer-based encoder offers advantages over linear encoding for VSA generation by providing direct access to all token representations within the input sequence rather than relying solely on the final token's hidden state. The self-attention mechanism allows the encoder to dynamically weight the relevance of different tokens based on their contribution to the mathematical problem structure, enabling more comprehensive information integration.

The multi-layer transformer architecture, with its combination of attention mechanisms and feedforward networks, provides greater representational capacity compared to a single linear transformation. This enhanced capacity enables the model to learn more complex mappings from LLM hidden states to VSA representations, potentially leading to improved encoding accuracy across different layer depths and input formats.

8.1 Results

8.1.1 Performance versus Intervention Layer



(a) Linear Encoder

(b) Transformer Encoder

Figure 8.1: Encoder RMSE loss versus intervention layer for linear encoder (left) and transformer encoder (right). While both approaches result in overfitting in deeper layers of the LLM, the validation loss for the transformer-based encoder is much smaller at earlier layers (note that the two graphs have different y-axis scales) allowing for successful conversion of hidden states to VSAs by intervening earlier in the LLMs forward pass.

The choice of intervention layer significantly impacts encoder performance and generalization capabilities. Figure 8.1a shows the RMSE loss across all LLM layers for the linear encoder, while Figure 8.1b presents the corresponding results for the transformer encoder.

The linear encoder exhibits severe overfitting beyond layer 15-17. Training loss decreases steadily to approximately 0.014 at layer 17, while validation loss initially follows this trend but diverges after layer 17, reaching 0.16 by layer 30.

Note that this overfitting pattern only occurs when the training dataset size is limited 1,000 training examples. When trained with 10,000 training examples, as done in Chapter 6, the encoders at each layer do not overfit, as shown in Section 8.4. Additionally, this work introduces varied question formats whereas the original paper did not, which may increase the difficulty of encoding accurate VSA representations.

Similar to the linear encoder, the transformer encoder also overfits in later layers, as shown in Figure 8.1b. Unlike the linear encoder, however, validation RMSE loss of the transformer-based encoder is much lower at earlier layers. This means that the transformer based encoder can obtain accurate VSA representations on problems it has not seen during training by choosing the intervention layer to be earlier in the LLM's forward pass.

Using the layers at which each approach achieves a minimum validation RMSE score, we select the intervention layer for the linear encoder to be 17 and the intervention layer for the transformer-based encoder to be 2. Selecting a fixed intervention layer for each approach is necessary in order to run the neurosymbolic system end-to-end in order to determine how the symbolic intervention improves the LLM's ability to output the correct answers.

8.1.2 End-to-End Performance

Table 8.1 presents the performance comparison across all problem types. The transformer encoder consistently matches or outperforms both the linear encoder and standard LLM across all problem types, achieving 93.1% average accuracy and 0.341 average cross-entropy loss compared to 53.6% accuracy and 2.389 loss for the linear encoder and 30.7% accuracy and 4.861 loss for the standard LLM. This corresponds to the transformer based encoder having on average a 73.7% higher accuracy and a 85.7% lower loss compared to the linear encoder, and a 203.3% higher accuracy and 92.9% lower loss compared to the standard LLM. In contrast, the linear encoder has on average a 74.6% higher accuracy and 50.9% lower loss than the standard LLM.

The substantial performance improvement of the transformer encoder over the linear encoder is particularly noteworthy given that both approaches use identical decoder architectures and symbolic algorithms. This suggests that encoding accuracy plays a pivotal role in overall neurosymbolic system performance, as the primary difference between the two approaches lies in the encoder's ability to generate accurate VSA representations from LLM hidden states.

Both neurosymbolic approaches demonstrate strong performance on unseen problem types. On addition problems, the transformer encoder matches the standard LLM's perfect performance

(100.0% accuracy), while the linear encoder achieves 95.7%. For integer division, the transformer encoder outperforms both the linear encoder (66.7% vs 50.0%) and standard LLM (52.8%), indicating effective generalization to problem types not encountered during training. The lower performance of the linear encoder on these problem types indicates the neurosymbolic system is incorrectly engaging for problem types not seen during training, which happens when the system cannot accurately determine the problem type asked in the prompt. The transformer-based encoder avoids this problem by more accurately encoding the correct problem type, leading to intervention only on problem types it has a symbolic algorithm for.

The linear encoder achieves higher accuracy than the standard LLM on all trained problem types except square modulo, where both approaches achieve 8.3% accuracy. This exception reflects the poor encoding accuracy of the linear encoder for this particular problem type, as shown in Section 8.2.

Table 8.1: Performance comparison between transformer-based and linear encoders across problem types. Scores represent accuracy (%) and losses represent cross-entropy values. Note that Addition and Integer Division problem types are not seen during training

Problem Type	**Transformer Encoder**		**Linear Encoder**		**Standard LLM**	
	Score (% ↑)	Loss (↓)	Score (% ↑)	Loss (↓)	Score (% ↑)	Loss (↓)
Multiplication	**100.0**	**0.032**	23.8	3.942	4.8	10.196
Modulo	**87.0**	**1.258**	69.6	1.458	50.0	3.644
GCD	**100.0**	**0.007**	100.0	0.153	73.1	1.131
LCM	**100.0**	**0.051**	24.1	4.201	0.0	6.683
Square Mod	**100.0**	**0.031**	8.3	5.233	8.3	4.973
Bitwise AND	**100.0**	**0.017**	43.8	3.311	3.1	6.064
Bitwise XOR	**100.0**	**0.050**	75.0	1.954	7.5	7.409
Bitwise OR	**76.9**	**1.851**	46.2	3.161	7.7	8.318
Addition	**100.0**	**0.000**	95.7	0.188	**100.0**	**0.000**
Division	**66.7**	**0.108**	50.0	0.186	52.8	0.192
Average	***93.1***	***0.341***	*53.6*	*2.389*	*30.7*	*4.861*

8.1.3 Encoder Accuracy

Figure 8.2a and figure 8.2b show the encoder average number of digits incorrectly represented per layer for the linear encoder and transformer-based encoder, respectively. Since 3 digit numbers are used in this problem, the maximum number of incorrectly encoded digits is 3, and the minimum

(a) Linear Encoder

(b) Transformer Encoder

Figure 8.2: Average number of incorrectly decoded digits versus intervention layer for linear encoder (left) and transformer encoder (right). The linear encoder achieves a minimum error rate around layers 17 and 19 of 0.5 digits out of 3, whereas the transformer-based encoder has a minimum error rate around layer 1 and 2 of 0.05 digits out of 3.

number is 0. At their intervention layers (layer 17 for the linear encoder and layer 2 for the transformer encoder), the linear encoder achieves an error rate of 0.59 digits out of 3, and the transformer encoder achieves an error rate of 0.032 digits out of 3. This substantial difference in the encoding error rate helps explain why the transformer based model is able to more significantly boost the performance of the base LLM relative to the linear encoder method.

8.2 Encoding Accuracy of Different Problem Types

The results in table 8.1 show that the linear encoder approach outperforms the standard LLM on most problem types, with the exception of the square mod, addition, and integer division problem types. In order to explain this phenomena, we can investigate the linear encoder's average encoding accuracy per problem type, as shown in Figure 8.3. This plot shows that the encoding accuracies for the aforementioned problem types the linear encoder performed the worst on, the encoding accuracies are all on average lower than that of the other problem types.

In contrast the transformer encoder's average encoding accuracy per problem type, shown in Figure 8.4, indicate that the average score for square mod, addition, and integer division problem types is not significantly lower than that of the other problem types. The average encoder accuracy for all problem types, including those not seen during training such as addition and integer division, is above 90%, which may partially explain why the transformer encoder was significantly more accurate than the linear encoder.

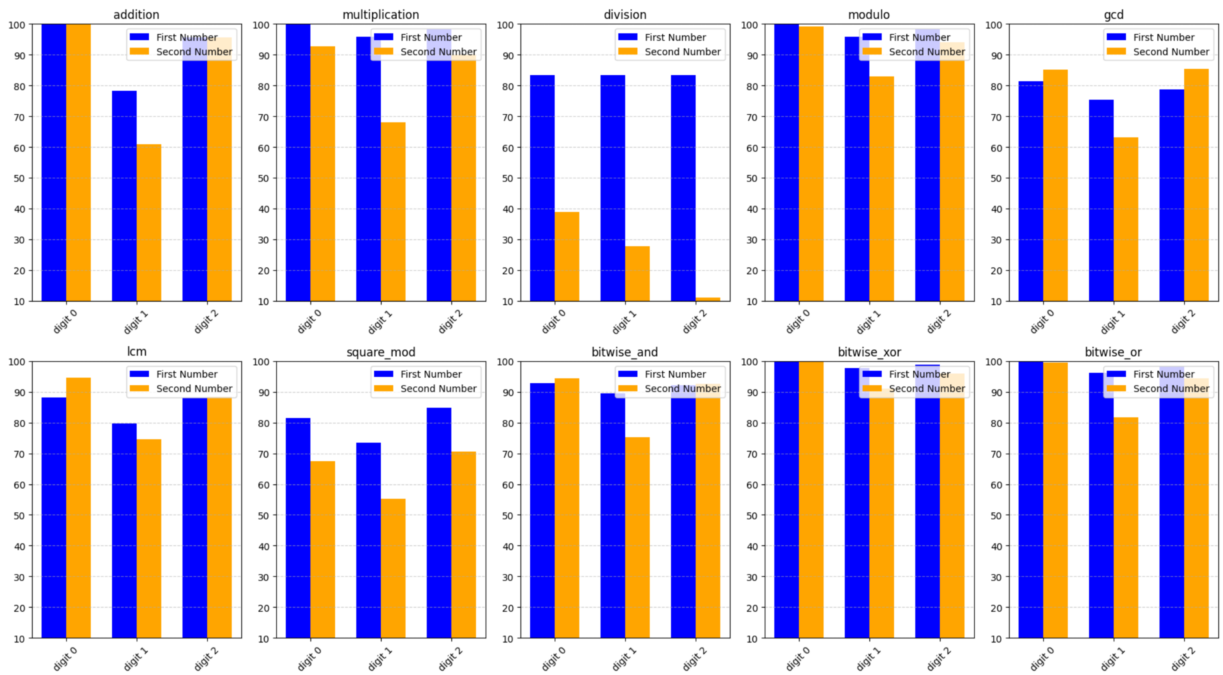


Figure 8.3: Encoding accuracy per digit of each problem type for the linear encoder. Note that (integer) division and addition were not problem types the encoder was presented during training.

In addition to the encoding accuracy, there may be other reasons why the performance of both the linear encoder and transformer encoder are not 100%. For example, the transformer encoder has an accuracy over 99% for the modulo problem type, but the end-to-end performance for this problem type is only 87.0%.

One potential reason for this phenomena is that for the modulo problem type, the algorithm the LLM is running in its forward pass is difficult to steer just by changing the hidden state of the most recent token at a particular layer. In the case of the transformer encoder, it is attempting to change the most recent token's hidden state at layer 2. One way to analyze the algorithm an LLM is running is by creating an attribution graph, which is derived from an LLM by creating an attribution graph, trained to predict the hidden state of future layers of an LLM using sparse

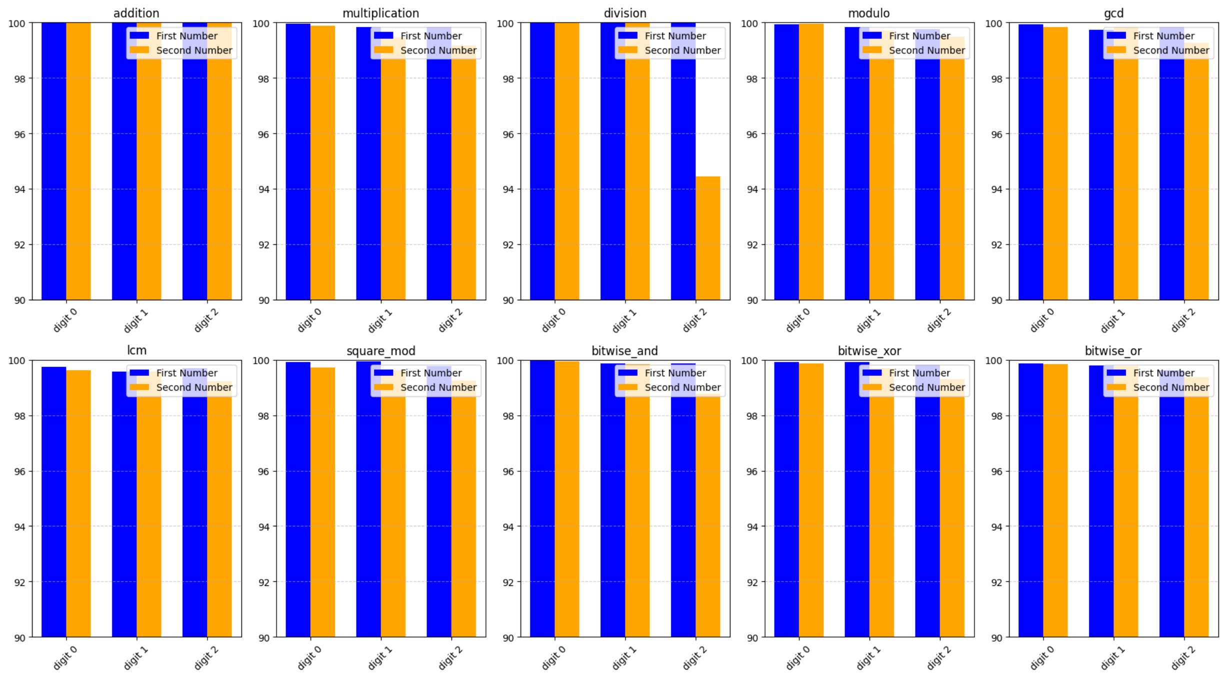


Figure 8.4: Encoding accuracy per digit of each problem type for the transformer encoder. Note that (integer) division and addition were not problem types the encoder was presented during training.

activations in earlier layers [82]. These attribution graphs show which sparse activations, or concepts, light up at various points in an LLMs computation process. For example, during addition, in earlier layers concepts related to the ones and tens digit of the earlier number may light up, and in later layers, concepts related to the sum of those two numbers may light up [82].

Since different problem types have different activation circuits (i.e., diagrams that show which activations feed into the hidden states of future layers and tokens), it is possible that changing the hidden state at a single layer for every problem type will not be completely effective in raising the end-to-end accuracy to 100%. One possible solution would be to place a decoder layer at various layers in the LLM (each using the same solution VSA vector as input), in order to modify the hidden state of the model at different points of its forward pass. Constructing these graphs and implementing such a solution is outside the scope of this current project, however, since it would require extra compute to train these additional components.

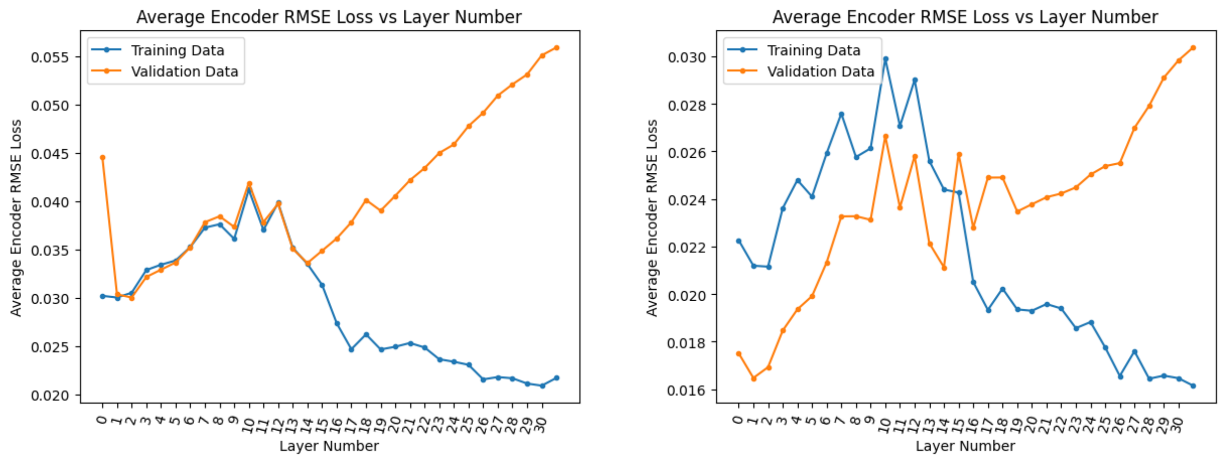
8.3 Validation Loss of Transformer Encoder with 2,000 Rows of Data

Since Figure 8.2b showed that the transformer based approach overfit to the training data beyond a particular layer number, it is important to determine whether this behaviour is reduced as the model is exposed to more training data. To test this, we train a transformer-based encoder on 2,000 rows of data, to see if the overfitting is reduced. Figure 8.5 shows the encoder validation and training loss of both the 1,000 and 2,000 row trained models. Note that the validation loss of the model trained on 2,000 rows of data is significantly less than that of the model trained on 1,000 rows of data, indicating that the overfitting behaviour reduces as the training data size increase.

As mentioned in the limitations section, the main limitation of this work was not being able to train the transformer-based encoder on more data. In Chapter 6, we use 10,000 rows of training data instead of 1,000, but due to the increase size of the transformer encoder dataset, we were forced to reduce our dataset size in the approach of this chapter. The results in this section show that increasing the dataset size of the transformer encoder reduces the effect of overfitting in the encoder, which suggests that a transformer-based encoder trained on large amounts of data would be able to accurately extract VSA representations from intermediate layers in addition to early layers.

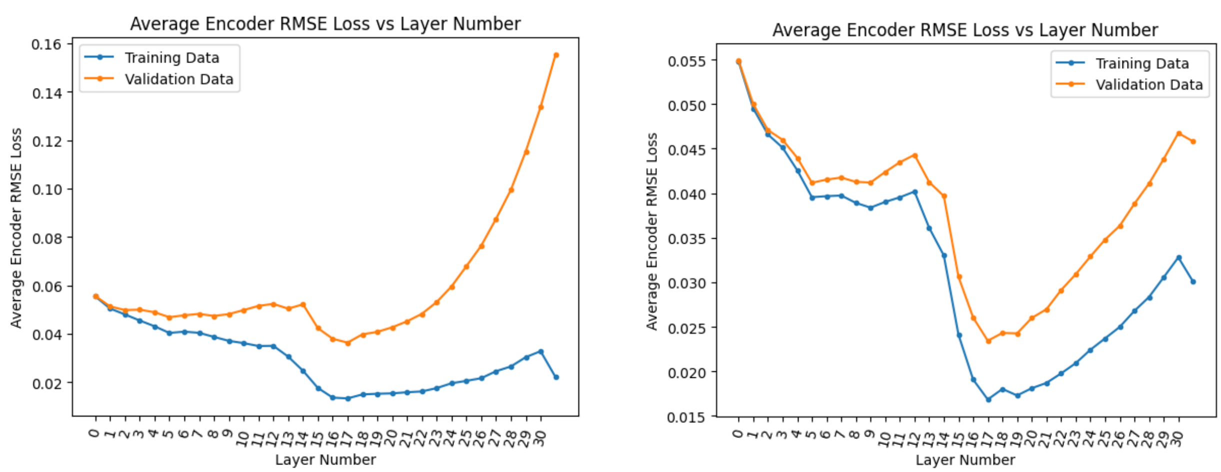
8.4 Validation Loss of Linear Encoder with 10,000 Rows of Data

The results in section 8.1.1 indicate that the linear encoder model tended to drastically overfit when trained on 1,000 rows of data. In Chapter 6, we also used a linear encoder, but it was trained with 10,000 rows of data. Figure 8.6 shows the impact of increasing the training dataset size to 10,000 rows from the 1,000 rows used in this study. Here, it is clear that the overfitting effect in later layers is much less significant when trained with 10,000 rows of data than when trained with only 1,000. This result indicates why in Chapter 6 we were able to accurately generate VSA representations from layer 16 of the LLM, since we used the larger 10,000 prompts to train their encoder.



(a) 1,000 rows of data (b) 2,000 rows of data

Figure 8.5: Average encoder loss per layer of the transformer encoder trained with 1,000 rows of data (left) and with 2,000 rows of data (right). Note that the scale on the graph trained with 2,000 rows of data goes to 0.030, whereas the scale of the graph trained with 1,000 rows of data goes to 0.055, indicating a higher degree of overfitting for the model trained with less data.



(a) 1,000 rows of data (b) 10,000 rows of data

Figure 8.6: Average encoder loss per layer of the linear encoder trained with 1,000 rows of data (left) and with 10,000 rows of data (right).

Chapter 9

Applications of Neurosymbolic Integration

The goal of this chapter is to demonstrate how the neurosymbolic integration method described in Chapter 6 can be extended to help LLMs in a variety of domains in which they struggle. The Visual Question Answering section shows how VSA integration into LLMs can provide an avenue in which LLMs can more reliably perform visual reasoning by querying a semantically segmented neurosymbolic representation of the image. The MiniGrid section shows how efficiently structured VSA representations (i.e., spatial semantic pointers) can be used by LLMs to navigate simple RL environments according to some natural language mission statement. Finally, we discuss an important issue LLMs have with maintaining reliably performance on long context windows, and how using the VSA methods discussed in this thesis can help address these problems.

Note that while we include results to support our claims about the neurosymbolic integration method improving the weaknesses of LLMs in these tasks, the results described in this chapter are preliminary, since they lack thorough hyperparameter testing and robust comparison to other methods. For example the Visual Question Answering approach lacks comparisons to other approaches to solving this problem, such as using multimodal LLMs. The projects described in this section are instead meant to demonstrate the potential of neurosymbolic integration in important open problems in language modeling, and are currently planned to be further researched in future work.

9.1 Visual Question Answering

Visual Question Answering (VQA) represents a fundamental challenge at the intersection of computer vision and natural language processing, requiring models to understand both visual content and natural language questions to generate accurate answers. Traditional approaches to VQA relied on separate vision and language models with carefully engineered fusion mechanisms, but the emergence of multimodal large language models has fundamentally transformed this landscape [4, 62].

The evolution toward multimodal LLMs began with models like CLIP [110] and ALIGN [63], which learned joint vision-language representations through contrastive learning on large-scale image-text pairs. These models demonstrated that visual and textual information could be embedded in a shared semantic space, enabling zero-shot transfer to various vision-language tasks. However, while these approaches excelled at retrieval and classification, they lacked the generative capabilities necessary for open-ended question answering.

Modern multimodal LLMs address VQA through various architectural strategies for integrating visual information into language models. Models like Flamingo [2] introduce cross-attention layers that allow the language model to attend to visual features extracted by a frozen vision encoder, preserving the pretrained language capabilities while adding visual understanding. BLIP-2 [76] employs a lightweight Querying Transformer (Q-Former) that learns to extract the most relevant visual features for the language model, reducing computational costs while maintaining performance. LLaVA [83] takes a simpler approach, using a linear projection layer to map visual features into the language model's embedding space, demonstrating that effective vision-language alignment can be achieved with minimal architectural modifications.

Despite these advances, multimodal LLMs face several persistent challenges in VQA tasks. The discretization problem arises from the mismatch between continuous visual features and discrete text tokens, making it difficult to preserve fine-grained visual information through the language model's processing [138]. Models often struggle with spatial reasoning, counting objects, and understanding compositional relationships between visual elements [149]. Furthermore, the computational cost of processing high-resolution images through transformer architectures limits practical deployment, leading to trade-offs between visual detail and efficiency.

Recent work has explored alternative approaches to bridge vision and language without requiring full multimodal pretraining. Frozen [129] demonstrates that a frozen language model can perform VQA when visual information is encoded as a sequence of continuous tokens prepended to the text input. Similarly, models like PaLM-E [32] embed visual observations directly into the language model's input sequence, treating images as another form of structured data. These approaches suggest that language models already possess latent capabilities for processing structured non-linguistic information when appropriately formatted.

The success of these methods motivates the exploration of neurosymbolic approaches to VQA, where visual information is encoded into structured symbolic representations that language models can process more naturally. By converting visual features into VSA representations, we can potentially leverage the compositional structure of VSAs to more reliably represent spatial relationships, object properties, and scene semantics in a format that aligns with the language model's existing reasoning mechanisms. This approach could address some fundamental limitations of current multimodal models while enabling language-only models to perform visual reasoning tasks without architectural modifications.

9.1.1 Methodology

Our methodology consists of 2 main stages. First, an image is segmented and labeled using various pretrained models, effectively converting the image into a VSA that represents the segmented objects in the image, as shown in Figure 9.1. Then, this VSA representation is used with the language model to answer simple questions about the image. The diagram in Figure 9.2 details how this information is integrated with forward pass of the LLM. Before we discuss the details of how the VSA representation of the image is generated and how it is used to steer the LLMs forward pass, we will describe the dataset and pretrained models used for this project.

Pretrained Models and Datasets

In this work, we use 4 pretrained models. Firstly, to segment images into objects, we use the Mask2Former model [17], which is a state-of-the-art image segmentation transformer model. Mask2Former uses a transformer decoder architecture with masked attention to perform unified segmentation across semantic, instance, and panoptic segmentation tasks. The model employs learnable object queries that are refined through multiple decoder layers, with each query learning to segment a specific object or region in the image.

Since this model was trained on the COCO (Common Objects in Context) dataset [80], we chose to use the COCO dataset as images to perform VQA on. COCO is a large-scale dataset containing over 330,000 images with 80 object categories, providing rich annotations including object segmentation masks, bounding boxes, and captions. The dataset emphasizes natural images containing common objects in their typical contexts, making it ideal for evaluating real-world visual understanding capabilities.

The output of the image segmentation model is, for each detected object in each COCO image, a polygon mask describing where the object was in the image and what shape it is, and a text label describing what type of object was in that location. In order to convert this image into a VSA representation, we must first create vector embeddings that capture both the semantic meaning of the object represented by a particular mask, as well as the spatial meaning of where the object is and what type of shape it represents.

In this work, we use the ViT-MAE pretrained model [53], which is an image autoencoder model designed to extract meaningful representations of images. The ViT-MAE-Large model uses a Vision Transformer architecture with 24 layers and 1024-dimensional hidden states, pretrained on ImageNet-1K through masked autoencoding where 75% of image patches are masked and reconstructed. This self-supervised pretraining enables the model to learn rich visual representations that capture both spatial relationships and semantic content, making it suitable for encoding object masks into vector representations. While ViT-MAE provides reasonable embeddings for our masked objects, a custom model trained specifically on segmented objects rather than full images would likely produce more specialized representations better suited for this task.

In addition to creating embeddings for the masks of each object in our image, we also create embeddings that encapsulate the semantic meaning of each word. We use RoBERTa, an encoder only language model to generate these embeddings. RoBERTa is a robustly optimized BERT model with 125 million parameters, trained on 160GB of text including BookCorpus, English Wikipedia, and additional web text [87]. The model uses a transformer encoder architecture with 12 layers and was trained with dynamic masking and larger batch sizes compared to BERT. Since RoBERTa was trained on diverse web text, it has encountered the COCO object category labels (such as "person," "car," "dog") during pretraining, enabling it to generate meaningful semantic embeddings for these concepts.

In addition to these pretrained models which are responsible for generating the VSA of a scene, we use the same pretrained decoder language model as our previous experiments, the Llama 3.1 8B model [127]. This model features 8 billion parameters across 32 transformer layers with a hidden dimension of 4096, trained on over 15 trillion tokens of text data. Notably, this is not a multimodal language model, which means it has no visual processing capabilities. The goal of our approach is to provide this pretrained language only LLM the capability to perform visual reasoning with neurosymbolic integration.

As mentioned previously, we use the COCO dataset in order to create questions that we train the LLM to answer. In this work we prompt our system with relatively simple questions that probe whether the LLM can query the VSA describing the image for relevant information to answer the prompt. The questions we prompt our model with ask it whether a certain type of object is present in the image. For example, we may ask the model whether a tree exists in an image.

The correct response to a given question will be either 'yes' or 'no'. Importantly, in our experiments, the correct response is not determined by whether or not the object actually exists in the image, but whether or not the object exists in the set of segmented objects identified by the pretrained segmentation model. This design choice was made because our goal was to train the LLM to accurately query and interpret data provided in the VSA representation of the image, not to train models that could accurately provide those VSA representations. If the correct response was determined by the object actually existing in the image, then errors due to the segmentation model not correctly encoding certain objects in the image would cause the neurosymbolic intervention mechanism to be penalized for errors made by the pretrained segmentation model. This would result in a more noisy training process, leading to lower performance. Future work includes creating a more realistic label for the data, based on whether certain objects appeared in the original image, rather than if they appeared in the output of the segmentation model.

VSA Generation

Once the pretrained segmentation model has segmented each image into different objects (i.e., into masks and labels per segmented object) and the pretrained image and text embedding models have generated vector embeddings for each of the segmented objects, this information is combined to form a VSA vector representing that image. This is done in a similar approach as shown in Equation 5.3, where randomly generated slot-filler VSA vectors are created to bind to different properties about an object. Specifically, we create slots for both the object mask and object label and fill those slots with the embedding vectors created from our pretrained models, allowing us to incorporate these pieces of information about the objects in an easy-to-query format. An example showing how this process works is shown in Figure 9.1.

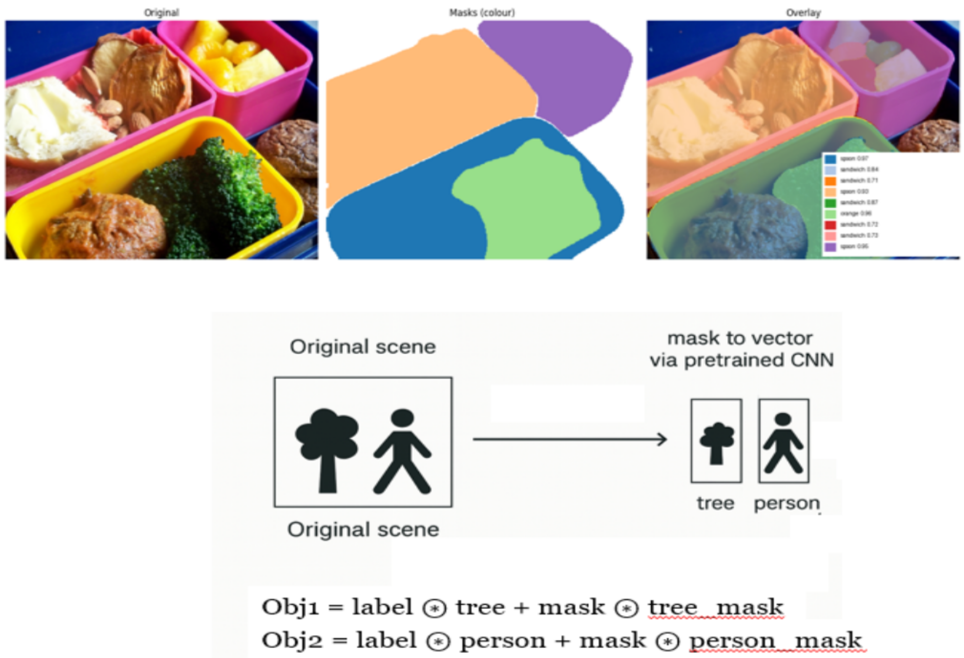


Figure 9.1: Example of how the VQA uses various pretrained models to create a VSA representation of an image in order to answer questions about it. The first step is to segment an image using a pretrained segmentation model. Then, for each object, pretrained text and image models are used to generate embeddings for the object label and object mask, respectively. These embedding vectors are used to create VSA vectors that represent each of the objects, shown as Obj_1 and Obj_2 in the Figure.

There are multiple possible ways of using this semantic information about different objects in an image to guide an LLM in visual question answering. Our initial approach involved creating a single compositional VSA vector representing the entire image, which could be queried for information about different objects in that image. This is achieved by defining the entire image VSA vector as:

$$\begin{aligned}\mathbf{VSA}_{\text{image}} &= \sum_{i=1}^{n} \mathbf{Object\_Tag}_i \circledast \mathbf{Obj}_i \\ &= \sum_{i=1}^{n} \mathbf{Object\_Tag}_i \circledast \left(\mathbf{label} \circledast \mathbf{object\_label}_i + \mathbf{mask} \circledast \mathbf{object\_mask}_i\right) \qquad (9.1)\end{aligned}$$

where $\mathbf{Object\_Tag}_i$, **label**, and **mask** are randomly created VSA vectors, and i goes from 1 to n, where n is dynamically determined by the number of segmentation masks generated by the pretrained segmentation model. Note that each $\mathbf{object\_label}_i$ and $\mathbf{object\_mask}_i$ are embedding vectors generated from the pretrained language and vision models (i.e., Roberta and ViT-MAE), respectively.

An alternate approach is to use the list of objects without summing them up. This means that instead of having a single VSA representation of the entire image that is created by bundling together the VSA vectors representing individual objects, we modify our VSA integration approach so that it can extract information from an arbitrary number of different object VSA vectors. Since our empirical testing found that the approach of presenting information about the image in separate object VSA vectors had higher performance compared to presenting the image information in a single image VSA vector, the following architecture section will focus on integrating information provided by multiple VSA vectors into the LLM's forward pass.

Alternate Approaches

Previous work has explored alternative approaches for creating VSA representations of images, notably SSPictr [103], which uses a pretrained segmentation model to identify different objects in images, and then creates VSA representations of those objects by sampling pixels within the object masks. This method sums the VSA representation of each pixel sampled from an object mask and binds them with an identifier for that segmented object in order to store information about the location of that object in the image.

In contrast, our method passes the object mask through the aforementioned image encoder in order to represent information about the shape, size, and location of each object, and passes the segmented object label through the aforementioned text encoder in order to represent the meaning of that object label. The advantage of our approach is that our representation for each object is formed from the combination of only two vector embeddings (the mask and label embeddings), whereas the SSPictr approach represents each object as a much larger sum of individual VSA vectors representing the location of sampled pixels within an image.

As shown in Chapter 5, querying information from a VSA is an approximate operation, where the amount of noise associated with the querying process increases with the size of the bundle (i.e., the number of vectors summed together). Because of this, our approach, which sums together significantly fewer vectors relative to other approaches like SSPictr, will create VSA representations of images that are much easier to query and manipulate without accruing significant amounts of noise.

Architecture

Similar to our previous work, our goal is to steer the LLM's forward pass using a VSA representation providing key information to help the LLM answer correctly. Unlike the neurosymbolic intervention system described in Chapter 6, however, our VSA is not derived from the hidden state of the LLM, since our LLM has no ability to generate semantically relevant information about the image. Instead, this VSA representation of the image is obtained from the aforementioned procedure involving using pretrained models to segment an image and create embeddings for the shapes and labels of the segmented objects.

Our approach instead involves creating an architecture that is capable of dynamically querying the information in the VSA representations in order to answer the question in the prompt. This is achieved by the architecture in Figure 9.2. In this Figure, the encoder and decoder networks are linear networks (just like in Chapter 6), and the encoder representation is further projected into a query and similarity vector using two additional linear networks, denoted as $\mathbf{q}$ and $\mathbf{s}$ in the Figure.

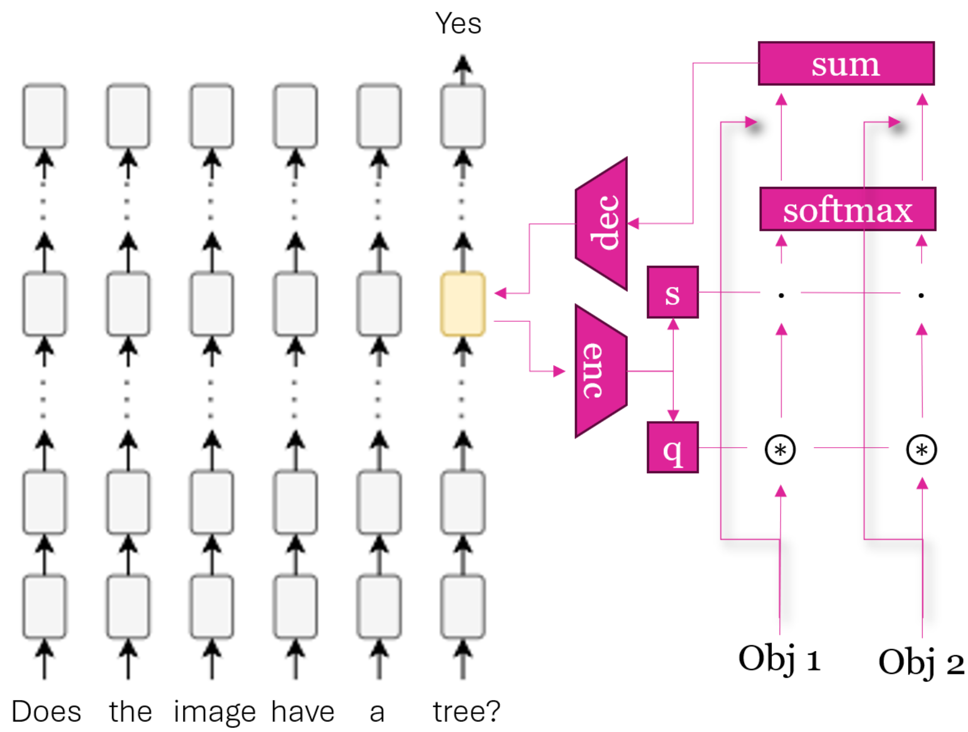


Figure 9.2: Diagram showing how an image VSA is queried in order to perform visual question answering.

This architecture enables the encoder and **q** matrices to convert an LLM hidden state representation into a query VSA vector, which is bound with each object VSA vector. This means that if we select the appropriate layer of the LLM such that the hidden state at that layer contains information about which object is being prompted, it will be possible for the encoder and **q** matrices to convert this hidden state representation into the VSA corresponding to the inverse of the slot being queried. For example, if the prompt asked if there is a tree in the image, the ideal value of the query vector would be $\mathbf{label}^{-1}$, since we want to query each object VSA for its label.

After each object vector is bound with the query vector, we take the dot product of this result with a similarity vector, which, similar to the query vector, is generated dynamically by passing the hidden state through the encoder, and then applying the **s** matrix projection. Ideally, this VSA vector will represent the object we wish to query from our network. For example, if the prompt asked if there is a tree in the image, the ideal value of the similarity vector would be **tree**, since we want to check if each of the object vectors label is a tree.

The result of the previous step provides a set of scalar similarity scores per object. Our next step applies a softmax to scale these similarity scores to be between 0 and 1, effectively normalizing them. Note that this step was mainly employed for numerical stability and to allow gradients to flow effectively to the encoder, **s**, and **q** matrices. A consequence of using the softmax to scale the similarities is that it becomes impossible for the normalized similarities to be 0 if the object is not in the image. Instead, assuming that all the un-normalized similarity scores would be close to 0, the normalized similarity scores would be roughly equal to $\frac{1}{\mathbf{n\_objects}}$, where **n_objects** is the number of objects detected by the segmentation model. Despite this undesirable property, we use the softmax to normalize the score in order to facilitate stable training of the encoder, **s**, and **q** matrices. Our empirical results show that using the softmax to normalize the scores is a critical step in order for the LLM to obtain high accuracies when performing VQA.

Once normalized scores are obtained per object, these scores are used to create a weighted sum of their corresponding object VSA vectors. The motivation for this is to generate a bundle of VSA vector objects, where objects not in the query are filtered out. This filtered vector is then fed into an decoder network, which converts this information into a format that is useful to the LLM (similar to the decoder network used in Chapter 6).

Note that while we prompt our system with questions about whether a certain object is present in an image, our same architecture would theoretically also be capable of answering questions like 'Is there an object in the top right corner of the image', if the **q** projection learns to generate the $\mathbf{mask}^{-1}$ VSA, and if the **s** projection learns to generate a VSA representing the mask embedding of an object in the top right corner of the image. We did not conduct these experiments since generating labels for these questions would be more complicated than for the type of questions we used.

Training

The training process for this system is similar to that of Chapter 6, except this system's encoder, decoder, **s**, and **q** matrices do not include a pretraining phase. Instead, they are only trained in the context of the entire LLM (i.e., in the same manner the decoder was fine tuned in Chapter 6). This was a design choice meant to make the training process simpler, but requires further experimentation to validate whether including a pretraining phase improves the system performance.

9.1.2 Results

In order to measure the efficacy of our system, we train our model over 100 epochs (with 1000 training images, 50 validation images, and 200 test images), and record the percentage of questions correctly answered. We use multi-shot prompting to force our model to answer either 'Yes' or 'No' to our questions (we only provide text prompts, we don't actually provide image VSA vectors or provide neurosymbolic intervention in the multi-shot prompts). Note that the model will achieve a 50% accuracy by randomly guessing, which we use as a baseline comparison for understanding how well our neurosymbolic system works.

The validation performance is shown in Figure 9.3, which shows that the system is able to learn how to query the object VSA vectors for information relevant to the question they were prompted with, and use the result to steer their output towards the correct solution. After training for 100 epochs, our model obtained a 92% accuracy on the test dataset, with a cross entropy loss of 0.2633. Compared to random guessing, which would obtain an accuracy of 50% and a cross entropy loss of $-\log 0.5 = 0.69314$ (assuming that the model assigns 50% probability to both the yes and no tokens), our model obtains a 84% higher accuracy and a 62.01% lower loss.

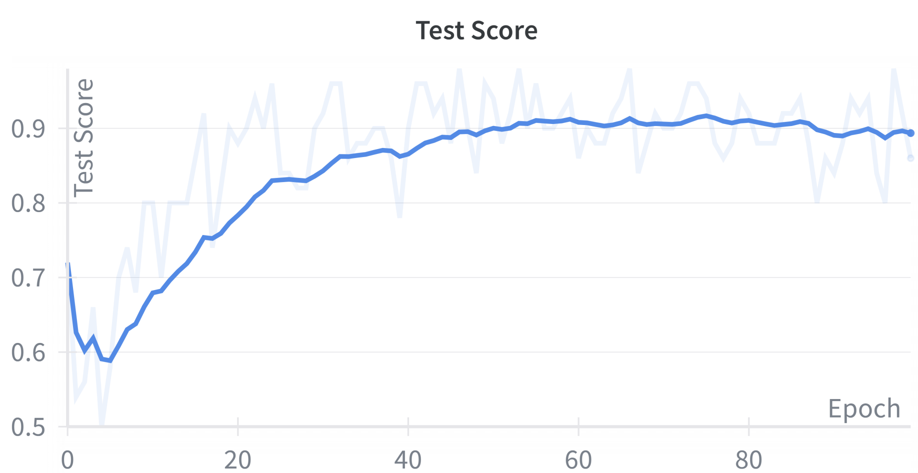


Figure 9.3: Average Reward per validation example of the LLM on the VQA task, showing the neurosymbolic system is able to effectively query the VSA representation of an image in order to perform simple visual reasoning on it.

9.2 Environment Representation

Another application of our neurosymbolic intervention method is to create RL agents that can efficiently explore an environment. Specifically, we are interested in RL problems that involve natural language components and symbolic components (e.g., grids). Fundamentally, we want to create a system that can manipulate, create, and use symbolic representations of environments in order to receive and provide natural language instructions.

The motivation is the fact that for many problems, creating abstract symbolic representations may better facilitate manipulation of these representations compared to creating representations in the space of natural language tokens. As mentioned in Chapter 3, many of the approaches researchers take towards improving the reasoning of LLMs are limited to the token space (e.g., CoT, LRMs, GRPO, etc.), which limits the type of problems LLMs can effectively reason over. Certain problems that involve visual or spatial reasoning may be solved by LLMs more optimally if they could manipulate visual or spatial symbols, rather than tokens.

One example of this type of problem is the MiniGrid task, where an RL agent must navigate an environment and perform actions in order to accomplish a mission statement (which is outlined in natural language) [18]. An example of a MiniGrid task is provided in Figure 9.4.

Our objective is to provide an LLM with the mission statement and to use a VSA representation of the agent's visible environment to get the agent to make a series of decisions that lead to it accomplishing its mission statement in the fewest number of moves.

9.2.1 Methodology

For this work, we use a smaller pretrained Llama 3 model in order to facilitate the more complicated training method we employ. Specifically, we use the pretrained Llama 3.2 1B model as our LLM [50].

In order to generate our VSA representation, we use the following github repository originally created by Nicole Dumont [33]: `https://github.com/vdhanraj/vsa-gym-wrapper/tree/llm_agent`. This library creates a VSA representation of an agent's visible environment using a representation called Spatial Semantic Pointers (SSPs) [71], also sometimes referred to as Fractional Power Encoding (FPE) [43]. These representations are an extension of HRRs, allowing for granular spatial information to be encoded by creating mechanisms that allow for repeated binding (like those used to represent numbers in Chapter 6) to be performed a non-integer number of times. This allows our VSA vector to represent objects in an environment which may be located a scalar distance away from the agent. For more details about how the VSA representing the RL environment is generated, refer to [33].

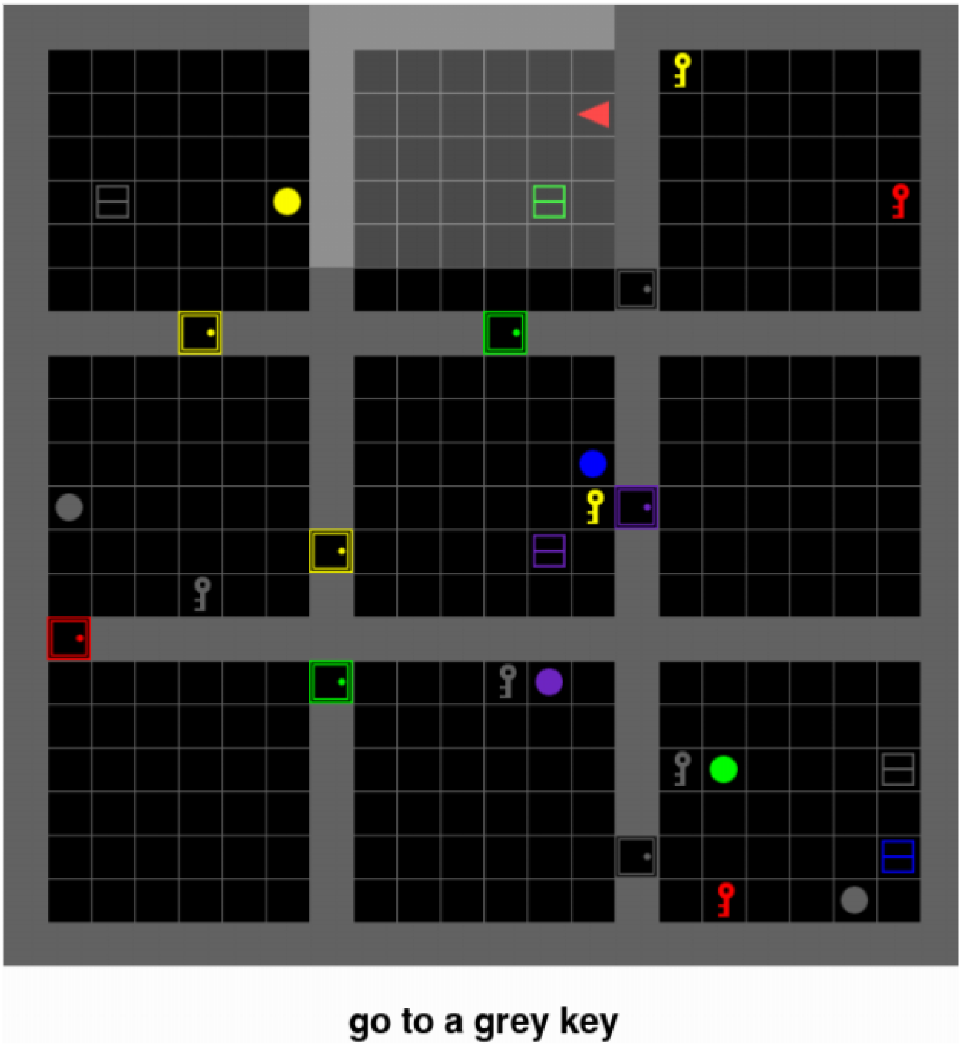


Figure 9.4: Example of a game, where the agent must accomplish the natural language mission statement provided. Note that the illuminated area around the agent (the red triangle) represents the agent's visible field of view.

Experiment Setup

For our experiment, we chose to start with one of the simplest MiniGrid problems to determine if our neurosymbolic integration method would be effective in providing the LLM agent information about the objects in the environment it was attempting to navigate through. Specifically, we chose the 'MiniGrid-Empty-5x5-v0' RL task, which involves placing an agent in a 5x5 grid with no objects other than a green square [18]. The mission statement of the problem is to navigate to the green square.

The actions the agent can make are to turn left or turn right, to go forward, to pick up or drop objects, to toggle the state of objects, and to say 'done' to stop taking actions. Note for this problem, only the actions turn left or right and go forward are necessary to solve the problem.

The LLM is initially prompted with examples of mission statements and actions it can take to accomplish those mission statements. After these example prompts and responses, the LLM is given its mission statement and is asked to output one action at a time. After the LLM selects an action, that action is applied to the agent, which modifies what the agent observes (thereby modifying the scene VSA vector that is used to steer the LLM). The LLM can keep outputting actions until either it reaches the goal or it runs out of moves before it reaches the goal. The reward given to the agent is proportional to how many moves it took to reach the goal. If it did not accomplish the goal, it receives a reward of 0.

In this specific MiniGrid task, the agent always spawns in one of the corners of the 5x5 grid, and the green square is always located in an opposite corner of the 5x5 grid. The maximum reward the agent can receive is 0.955, assuming it wastes no actions in getting to the green square, and the average reward it will receive performing random actions is 0.13. Alternatively, if the agent is limited to randomly selecting actions from 'left', 'right', and 'forward' (i.e., the only actions required to accomplish this specific MiniGrid task), then it's average reward increases to 0.37. We use this score as a baseline of comparison to our method, since if the LLM cannot interpret the scene VSA vector at all, it will be forced to perform random actions. This implies that a reward above this random action baseline means the model is able to utilize the scene VSA vector representing the environment.

Architecture

Our strategy for integrating the information about the scene into the LLMs forward pass involves taking a hidden representation from the LLM at a particular layer (layer 12 in our case), passing it through an encoder, adding it to the scene SSP (i.e., the VSA vector describing the scene), then passing that sum through the decoder in order to steer the LLM towards the correct action. This process is depicted in Figure 9.5.

The intuition for this architecture is that while the scene SSP contains information about the environment the agent is placed in, it does not contain information about the objective the agent is trying to achieve. This information is added using the output of the encoder network, which is trained to modify the scene SSP with information needed to isolate relevant information that will help it make more accurate decisions. The decoder then converts this information into a format which will help the LLM pick the best action.

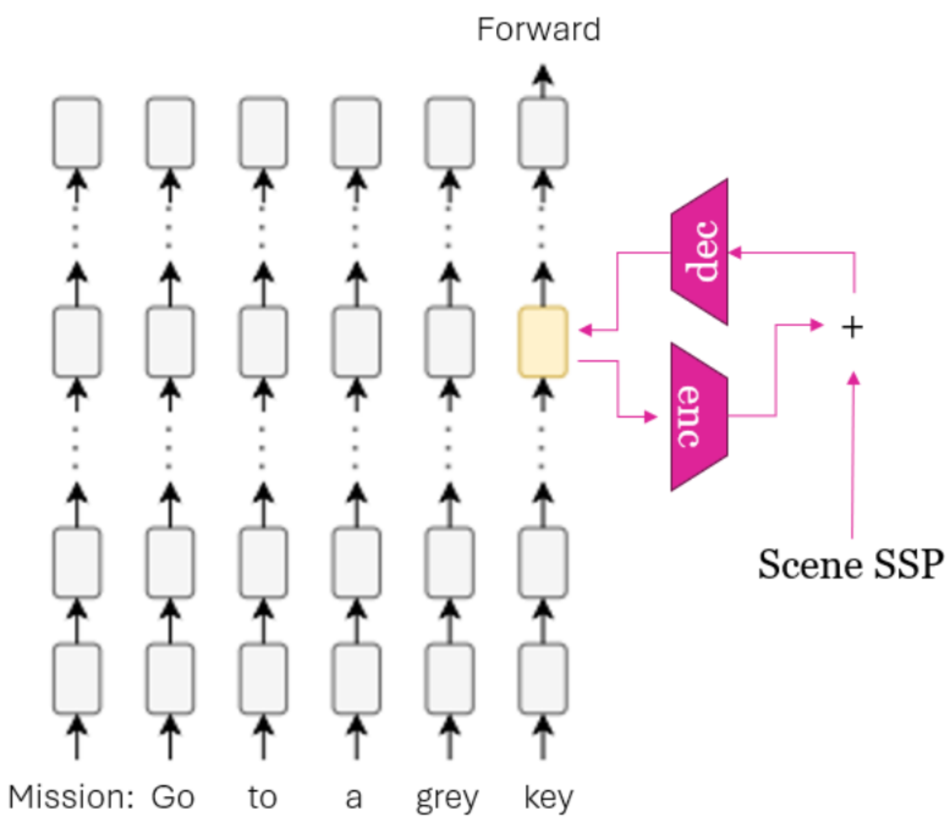


Figure 9.5: Diagram showing how our system uses a MiniGrid mission statement to query a VSA representing a scene for information required to select an action.

Training

Training the encoder and decoder in this system is more complicated than it was to train the VQA and arithmetic-solving neurosymbolic integration methods (discussed in section 9.1 and Chapter 6). This is because for any given action the model takes, it is not immediately known what the score or loss of that particular choice was. The reward of an episode is only known after the agent has made all of its actions and the episode is over.

In order to address this issue, we employ GRPO, a well-known RL training paradigm. GRPO was discussed briefly in Chapter 3 as a method for improving the reasoning capabilities of LLMs. Typically, when used in this way, the weights of the entire base LLM are modified in order to teach the agent to select actions that are parts of trajectories with relatively higher scores, and to avoid actions that are associated with trajectories with relatively lower scores. In this work, we employ GRPO, but keep the base LLM frozen while allowing the encoder and decoder weights to be trained. This teaches the LLM to modify and interpret the scene SSP such that it is more likely to select trajectories that are associated with higher rewards.

Results

To measure the performance of our approach, we measure the average reward received per episode over 120 episodes. Our results are shown in Figure 9.6. These results show that the system is able to reach an average reward around 0.6 after training. Relative to the baseline scores of 0.16 when taking totally random actions and 0.37 when taking semi-random actions, this shows our approach successfully creates a system that can interact with the scene SSP to extract useful information to inform the agent towards making sequences of actions that obtain high reward values.

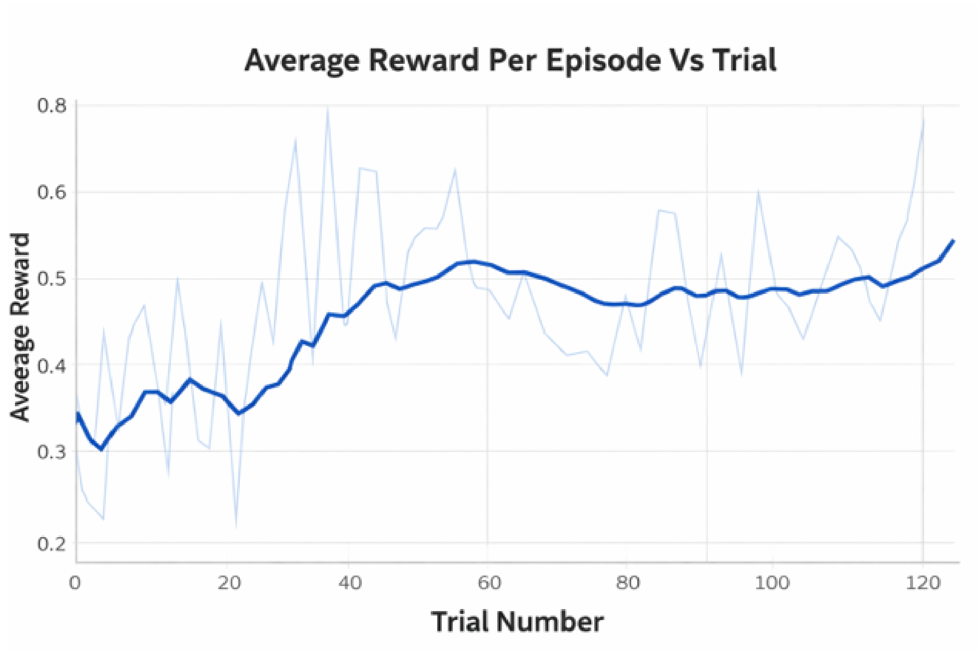


Figure 9.6: Average reward per validation example of the LLM on the MiniGrid exploration task, showing the LLM is able to slowly learn how to navigate the environment more accurately.

9.3 Context Summarization

The final application of our neurosymbolic integration method we discuss in this thesis is to compress contexts into VSA representations rather than provide them to the model in the form of additional tokens. The motivation for this objective is the fact that LLM performance is known to degrade as the size of the context increases [85, 59]. Additionally, the memory cost of LLM inference scales quadratically with context size due to the attention mechanism's requirement to compute pairwise interactions between all tokens, making inference with large contexts computationally expensive [124, 27].

The inability of LLMs to perform optimally over large contexts is a serious concern, since currently the predominant approach LLMs take towards continual learning is in-context learning [13, 31]. In-context learning is the phenomenon where a base LLM can quickly adapt to new information by providing that information in the context of the LLM's forward inference. This can be used to provide LLMs with domain specific knowledge (e.g., most RAG approaches), or to provide LLMs with their own outputs to allow them to perform more complex reasoning (e.g., CoT and complex reasoning models). In this section, we focus on the latter application, since we attempt to create a neurosymbolic LLM system which is capable of performing in-context learning by representing its own past outputs as a VSA vector, rather than as additional tokens.

9.3.1 Methodology

Our methodology involves presenting our neurosymbolic enhanced LLM with reasoning problems that require the LLM to make multiple guesses, and then represent incorrect guesses in VSA format, integrating that information into the LLM forward pass when asking it to try again. This approach is in contrast to the standard CoT approach of prompting the LLM to solve a problem, and then storing its previous guesses in the form of tokens that are re-fed back into the LLM to produce new guesses until it solves the problem. Note that in this work, we use Llama 3.1 8B as our base LLM [127].

Dataset

In order to properly test our system's ability to create VSA representations of the model's previous context, we select a reasoning problem that typically requires the LLM to provide multiple guesses before arriving at the correct solution.

One such dataset is the game of 24, wherein an LLM is provided with 4 positive integers, and is asked to combine those numbers (using each exactly once) with the addition, subtraction, multiplication, and division operations to obtain 24. This problem is commonly used to test LLMs' ability to perform intelligent search over the space of possible combinations to find the correct solution. For example, the researchers who proposed the tree of thoughts methods (as discussed in Chapter 3) have used the game of 24 to benchmark the performance of their approaches [148].

In our experiment, we utilize a game of 24 dataset [97] which provides starting numbers and difficulties for various game of 24 problems. Note that the difficulties of problems are determined by the problems' solve percentage, averaged over 6.4 million human solution attempts [97].

VSA Generation

After the LLM is prompted to solve a game of 24 problem given 4 numbers, it will attempt to create some combination of those 4 numbers that solve the problem. This output is parsed in order to determine if it correctly solved the problem. If it did, no memory VSA is generated since the LLM does not need to guess again. However, if it got an incorrect expression (e.g., did not use the 4 starting numbers, made mathematical mistakes, or the expression did not evaluate to 24), then a VSA representing the guess of the LLM is created.

This VSA is created by compositionally constructing a VSA representation of the full arithmetic expression, where each operand and operator is recursively bound and bundled according to the syntactic structure of the expression tree. Specifically, each binary operation is represented as

$$\mathbf{N_1} \circledast \text{VSA(left operand)} + \mathbf{OP} \circledast \text{VSA(operator)} + \mathbf{N_2} \circledast \text{VSA(right operand)}. \tag{9.2}$$

It is then recursively nested to reflect the hierarchical composition of sub-expressions. The resulting high-dimensional vector encodes both the symbolic content (which numbers and operations were used) and the structural form (how they were combined).

For example, if the LLM was prompted to solve the game of 24 given the starting numbers 3, 6, 6, and 11, and it responded with `3*6+6-11`, the corresponding guess VSA that would be created is shown below. The expression is formatted on multiple lines to clearly illustrate its recursive structure:

$$\begin{aligned}\mathbf{VSA}_{\text{guess}} = \mathbf{N_1} \circledast \Big(&\mathbf{N_1} \circledast \Big(\mathbf{N_1} \circledast \big(\mathbf{N_1} \circledast \mathbf{THREE} + \mathbf{OP} \circledast \mathbf{MULT} \\ &+ \mathbf{N_2} \circledast \mathbf{SIX}\big) + \mathbf{OP} \circledast \mathbf{ADD} + \mathbf{N_2} \circledast \mathbf{SIX}\Big) \\ &+ \mathbf{OP} \circledast \mathbf{SUB} + \mathbf{N_2} \circledast \mathbf{ELEVEN}\Big).\end{aligned} \tag{9.3}$$

Note that in the above equation, VSA vectors for numbers like **THREE** are represented as **ONE** $\circledast$ **ONE** $\circledast$ **ONE**, as described in Section 5.1.1.

This 'guess VSA' serves as a memory trace of the LLM's failed attempt, which is later bound with a trial marker VSA to record which iteration of the reasoning process it corresponds to. Specifically, the cumulative memory representation after n trials is defined as

$$\mathbf{VSA}_{\text{memory}} = \sum_{i=0}^{n} \mathbf{VSA}_{\mathbf{attempt}_i} \circledast \mathbf{VSA}_{\text{guess}_i} \tag{9.4}$$

where $\mathbf{VSA}_{\text{guess}_i}$ is the VSA representation one of the n previous guesses made by the LLM and each of $\mathbf{VSA}_{\mathbf{attempt}_i}$ are randomly generated VSAs.

Architecture

A diagram depicting our architecture is shown in Figure 9.7. As the figure shows, the architectural modifications to the base LLM are quite limited, involving only a decoder (we use a decoder at layer 17) to convert the memory VSA vector into a form interpretable by the LLM during its forward pass. The role of this linear decoder layer is to use the information in the memory VSA vector, which contains representations of incorrect solutions, to steer the LLM away from guessing answers it has already provided in previous guesses. Note that the LLM only has access to its previous guesses via this memory VSA vector, since we do not provide actual guesses in its context when asking it to solve a particular game of 24 puzzle for repeated attempts.

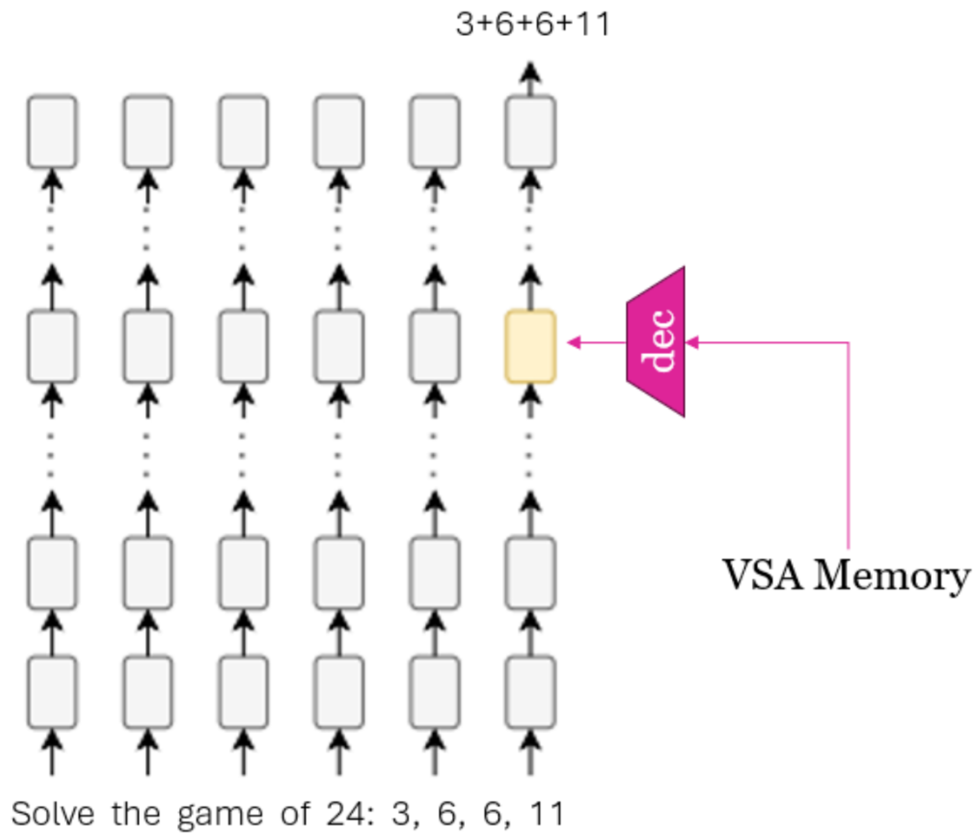


Figure 9.7: Diagram showing how our system uses VSAs representing the previous guesses of the LLM to help train the LLM to provide different guesses.

Training

To train the decoder network, we adopt an approach similar to that of Section 9.2. That is, we use GRPO to train the decoder. This is because the outputs from the model cannot be evaluated at a token-level, since the LLM solves the question correctly only if the entire sequence of tokens it outputs is correct. We compute our reward signal as a weighted combination of three components: a **main reward**, which measures whether the generated expression is both numerically equivalent to a valid solution and uses the same operands; a **mathematical accuracy reward**, which verifies whether the generated equation is mathematically correct (i.e., the left-hand and right-hand sides of the equality are equal); and an **input number reward**, which checks whether the model used each of the input numbers exactly once. Formally, the total group reward for each generation step is given by

$$R_i = 10\, r_{\text{main}}^{(i)} + 1\, r_{\text{math}}^{(i)} + 2\, r_{\text{input}}^{(i)}, \tag{9.5}$$

where $r_{\text{main}}^{(i)}$, $r_{\text{math}}^{(i)}$, and $r_{\text{input}}^{(i)}$ denote the individual reward components at sample i. These rewards are computed for each generated expression and normalized within a group to obtain an advantage estimate,

$$A_i = \frac{R_i - \mu_R}{\sigma_R + 10^{-2}}, \tag{9.6}$$

where μ_R and σ_R are the mean and standard deviation of the group rewards, respectively. The decoder is then updated using the GRPO objective, which maximizes the expected return under the clipped importance ratio between the new and old policy probabilities.

Note that while training our model, we employ curriculum learning by starting our training process with the game of 24 problems that were solved by the highest percentage of humans, eventually picking more difficult problems as the training proceeds. This is because GRPO requires a diverse set of reward signals in order to operate, meaning that starting the training with difficult questions where the base LLM struggles to get any of it's group rewards to be non-zero would prevent any learning from occurring.

Results

We measure the average reward obtained by our system as it attempts to answer progressively more difficult problems in order to determine its accuracy. Figure 9.8 shows that the average reward increases as the training progresses, reaching a reward of up to 6. Since the problems become progressively more difficult, the fact that the reward increases over problem number suggests that our neurosymbolic memory system is partially able to provide the LLM information about its previous guessing during its forward pass.

In addition to testing our VSA memory model, we also test a vanilla LLM on the same problems. In this setup, we simply prompt the LLM 10 times (i.e., the same number of times we allow the VSA memory model to attempt a question) with the question and allow it to generate answers. Figure 9.8 shows that after being trained on about 200 problems, the VSA memory model achieves a higher level of performance than the vanilla LLM, further indicating that our VSA memory approach is able to utilize the VSA memory state to obtain larger rewards than a baseline LLM. More extensive validation and comparisons are required to verify this conclusion, however, and are planned to be conducted in future work.

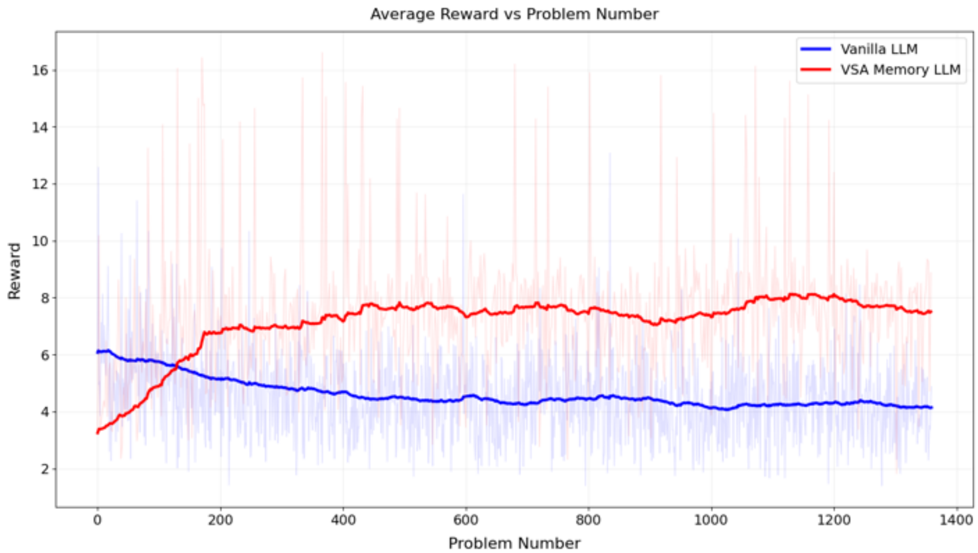


Figure 9.8: Average Reward per example of the LLM on the game of 24 task, showing the LLM is slowly learning to use the VSA vector providing information about its previous guesses to improve the reward it receives relative to the baseline LLM. Note that as the problem number increases, the difficulty (as assessed by human solve rate) also increases, meaning the LLM is faced with more challenging problems as it trains longer.

Chapter 10

Conclusion

This thesis has presented a framework for integrating symbolic reasoning capabilities into large language models through Vector Symbolic Algebras. By developing methods to extract, manipulate, and inject symbolic representations during neural network computation, we have demonstrated that the dichotomy between neural and symbolic approaches to artificial intelligence can be bridged using VSAs. Our work reveals both theoretical insights about how LLMs represent information internally and practical techniques for augmenting their reasoning capabilities.

10.1 Summary of Contributions

Our primary contribution is the development of a neurosymbolic integration method that enables symbolic algorithms to operate directly within the computational flow of transformer-based language models. This approach addresses fundamental limitations in LLM reasoning by creating a bridge between the continuous, high-dimensional representations of neural networks and the discrete, rule-based operations of symbolic systems.

The core technical innovation involves encoding LLM hidden states into Vector Symbolic Algebras, compositional representations that support symbolic manipulation while remaining compatible with gradient-based learning. Our experiments on mathematical reasoning tasks demonstrate the effectiveness of this approach, achieving 88.6% lower cross-entropy loss and solving 15.4 times more problems correctly than state-of-the-art alternatives including chain-of-thought prompting and LoRA fine-tuning. Importantly, our selective intervention mechanism preserves the model's performance on tasks outside the symbolic domain, avoiding the catastrophic forgetting often associated with task-specific fine-tuning.

A key theoretical finding is that LLMs naturally develop internally separable representations for symbolic concepts during standard pretraining. Our ability to accurately extract mathematical operands, operators, and problem types using simple linear transformations suggests that transformer models implicitly learn to organize symbolic information in distinguishable subspaces of their hidden representations. This discovery has implications beyond our specific application, contributing to the broader understanding of how neural networks represent and process structured information.

We extended the basic framework in several important directions. The development of transformer-based encoders demonstrated that architectural improvements can enhance robustness to input variation while reducing training data requirements. Empirically, our transformer encoder achieved 73.7% higher accuracy than linear encoders when trained on only 1,000 examples with varied question formats. This advancement makes neurosymbolic integration more practical for real-world applications where training data may be limited and question formats are diverse.

10.2 Broader Applications and Implications

The versatility of our neurosymbolic framework is demonstrated through applications beyond mathematical reasoning. In visual question answering, we showed that language-only models can perform visual reasoning by encoding segmented images as compositional VSAs, achieving 92% accuracy without requiring multimodal architectures. This approach sidesteps fundamental challenges in vision-language models, such as the discretization problem and the computational cost of processing high-resolution images through attention mechanisms. It also provides a framework for more interpretable and reliable multimodal models, where different modular components work together to semantically decompose images and then dynamically query them for information necessary to solve a given task.

Our work on environment navigation revealed that LLMs can use spatial semantic pointers to interpret and act within structured environments according to natural language instructions. The system learned to query VSA representations of spatial scenes to make navigation decisions, demonstrating that abstract spatial reasoning can be performed in the neurosymbolic space rather than through explicit tokenization of an agents environment.

The context compression application addresses one of the most pressing limitations of current LLMs: performance degradation with increasing context length. By encoding reasoning histories as compositional VSA vectors, we maintain essential information about past attempts while avoiding the quadratic scaling of attention mechanisms. This approach points toward a future where LLMs can engage in extended problem-solving without the computational and performance penalties currently associated with long contexts.

10.3 Limitations and Future Directions

While our results are promising, several limitations warrant discussion. First, our current implementation requires manual specification of the symbolic algorithms to be applied for each problem type. Future work should explore methods for automatically learning or discovering appropriate symbolic manipulations from data, potentially through program synthesis or meta-learning approaches.

Second, a critical direction for future work involves developing unsupervised methods for generating VSA representations, analogous to how sparse autoencoders (SAEs) discover interpretable features without labeled data. Currently, our approach requires supervised training with explicitly defined VSA structures for each concept. An unsupervised VSA discovery mechanism would allow the system to automatically identify and encode concepts the LLM is processing, creating libraries of compositional representations without manual annotation. This would represent a significant advance over current SAEs, which, while providing unbiased interpretations of LLM internals, cannot naturally represent hierarchical and compositional structures. The inherent support of VSAs for binding and bundling operations enables them to capture relationships between concepts, nested structures, and abstract compositions that flat sparse representations cannot express. Such a system could potentially discover not just what concepts an LLM is considering, but how those concepts relate to and compose with each other, providing deeper insights into the model's reasoning process, while also paving the way for integration with more complex symbolic reasoning modules.

Third, our applications beyond mathematical reasoning, while demonstrating the versatility of the neurosymbolic framework, require more comprehensive evaluation to establish definitive conclusions. The visual question answering system, though achieving 92% accuracy on simple object detection queries, has only been tested on a limited set of question types using COCO dataset images. More complex visual reasoning tasks, such as counting objects, understanding spatial relationships, or answering questions requiring visual inference, remain unexplored. Similarly, our environment navigation experiments were confined to simple 5×5 MiniGrid environments with basic objectives. Scaling to larger, more complex environments with partial observability, dynamic obstacles, and multi-step goals would provide stronger evidence for the approach's practical utility. The context compression application, while showing promising initial results on the game of 24, lacks comparison to existing context management techniques and has not been evaluated on diverse reasoning tasks where context accumulation is critical, such as multi-hop question answering or extended dialogue understanding.

Finally, the scalability of our approach to larger models and different model families remains to be fully explored. While we demonstrated effectiveness on models up to 8 billion parameters, the behaviour of neurosymbolic integration in models with hundreds of billions of parameters, or in different families of models (such as Qwen, OpenAI, Mistral, etc.), requires further investigation.

10.4 Toward Hybrid Intelligence Systems

This work contributes to a broader goal of hybrid intelligence systems that combine the complementary strengths of neural and symbolic computation. Human cognition appears to seamlessly integrate pattern recognition with rule-based reasoning, statistical inference with logical deduction, and intuitive understanding with systematic analysis. Our neurosymbolic framework represents a step toward artificial systems with similar capabilities.

The ability to extract and manipulate symbolic representations from neural networks has implications beyond immediate performance improvements. It provides a pathway toward more interpretable AI systems, where the reasoning process can be inspected, verified, and corrected at the symbolic level. This interpretability is crucial for high-stakes applications where understanding and trusting AI decisions is paramount.

As language models continue to grow in scale and capability, the need for reliable reasoning mechanisms becomes increasingly critical. The neurosymbolic approach presented in this thesis offers a practical path forward, which enhances reasoning capabilities while maintaining computational efficiency and preserving the models' broad competencies. While significant challenges remain, our results suggest that the neurosymbolic integration is as a viable and valuable direction for advancing artificial intelligence toward systems that combine the best of both neural and symbolic paradigms.

The ultimate goal of artificial intelligence research is to create systems that can reason, learn, and interact with the world as flexibly and reliably as humans do. This requires both the pattern recognition capabilities of neural networks and the systematic reasoning of symbolic systems. The work presented in this thesis contributes to this goal by demonstrating that these capabilities need not exist in isolation but can be integrated within a unified computational framework. As we continue to push the boundaries of what artificial intelligence can achieve, such integration will likely prove essential for creating truly intelligent systems.

References

[1] Joshua Ainslie, Santiago Ailloud, Chris Coombes, James Deaton, Philipp Dufter, Zi Lin Goldman, James Lee-Thorp, Tao Lei, Lin Yang, Bo Yu, et al. GQA: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

[2] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems*, 35:23716–23736, 2022.

[3] Emmanuel Ameisen, Jack Lindsey, Adam Pearce, Wes Gurnee, Nicholas L. Turner, Brian Chen, 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, Andrew Persic, Zhenyi Qi, T. Ben Thompson, Sam Zimmerman, Kelley Rivoire, Thomas Conerly, Chris Olah, and Joshua Batson. Circuit tracing: Revealing computational graphs in language models. *Transformer Circuits Thread*, 2025.

[4] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *Proceedings of the IEEE international conference on computer vision*, pages 2425–2433, 2015.

[5] J. Austin, A. Odena, M. Nye, M. Bosma, S. Mishra, S. Khan, Q. Le, J. Dean, and J. Devlin. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[6] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

[8] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.

[9] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

[10] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *Journal of machine learning research*, 3:1137–1155, 2003.

[11] Lukas Berglund, Meg Tong, Max Kaufmann, Mikita Balesni, Asa Cooper Stickland, Tomasz Korbak, and Owain Evans. The reversal curse: Llms trained on ”a is b” fail to learn ”b is a”. *arXiv preprint arXiv:2309.12288*, 2023.

[12] Ralph Allan Bradley and Milton E Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.

[13] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[14] Stephen Casper, Xander Davies, Claudia Shi, Thomas Krendl Gilbert, Jérémy Scheurer, Javier Rando, Rachel Freedman, Tomasz Korbak, David Lindner, Pedro Freire, et al. Open problems and fundamental limitations of reinforcement learning from human feedback. *arXiv preprint arXiv:2307.15217*, 2023.

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

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

[17] Bowen Cheng, Ishan Misra, Alexander G Schwing, Alexander Kirillov, and Rohit Girdhar. Masked-attention mask transformer for universal image segmentation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 1290–1299, 2022.

[18] Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo de Lazcano, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *CoRR*, abs/2306.13831, 2023.

[19] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[20] Francois Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. Arc prize 2024: Technical report, 2025.

[21] Francois Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. Arc-agi-2: A new challenge for frontier ai reasoning systems, 2025.

[22] Xuan Choo and Chris Eliasmith. A spiking neuron model of serial-order recall. In Richard Cattrambone and Stellan Ohlsson, editors, *32nd Annual Conference of the Cognitive Science Society*, Portland, OR, 08/2010 2010. Cognitive Science Society.

[23] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[24] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

[25] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. What does bert look at? an analysis of bert's attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP*, pages 276–286, 2019.

[26] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[27] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, 2022.

[28] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2019.

[29] Varun Dhanraj and Chris Eliasmith. Improving rule-based reasoning in LLMs using neurosymbolic representations. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng, editors, *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 30577–30596, Suzhou, China, November 2025. Association for Computational Linguistics.

[30] Ning Ding, Yutong Zheng, et al. Delta tuning: A comprehensive study of parameter-efficient methods for pre-trained language models. *arXiv preprint arXiv:2203.06904*, 2022.

[31] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.

[32] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023.

[33] Nicole Sandra-Yaffa Dumont. *Symbols, Dynamics, and Maps: A Neurosymbolic Approach to Spatial Cognition*. Phd thesis, University of Waterloo, Waterloo, ON, 2025.

[34] Jacob Dunefsky, Philippe Chlenski, and Neel Nanda. Transcoders find interpretable llm feature circuits. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

[35] Jacob Dunefsky, Philippe Chlenski, and Neel Nanda. Transcoders find interpretable llm feature circuits. *arXiv preprint arXiv:2406.11944*, 2024.

[36] Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D Hwang, et al. Faith and fate: Limits of transformers on compositionality. *arXiv preprint arXiv:2305.18654*, 2024.

[37] Nelson Elhage, Neel Nanda, et al. Mathematical interpretability with sparse autoencoders, 2021.

[38] Nelson Elhage, Neel Nanda, et al. A mathematical framework for transformer circuits. *Transformer Circuits Thread, OpenAI*, 2022.

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

[40] Chris Eliasmith. How to build a brain: A neural architecture for biological cognition. *Oxford University Press*, 2013.

[41] Chris Eliasmith and Charles H Anderson. *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. MIT Press, 2003.

[42] Chris Eliasmith, Terrence C. Stewart, Xuan Choo, Trevor Bekolay, Travis DeWolf, Yichuan Tang, and Daniel Rasmussen. A large-scale model of the functioning brain. *Science*, 338(6111):1202–1205, 2012.

[43] E. Paxon Frady, Denis Kleyko, Christopher J. Kymn, Bruno A. Olshausen, and Friedrich T. Sommer. Computing on functions using randomized vector representations. In *Proceedings of the Neuro-Inspired Computational Elements Conference (NICE '22)*, pages 115–122, Virtual Event, USA, 2022.

[44] Markus Freitag and Yaser Al-Onaizan. Beam search strategies for neural machine translation. *arXiv preprint arXiv:1702.01806*, 2017.

[45] Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. *arXiv preprint arXiv:2210.10760*, 2023.

[46] Artur S d'Avila Garcez, Luis C Lamb, and Dov M Gabbay. *Neural-Symbolic Cognitive Reasoning*. Springer, 2009.

[47] Ross W. Gayler. Vector symbolic architectures answer jackendoff's challenges for cognitive neuroscience, 2003.

[48] Ross W. Gayler and Simon D. Levy. Vector symbolic architectures: A new building material for artificial general intelligence. In *Proceedings of the 2008 Conference on Artificial General Intelligence*, pages 122–123, Amsterdam, The Netherlands, 2008. IOS Press.

[49] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[50] Aaron Grattafiori et al. The llama 3 herd of models. arXiv preprint arXiv:2407.21783, 2024.

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

[52] Michael Hanna, Tiago Pimentel, Dieuwke Stojanovski, and Ryan Cotterell. How does gpt-2 compute greater-than?: Interpreting mathematical abilities in a pre-trained language model. *arXiv preprint arXiv:2305.00586*, 2024.

[53] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 16000–16009, 2022.

[54] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[55] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[56] John Hewitt and Christopher D. Manning. 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, 2019.

[57] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[58] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2020.

[59] Kelly Hong, Anton Troynikov, and Jeff Huber. Context rot: How increasing input tokens impacts llm performance. Technical report, Chroma, July 2025.

[60] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, et al. Parameter-efficient transfer learning for nlp. In *Proceedings of the 36th International Conference on Machine Learning*, 2019.

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

[62] Drew A Hudson and Christopher D Manning. Gqa: A new dataset for real-world visual reasoning and compositional question answering. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6700–6709, 2019.

[63] Chao Jia, Yinfei Yang, Ye Xia, Yi-Ting Chen, Zarana Parekh, Hieu Pham, Quoc Le, Yun-Hsuan Sung, Zhen Li, and Tom Duerig. Scaling up visual and vision-language representation learning with noisy text supervision. *arXiv preprint arXiv:2102.05918*, 2021.

[64] Isaac Joffe and Chris Eliasmith. An exploration of a biologically plausible model of mental multiplication. Unpublished manuscript, University of Waterloo, 2024.

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

[66] Damjan Kalajdzievski. Scaling laws for forgetting when fine-tuning large language models. *arXiv preprint arXiv:2401.05605*, 2024.

[67] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[68] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2015.

[69] Robert Kirk, Ishita Mediratta, Christoforos Nalmpantis, Jelena Luketina, Eric Hambro, Edward Grefenstette, and Roberta Raileanu. Understanding the effects of rlhf on llm generalisation and diversity. *arXiv preprint arXiv:2310.06452*, 2023.

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

[71] Brent Komer, Terrence C. Stewart, Aaron R. Voelker, and Chris Eliasmith. A neural representation of continuous space using fractional binding. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, University of Waterloo, Waterloo, ON, Canada, 2019. Centre for Theoretical Neuroscience.

[72] Tamera Lanham, Anna Chen, Ansh Radhakrishnan, Benoit Steiner, Carson Denison, Danny Hernandez, Dustin Li, Esin Durmus, Evan Hubinger, Jackson Kernion, et al. Measuring faithfulness in chain-of-thought reasoning. *arXiv preprint arXiv:2307.13702*, 2023.

[73] Andrew Le et al. Probing neural networks for interpretability. *arXiv preprint arXiv:2110.02096*, 2021.

[74] Seungpil Lee, Woochang Sim, Donghyeon Shin, Wongyu Seo, Jiwon Park, Seokki Lee, Sanha Hwang, Sejin Kim, and Sundong Kim. Reasoning abilities of large language models: In-depth analysis on the abstraction and reasoning corpus. *ACM Transactions on Intelligent Systems and Technology*, 2025. Evaluates LLMs on the ARC benchmark and finds that, although they show some inference ability, they still significantly lag behind human-level reasoning in logical coherence, compositionality, and productivity, highlighting ARC as challenging for LLMs relative to humans.

[75] Jan Leike, David Krueger, Tom Everitt, Miljan Martic, Vishal Maini, and Shane Legg. Scalable agent alignment via reward modeling: a research direction. *arXiv preprint arXiv:1811.07871*, 2018.

[76] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. *arXiv preprint arXiv:2301.12597*, 2023.

[77] Q. Li et al. Gsm-plus: A comprehensive benchmark for evaluating grade school math reasoning in large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024. Reports a screened human baseline on GSM8K-style problems and finds frontier LLMs match or exceed that baseline with chain-of-thought style solutions.

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

[79] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

[80] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755, 2014.

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

[82] 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, Andrew Persic, Zhenyi Qi, T. Ben Thompson, Sam Zimmerman, Kelley Rivoire, Thomas Conerly, Chris Olah, and Joshua Batson. On the biology of a large language model. *Transformer Circuits Thread*, 2025.

[83] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *Advances in neural information processing systems*, 36, 2024.

[84] Nelson F Liu, Matt Gardner, Yonatan Belinkov, Matthew Peters, and Noah A Smith. 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, 2019.

[85] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

[86] Xiaotian Liu et al. Symbolic reasoning and large language models. *arXiv preprint arXiv:2303.04249*, 2023.

[87] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[88] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2017.

[89] Liangchen Luo, Yudong Xu, Hongkun Zhao, Zhe Cao, Yilun Shu, Wentao Yu, Jing Liu, Hanxiao Liu, and Kai Chen. Improve mathematical reasoning in language models by automated process supervision. *arXiv preprint arXiv:2406.06592*, 2024.

[90] Kyle Mahowald, Anna A Ivanova, Idan A Blank, Nancy Kanwisher, Joshua B Tenenbaum, and Evelina Fedorenko. Dissociating language and thought in large language models. *Trends in Cognitive Sciences*, 2024.

[91] Gary Marcus. Deep learning is hitting a wall. *Nautilus*, 2022.

[92] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2018.

[93] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, 2010.

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

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

[96] Neel Nanda, Lawrence Chan, Tom Liberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability. *arXiv preprint arXiv:2301.05217*, 2023.

[97] N Nile. 24-game dataset, 2023.

[98] Maxwell Nye, Jacob Andreas, et al. Work hard, play hard: Language models in learning and reasoning. *Advances in Neural Information Processing Systems*, 2021.

[99] Chris Olah, Arvind Satyanarayan, et al. Zoom in: An introduction to circuits. *Distill*, 2020.

[100] OpenAI. Learning to reason with llms. *OpenAI Blog*, 2024.

[101] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[102] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013.

[103] Anna Penzkofer, Karim Habashy, Abhishek Tyagi, Yuhao Zhu, Chris Eliasmith, and Andreas Bulling. SSPictr: Spatial semantic pointer picture representation, 2024.

[104] Flavio Petruzzellis, Alberto Testolin, and Alessandro Sperduti. Assessing the emergent symbolic reasoning abilities of llama large language models, 2024.

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

[106] Reiner Pope, Siddhartha Balaji, Leo Crall, Sheila Chettiar, Firas Abuzaid, Christopher Klein, Michael Mahoney, and Brent Saeta. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems*, volume 5, pages 258–274, 2023.

[107] Python Software Foundation. Arbitrary-precision integer implementation details. https://docs.python.org/3/library/stdtypes.html#int, 2024. Section "*Plain integer objects*". Accessed 12 May 2025.

[108] Linlu Qiu, Peter Lin, Jack Hessel, Yejin Choi, and Liangze Yu. Large language model guided tree-of-thought, 2023.

[109] Linlu Qiu, Peter Shaw, Panupong Pasupat, Sean Welleck, Niket Tandon, Omer Levy, Asli Celikyilmaz, Mohit Bansal, Chris Callison-Burch, et al. Phenomenal yet puzzling: Testing inductive reasoning capabilities of language models with hypothesis refinement. *arXiv preprint arXiv:2410.15943*, 2024.

[110] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763, 2021.

[111] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *OpenAI preprint*, 2018.

[112] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[113] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*, 2023.

[114] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. A primer on neural network architectures for natural language processing. *Journal of Artificial Intelligence Research*, 67:1–96, 2020.

[115] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023.

[116] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[117] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[118] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, YK Li, Y Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[119] Noam Shazeer. Fast transformer decoding: One write-head is all you need. In *arXiv preprint arXiv:1911.02150*, 2019.

[120] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

[121] Terrence C Stewart, Trevor Bekolay, Xuan Choo, and Chris Eliasmith. Representing and reasoning with vector symbolic architectures in nengo. In *Proceedings of the 37th Annual Conference of the Cognitive Science Society*, pages 2290–2295, 2015.

[122] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020.

[123] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[124] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Computing Surveys*, 55(12):1–28, 2022.

[125] Adly Templeton, Tom Conerly, Jonathan Marcus, Jack Lindsey, Trenton Bricken, Brian Chen, Adam Pearce, Craig Citro, Emmanuel Ameisen, Andy Jones, et al. Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet. *Anthropic*, 2024.

[126] Ian Tenney, Dipanjan Das, and Ellie Pavlick. Bert rediscovers the classical nlp pipeline. *arXiv preprint arXiv:1905.05950*, 2019.

[127] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[128] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[129] Maria Tsimpoukelli, Jacob L Menick, Serkan Cabi, SM Eslami, Oriol Vinyals, and Felix Hill. Multimodal few-shot learning with frozen language models. *Advances in neural information processing systems*, 34:200–212, 2021.

[130] Miles Turpin, Julian Michael, Ethan Perera, and Samuel R Bowman. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. *arXiv preprint arXiv:2305.04388*, 2024.

[131] Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.

[132] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

[133] Jesse Vig and Yonatan Belinkov. Analyzing the structure of attention in a transformer language model. *arXiv preprint arXiv:1906.04284*, 2019.

[134] Jianfeng Wang, Fei Huang, et al. Efficient fine-tuning of large language models with lora. *arXiv preprint arXiv:2303.01234*, 2023.

[135] Kevin Wang, Alexandre Variengien, Arthur Conmy, Buck Shlegeris, and Jacob Steinhardt. Interpretability in the wild: A circuit for indirect object identification in gpt-2 small. *arXiv preprint arXiv:2211.00593*, 2023.

[136] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2023.

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

[138] Yizhou Wang, Shubham Jha, and Kamalika Chaudhuri. What makes multi-modal learning better than single (provably). *Advances in Neural Information Processing Systems*, 36, 2023.

[139] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

[140] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

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

[142] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

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

[144] Zhaofeng Wu, Linlu Qiu, Alexis Ross, Ekin Akyürek, Boyuan Chen, Bailin Hu, Shashank Raghavendra, and Jacob Andreas. Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks. *arXiv preprint arXiv:2307.02477*, 2024.

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

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

[147] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On layer normalization in the transformer architecture. *arXiv preprint arXiv:2002.04745*, 2020.

[148] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023.

[149] Mert Yuksekgonul, Federico Bianchi, Pratyusha Kalluri, Dan Jurafsky, and James Zou. When and why vision-language models behave like bags-of-words, and what to do about it? *arXiv preprint arXiv:2210.01936*, 2023.

[150] Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D Goodman. Quiet-star: Language models can teach themselves to think before speaking. *arXiv preprint arXiv:2403.09629*, 2024.

[151] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

[152] Hugh Zhang, Jeff Da, Dean Lee, Vaughn Robinson, Catherine Wu, Will Song, Pranav Raja, Charlotte Zhuang, Dylan Slack, Qin Lyu, Sean Hendryx, Russell Kaplan, Michele Lunati, and Summer Yue. A careful examination of large language model performance on

grade school arithmetic. *arXiv preprint arXiv:2405.00332*, 2024. Introduces GSM1k, a contamination-controlled math reasoning benchmark. Finds frontier LLMs maintain high accuracy close to (or at) human-level even on unseen problems, implying true multi-step reasoning rather than pure memorization.

[153] Denny Zhou, Quoc Le, Xuezhi Wang, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

[154] Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*, 2019.

APPENDICES

Appendix A

Error Analysis of Chain-of-Thought Reasoning

One interesting result of Section 7.2 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 and their weaknesses in performing program synthesis in the token space.

A.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 7.1). This demonstrates the LLM's strength in pattern-matching tasks, while underscoring its limitations in program 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

A.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:

...

A.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.
>
> ...