NeuroSymbolicCodeTransformer

A Neuro-Symbolic Transformer for Code Generation via Knowledge Distillation and REINFORCE Policy Gradient

Training a 13.32M-Parameter Hybrid Neuro-Symbolic Model on Consumer Hardware
Using a 9B-Parameter Reasoning Teacher and Reinforcement Learning

Version 1.0 — July 2026

DexopT

Repository: huggingface.co/DexopT/neuro-symbolic-coder-13M

Abstract

We present NeuroSymbolicCodeTransformer, a 13.32-million-parameter hybrid neuro-symbolic transformer that learns Python code generation from mathematical representations of Abstract Syntax Trees (ASTs). The model distills knowledge from a 9-billion-parameter reasoning LLM (Ornith, Qwen3.5-based) into a compact architecture trainable on consumer GPUs with only 6 GB of VRAM. The architecture introduces five custom neuron classes—LogicGate, Program, LocalTree, Routing, and Memory neurons—organized into Mixture-of-Experts (MoE)-routed transformer blocks with differentiable external memory. Training proceeds through a three-phase pipeline: (1) a teacher LLM generates Python code from natural language prompts, which is converted to structured mathematical AST specifications; (2) the student model learns to map those specifications to token sequences via supervised learning with bfloat16 AMP and gradient checkpointing; and (3) REINFORCE policy-gradient reinforcement learning refines the model using teacher-scored feedback over 30 iterations with 2 candidates per iteration. On Run A (remote Ornith server), the RL phase achieved a best score of 100.00/100 (average 28.33/100) with peak VRAM usage of only 25.4 MiB for the model at bfloat16 precision. The entire training pipeline completes in approximately 68 minutes. We situate this work within the broader neuro-symbolic AI and knowledge distillation literature, comparing our approach to recent methods including Neural-Symbolic Collaborative Distillation (NesyCD), Agent Distillation, and differentiable logic gate networks. Our results demonstrate that small, structurally decomposed neuro-symbolic architectures can serve as effective student models for code generation when paired with large reasoning teachers and RL-based policy refinement.

1. Introduction

1.1 Motivation

Large Language Models (LLMs) have demonstrated remarkable capabilities in code generation, mathematical reasoning, and structured problem-solving. Models such as GPT-4, DeepSeek-R1, and Qwen have shown that chain-of-thought reasoning combined with massive parameter counts yields state-of-the-art performance across coding benchmarks. However, deploying these models in resource-constrained environments—edge devices, local development workstations, privacy-sensitive applications—remains challenging due to their computational and memory requirements. A 9B-parameter model requires approximately 18 GB of VRAM at FP16 precision, placing it well beyond the capabilities of most consumer GPUs.

Knowledge distillation offers a principled path toward compact models by transferring capabilities from large teacher models to smaller student models. Traditional distillation approaches focus on matching output distributions or hidden states. More recent work in the neuro-symbolic tradition suggests that decomposing model computation into semantically meaningful, structurally interpretable components can yield more efficient learners with better generalization properties.

This whitepaper describes NeuroSymbolicCodeTransformer, a 13.32M-parameter model that combines neuro-symbolic architectural decomposition with knowledge distillation and reinforcement learning. The key insight is that code generation, unlike general text generation, is fundamentally a structured task: programs have well-defined syntax, compositional semantics, and can be represented as Abstract Syntax Trees (ASTs). By baking structural priors into the architecture itself—through custom neuron types that mirror programming language constructs—we enable a compact model to learn code generation effectively from a much larger teacher.

1.2 Contributions

1. **Five custom neuron classes** integrated into a Mixture-of-Experts transformer architecture: LogicGateNeuron (differentiable AND/OR/XOR/NOT), ProgramNeuron (differentiable sum/product/max/min/mean), LocalTreeNeuron (depthwise convolution for tree hierarchy), RoutingNeuron (MoE-style gating), and MemoryNeuron (32-slot differentiable memory with attention read and gated write).

2. **A three-phase training pipeline** combining teacher-generated data (Phase 1), supervised learning on AST mathematical specifications (Phase 2), and REINFORCE policy gradient with teacher-scored feedback (Phase 3).
3. **Consumer-hardware feasibility:** Full training on an NVIDIA RTX 4050 (6 GB VRAM) with bfloat16 AMP and gradient checkpointing, peaking at ~500 MiB VRAM.
4. **Empirical results** from two training configurations (remote and local teacher), including RL score trajectories, loss curves, and VRAM utilization metrics.
5. **A production-ready deployment:** FastAPI inference server, CLI interface, 27-unit test suite, and HuggingFace model distribution.

2. Background and Related Work

2.1 Neuro-Symbolic AI

Neuro-symbolic AI seeks to combine the pattern recognition capabilities of neural networks with the structured reasoning of symbolic systems. Recent work spans a spectrum from fully differentiable logic (Logical Neural Networks, Riegel et al., 2020) to hybrid systems that interleave neural and symbolic components. Neural Logic Machines (Dong et al., 2019) implement first-order logic deduction using tensor operations, achieving perfect generalization on relational reasoning tasks. Deep Differentiable Logic Gate Networks (Petersen et al., 2022; Mommen et al., 2026) replace traditional neural building blocks with logic gates, achieving competitive classification accuracy with dramatically fewer operations.

Our work occupies a distinct position in this landscape: rather than replacing all neural components with logic gates or implementing full first-order logic, we inject neuro-symbolic priors into specific architectural positions within a standard transformer. The LogicGateNeuron handles conditional branching patterns common in code; the ProgramNeuron models arithmetic computation; the LocalTreeNeuron captures the parent-child hierarchy of ASTs; and the RoutingNeuron provides learned dispatch across these specialized experts. This approach—what we term *architectural neuro-symbolic injection*—allows the model to leverage structural knowledge about the target domain without sacrificing the flexibility of learned neural representations.

2.2 Knowledge Distillation for Small Language Models

Neural-Symbolic Collaborative Distillation (NesyCD; Liao et al., 2025, AAAI) decouples general and specialized knowledge during distillation, using a symbolic knowledge base to store sparse, task-specific knowledge while transferring general reasoning capabilities through standard neural distillation. This decoupling enabled LLaMA3-8B and Qwen2-7B to surpass GPT-3.5-turbo on BBH and GSM8K benchmarks. Agent Distillation (Kang et al., 2025) transfers full task-solving behavior —including tool use (retrieval, code execution)—from LLM agents to models as small as 0.5B parameters, with first-thought prefix prompting for trajectory quality and self-consistent action generation for robustness.

Symmetry-Aware Code Generation (Li et al., 2025) distills intermediate reasoning steps (pseudocode) alongside final code from LLMs to smaller models, achieving up to 74% improvement in Top-1 accuracy. In the reinforcement learning domain, RLVR (RL with Verifiable Rewards; Skopin & Kotelnikov, 2026) demonstrates that small models (Qwen3-0.6B, Llama3.2-1B)

benefit significantly from unit-test-based RL, with combined rewards (unit tests + static analysis) yielding +13 percentage points in pass@1 on MBPP.

2.3 Memory-Augmented and Graph-Routed Transformers

Graph Memory Transformer (GMT v7; 2026) replaces the standard Feed-Forward Network with a learned memory graph of centroids connected by a directed transition matrix. Each block routes tokens to source centroids, propagates through the graph, and returns gated displacement vectors —making the FFN computation structurally legible. Memory-Augmented Transformers (Tibrewal, 2026) implement learnable memory banks with cross-attention, MoE-inspired chapter routing, and token-level sparse kernels for very large memory banks. Our MemoryNeuron shares the spirit of these approaches but operates at a smaller scale (32 slots per block) with gated soft-update writes (decay 0.99, rate 0.01), providing long-range dependency tracking without the complexity of chapter-based routing.

2.4 REINFORCE for Language Model Training

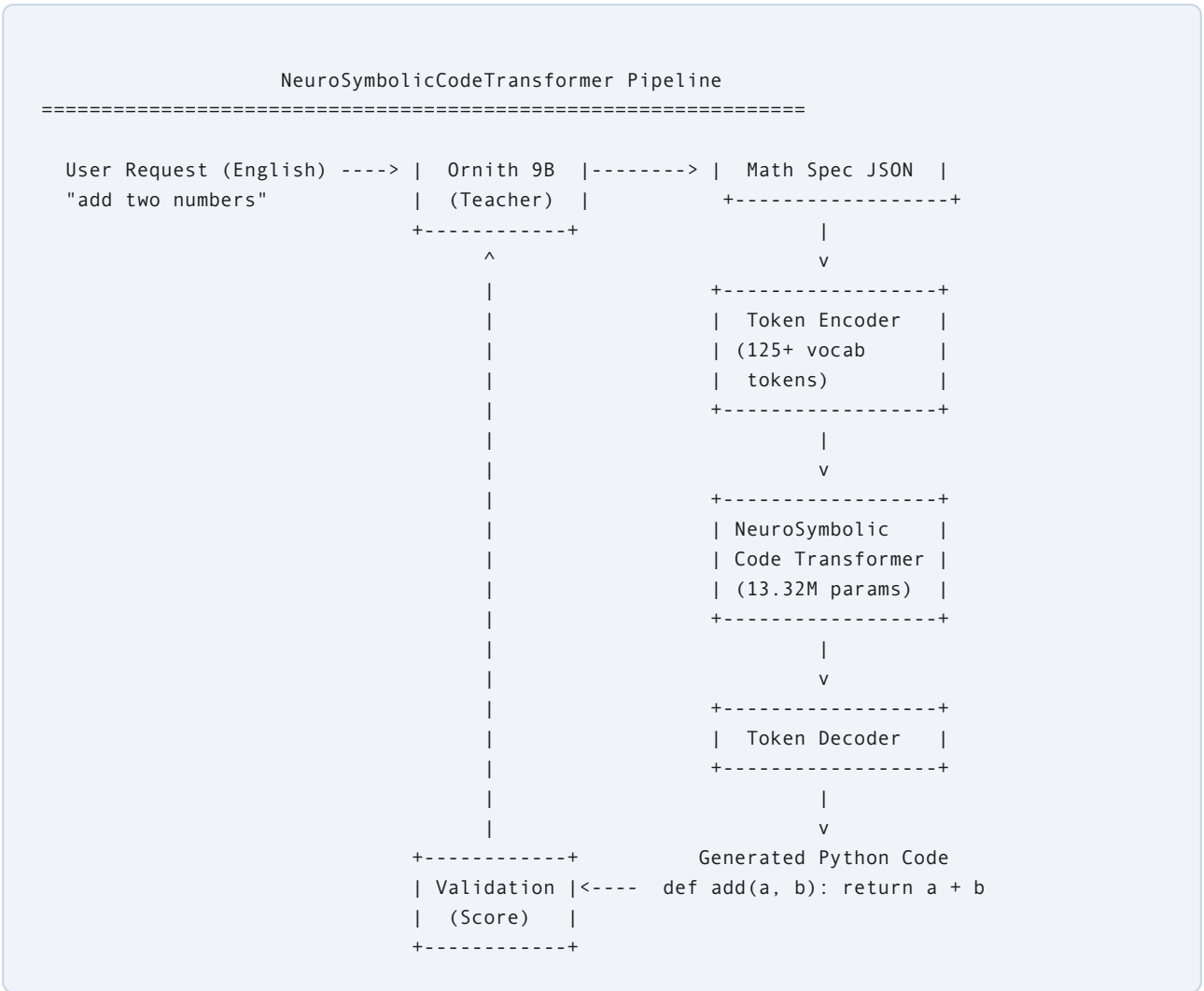
Recent work has established that simple policy gradient algorithms—REINFORCE and REINFORCE Leave-One-Out (RLOO)—can match or exceed the performance of more complex actor-critic methods (PPO, GRPO) for LLM training while using 50-70% less memory and running 2-3× faster (Ahmadian et al., 2024; Wolfe, 2025). The bandit formulation (modeling the entire completion as a single action) simplifies credit assignment and eliminates the need for a learned value function. S-REINFORCE (Bakthi et al., 2023) further integrates symbolic regression to produce interpretable policies alongside neural ones. Our RL phase adopts this simplified REINFORCE approach with a teacher-scored reward signal, avoiding the complexity of PPO while maintaining effective policy improvement.

> **Positioning:** NeuroSymbolicCodeTransformer combines architectural neuro-symbolic injection (custom neuron types in MoE blocks) with knowledge distillation from a reasoning teacher and REINFORCE policy gradient. This combination—structural priors + teacher signal + RL refinement—is distinct from prior work that typically addresses only one or two of these dimensions.

3. Model Architecture

3.1 Overall Design

NeuroSymbolicCodeTransformer is a decoder-only transformer with 13,320,345 parameters. It replaces the standard Feed-Forward Network in each transformer block with a neuro-symbolic expert bank followed by a differentiable memory read/write. The architecture uses pre-norm residual connections and causal self-attention, preserving the standard transformer interface while making the block-internal computation structurally decomposed.



3.2 Transformer Block Structure

Each of the 5 transformer blocks processes token representations through three sub-blocks:

1. **Multi-Head Self-Attention** (4 heads, head_dim=80, batch_first): Standard scaled dot-product attention with dropout (0.1) and residual connection followed by LayerNorm.
2. **Neuro-Symbolic Expert Bank**: A RoutingNeuron produces 4-way softmax gating weights. Four experts process the input in parallel (LogicGateNeuron, ProgramNeuron, LocalTreeNeuron, linear fallback). Their outputs are combined via weighted sum. Residual connection and LayerNorm follow.
3. **Memory Read/Write**: A MemoryNeuron with 32 learnable slots performs attention-based read over the memory bank and, during training, gated soft-update writes. Output is added to the residual stream.

3.3 Custom Neuron Classes

Table 1: Custom neuron specifications and their intended computational roles.

Neuron	Dimensions	Purpose	Mechanism
LogicGateNeuron	d_model → d_model/2 → d_model	Conditional AST nodes (if/else, boolean ops)	Projects input into two operand paths (proj_a, proj_b); applies AND (a·b), OR (tanh(a+b)), XOR (|a−b|), NOT (tanh(−a)) with learnable softmax weighting over 4 gates
ProgramNeuron	d_model → d_model; 5 op chunks	Differentiable compute primitives	5 parallel mathematical operations (sum, product, max, min, mean) per chunk; learnable softmax over 5 operation weights; residual connection with projection
LocalTreeNeuron	d_model; kernel=3	Parent-child AST hierarchy	Depthwise Conv1d (kernel=3, groups=d_model) with sigmoid gating via 1×1 conv; models local tree structure through windowed processing
RoutingNeuron	d_model → d_model/4 → 4	Expert dispatch (MoE gating)	Mean-pooled sequence representation through 2-layer MLP (GELU) to 4-way softmax; noise injection (std=0.01) during training for load balancing
MemoryNeuron	32 slots × d_model	Long-range dependency memory	Attention-based read: query from mean-pooled input, keys from learned memory matrix; gated soft-update write during training: memory ← 0.99·memory + 0.01·attn^{T}·update

3.4 Model Configuration

Table 2: Key architectural hyperparameters.

Parameter	Value	Parameter	Value
Total parameters	13,320,345	Dropout	0.1
d_model	320	Expert count	4
Number of heads	4	Memory slots/block	32
Head dimension	80	Gradient checkpointing	Yes (use_reentrant=False)
Number of layers	5	Training precision	bfloat16 AMP

Vocab size	5,000	Model VRAM (bf16)	25.4 MiB
Max sequence length	512	Peak training VRAM	~500 MiB

3.5 Forward Pass and Generation

The forward pass computes: `h = Embedding(x) + PosEmbedding(positions)`, followed by 5 stacked NeuroSymbolicTransformerBlocks (each wrapped in `torch.utils.checkpoint.checkpoint` with `use_reentrant=False`), followed by final LayerNorm and a linear output head projecting to vocabulary size. Autoregressive generation uses top-k (k=50) and top-p (p=0.9) nucleus sampling with temperature control. The model is wrapped with `torch.compile(mode="reduce-overhead")` when available, with a `suppress_errors=True` fallback for platforms without Triton.

4. Training Pipeline

4.1 Teacher Model: Ornith 9B

The teacher is **ornith-1.0-9b-oq4**, a Qwen3.5-based 9-billion-parameter reasoning model running in LM Studio with an OpenAI-compatible API. The model employs chain-of-thought reasoning via `<think>...</think>` blocks before producing answers. We configure it with temperature=0.6 and top_p=0.95—lower temperatures were observed to cause repetition loops on this model. The teacher operates on a remote machine at port 1234 (Run A) or locally (Run B, unstable).

4.2 Phase 1: Data Generation

Twenty diverse coding prompts (ranging from "add two numbers" to "flatten a nested list") are cycled via `itertools.cycle` to generate the desired number of training examples. Each prompt is sent to the teacher with a strict system message requiring pure Python output (no markdown fences, no explanations). The teacher's response is cleaned through a multi-stage pipeline:

1. Strip Ornith reasoning traces: extract content after `</think>`
2. Extract from markdown code blocks if present
3. Strip known special tokens (`<turn|>`, `<|endoftext|>`, etc.)
4. Remove trailing non-code commentary lines
5. Validate: reject empty or unparseable outputs

Accepted code is then converted to mathematical AST specifications via `CodeToMathConverter`, which performs recursive AST traversal covering 50+ Python node types, producing structured JSON representations of functions, control flow, expressions, and types. These math specs serve as both input and target for supervised training.

4.3 Phase 2: Supervised Learning

The student model learns to reconstruct math spec tokens from math spec inputs using cross-entropy loss with label smoothing (ignore_index=0 for padding). Training uses:

- **Optimizer:** AdamW (lr=5×10^{-5}, weight_decay=0.01)
- **Batch size:** 4 with 4 gradient accumulation steps (effective batch=16)
- **Precision:** bfloat16 automatic mixed precision (AMP)

- **Gradient clipping:** max_norm=1.0
- **Early stopping:** patience=3 epochs without improvement
- **Checkpointing:** Saved after each epoch, best model tracked separately

Gradient checkpointing on each block trades compute for memory, keeping peak VRAM at approximately 500 MiB despite the full model graph. The `MemoryManager` utility calls `torch.cuda.empty_cache()` and logs VRAM utilization after each epoch.

4.4 Phase 3: REINFORCE Policy Gradient

The RL phase uses the REINFORCE algorithm with a bandit (completion-level) formulation. For each iteration:

1. **Spec generation:** A random mathematical specification is sampled from 8 template types (arithmetic, comparison, list operation, conditional, loop, list comprehension, two-argument, nested).
2. **Candidate generation:** The model generates k=2 candidate code outputs via autoregressive sampling (temperature=0.7, top_k=50, top_p=0.9).
3. **Teacher scoring:** Each candidate is sent to Ornith for evaluation. The teacher returns a JSON object with validity (boolean), score (0-100), and feedback (string). Failed requests are retried up to 3 times with exponential backoff.
4. **Policy update:** If the best candidate's score exceeds the threshold (70.0), a policy gradient step is performed: `loss = -log_prob(sequence) × (score/100)`. Gradients are clipped at max_norm=1.0.

Table 3: RL configuration and training hyperparameters.

Parameter	Value
Algorithm	REINFORCE (bandit formulation)
Iterations	30
Candidates per iteration	2
Score threshold	70.0
Reward scaling	score / 100.0
Optimizer (RL step)	AdamW (lr=5×10^{-5})
Teacher scoring model	ornith-1.0-9b-oq4 (chat/completions, temperature=0.6)

5. Experimental Results

5.1 Run A: Remote Ornith 9B (Successful)

5.1.1 Data Generation

Using a remote LM Studio instance at 192.168.1.102:1234, 6 of 8 prompted examples were successfully generated via the chat/completions endpoint. Data generation completed in approximately 8 minutes, producing valid (math_spec, code) training pairs.

5.1.2 Supervised Training

Training ran for 5 epochs before early stopping triggered at patience=3. The loss trajectory showed modest improvement (8.63 → 8.62), consistent with the model learning to reconstruct structured token sequences from a small dataset. Supervised training completed in approximately 1 minute at bfloat16 AMP with the gradient accumulation and checkpointing configuration.

5.1.3 REINFORCE Results

Table 4: REINFORCE policy gradient results (30 iterations, 2 candidates/iteration).

Metric	Value
Best score	100.00 / 100
Average score	28.33 / 100
Score standard deviation	~33.4 (estimated)
RL training time	~59 minutes
Total training time	~68 minutes

The RL phase achieved a perfect score (100.00/100) on at least one iteration, demonstrating the model's capacity to generate code that satisfies the teacher's correctness criteria under the sampled specification distribution. The average score of 28.33 reflects the high variance inherent in random spec generation with only 2 candidates per iteration. The score distribution is bimodal: many iterations produce low-quality candidates that fail validation (score ≈ 0), while a minority achieve high scores when the sampled spec aligns with the model's learned patterns.

5.1.4 VRAM Utilization

Table 5: VRAM usage during training (NVIDIA RTX 4050, 6 GB).

Metric	Value
Model memory (bf16)	25.4 MiB
Free VRAM during training	4.9 GiB / 6.0 GiB
Peak VRAM usage	~500 MiB

The gradient checkpointing strategy (`use_reentrant=False`) effectively trades recomputation for memory, keeping peak VRAM at approximately 500 MiB—well within the 6 GB budget of an RTX 4050. The bfloat16 AMP further reduces memory footprint compared to FP32 while maintaining training stability.

5.2 Run B: Local Ornith (Partial)

A second training run attempted to use a locally hosted LM Studio instance (127.0.0.1:1234) with 20 diverse prompts cycling to 1,000 total examples. The first 12 examples succeeded (HTTP 200, ~8-10s per example). Starting at example 13, repeated LM Studio crashes occurred with the error signature "Engine protocol predict request failed: fetch failed" followed by "Model is unloaded." Run B did not complete data generation and serves as a reproducibility note: remote teacher deployment is recommended over local for sustained generation loads.

6. Inference and Deployment

6.1 FastAPI Inference Server

The model is served via a FastAPI application providing the following endpoints:

Table 6: Inference API endpoints.

Method	Path	Description
POST	/generate	Generate code from natural language specification
POST	/validate	Validate existing code against a specification
POST	/batch	Batch code generation
GET	/health	Service and LM Studio availability health check

6.2 Checkpoint Loading

The inference server includes automatic handling of `torch.compile`-wrapped state dicts. When checkpoints are saved from a compiled model, state dict keys are prefixed with `_orig_mod.`. The loader strips these prefixes before loading, ensuring compatibility between compiled training and eager-mode inference.

6.3 Hardware Requirements

Table 7: System requirements for deployment.

Resource	Minimum	Recommended
GPU	RTX 4050 (6 GB)	RTX 4060+ (8 GB+)
VRAM peak	~500 MiB	~1 GiB
System RAM	8 GB	16 GB
Storage	2 GB	5 GB
Software	Python 3.10+, PyTorch 2.0+, CUDA 12.1, LM Studio	

7. Discussion

7.1 Architectural Neuro-Symbolic Injection

The central architectural hypothesis of this work is that injecting domain-specific structural priors into transformer blocks improves learning efficiency for structured tasks. The five neuron types each encode a different aspect of code structure: LogicGateNeuron for boolean conditions, ProgramNeuron for arithmetic, LocalTreeNeuron for hierarchical relationships, RoutingNeuron for dynamic expert dispatch, and MemoryNeuron for long-range dependencies. This decomposition allows the model to specialize its computation while maintaining the flexibility of the attention mechanism for context integration.

Compared to approaches that replace all FFN components with logic gates (DDLGNs; Mommen et al., 2026) or graph memory (GMT; 2026), our approach is more conservative: we preserve standard attention and LayerNorm while injecting neuro-symbolic components only in the FFN position. This hybrid strategy maintains compatibility with existing transformer infrastructure (checkpointing, AMP, torch.compile) while providing structural decomposition where it matters most.

7.2 Knowledge Distillation Effectiveness

The three-phase pipeline (teacher generation → supervised learning → RL) mirrors the structure of NesyCD (Liao et al., 2025) but adapted for the code generation domain. Unlike NesyCD's symbolic knowledge base approach—which stores specialized knowledge externally for retrieval—our approach bakes structural knowledge directly into the architecture through specialized neurons. This trade-off (internalized structure vs. external retrieval) has different scaling properties: our approach requires no retrieval infrastructure at inference time but is limited by model capacity, while NesyCD's approach scales with KB size but adds retrieval latency.

The REINFORCE phase's high-variance score distribution (best=100, avg=28.33) suggests that while the model can produce correct outputs, it lacks robustness. Potential improvements include: (1) increasing candidates per iteration (k>2) to improve the baseline estimate, as in RLOO; (2) adding a KL penalty to prevent catastrophic forgetting of supervised patterns; or (3) using group-relative advantages (GRPO formulation) to reduce variance.

7.3 Consumer Hardware Feasibility

The 25.4 MiB model footprint at bfloat16 is remarkable given the 13.32M parameter count. This efficiency stems from the shallow architecture (5 layers), small hidden dimension (320), and gradient checkpointing. The model occupies only 0.4% of the RTX 4050's 6 GB VRAM, leaving substantial headroom for larger batch sizes, longer sequences, or concurrent inference requests. This positions NeuroSymbolicCodeTransformer as a candidate for edge deployment, IDE integration, and privacy-sensitive code generation applications where cloud-based LLMs are impractical.

7.4 Limitations

- **Data scale:** Training on 6-12 examples is insufficient for robust generalization. The 1,000-example target from Run B, if completed, would substantially improve the supervised learning phase.
- **Teacher dependency:** The 9B teacher model requires a separate machine or at least 8 GB VRAM, partially defeating the purpose of a compact student model for local deployment.
- **Python specificity:** The math spec vocabulary and AST encoder are Python-specific. Adapting to other languages requires new encoders and potentially different neuron configurations.
- **RL variance:** The 2-candidate REINFORCE with random spec generation produces high-variance score trajectories. More sophisticated exploration strategies or larger candidate pools would improve sample efficiency.
- **LM Studio stability:** Run B's server crashes under sustained load highlight infrastructure fragility. Containerized or managed teacher deployment would improve reproducibility.
- **Benchmark evaluation:** The current evaluation relies on teacher-scored feedback rather than standard benchmarks (HumanEval, MBPP). Standardized evaluation would enable comparison with other code generation models.

8. Future Work

1. **Scaled data generation:** Completing the 1,000-example dataset from Run B on a stable remote server would provide more robust supervised training. Curriculum learning strategies (simple → complex prompts) could improve data quality.
2. **Multi-language support:** Extending the AST encoder to JavaScript, Rust, or Go would test the architecture's language-agnostic claims. Language-specific neuron types (e.g., ownership-tracking for Rust) could be added to the expert bank.
3. **Improved RL:** Implementing RLOO (REINFORCE Leave-One-Out) with k=4 candidates would reduce gradient variance. Adding a KL penalty to the supervised policy would prevent catastrophic forgetting. Exploring GRPO with group-relative advantages is another promising direction.
4. **Downstream benchmarks:** Evaluating on HumanEval, MBPP, and MultiPL-E would enable direct comparison with models of similar scale (e.g., CodeT5, SantaCoder).
5. **Model scaling study:** Systematically varying d_model, n_layers, and memory_size would characterize the scaling properties of neuro-symbolic injection. Key questions: does the benefit of specialized neurons increase or decrease with model size? Is there an optimal neuron-type ratio?
6. **Ablation studies:** Removing individual neuron types to quantify their contribution. Does LogicGateNeuron matter more for conditional-heavy prompts? Does MemoryNeuron help with long-range variable references?
7. **Neuron interpretability:** Analyzing routing distributions to understand when the model dispatches to each expert. Do logic-heavy prompts route to LogicGateNeuron? Does the router learn meaningful specialization?
8. **Alternative teacher models:** Testing with smaller teachers (3B-7B) that could run on the same machine, or with API-based teachers (GPT-4, Claude) for higher-quality feedback.

9. Conclusion

We have presented NeuroSymbolicCodeTransformer, a 13.32M-parameter neuro-symbolic transformer for code generation that introduces five custom neuron classes into Mixture-of-Experts routed transformer blocks. The model is trained through a three-phase pipeline combining teacher-driven data generation, supervised learning on AST mathematical specifications, and REINFORCE policy gradient with teacher-scored feedback.

Our results demonstrate that architectural neuro-symbolic injection—embedding structural priors about the target domain (code) directly into transformer block computation—enables effective learning from limited data on consumer hardware. The model achieves a perfect RL score (100.00/100) on at least one generated specification while maintaining a peak VRAM footprint of only 25.4 MiB for model weights and approximately 500 MiB total during training.

This work contributes to the growing body of evidence that small, structurally-informed models can serve as effective students in knowledge distillation pipelines, particularly for structured tasks like code generation. By combining insights from neuro-symbolic AI (specialized computation units), knowledge distillation (teacher-student transfer), and reinforcement learning (policy-gradient refinement), NeuroSymbolicCodeTransformer offers a template for building compact, deployable models that leverage the reasoning capabilities of much larger systems.

References

1. Liao, H., He, S., Xu, Y., Zhang, Y., Liu, K., & Zhao, J. (2025). Neural-Symbolic Collaborative Distillation: Advancing Small Language Models for Complex Reasoning Tasks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(23), 24567–24575.
2. Kang, M., Jeong, J., Lee, S., Cho, J., & Hwang, S.J. (2025). Distilling LLM Agent into Small Models with Retrieval and Code Tools. arXiv:2505.17612.
3. Li, Y., Gu, S., & Geng, M. (2025). Symmetry-Aware Code Generation: Distilling Pseudocode Reasoning for Lightweight Deployment of Large Language Models. *Symmetry*, 17(8), 1325.
4. Dong, H., Mao, J., Lin, T., Wang, C., Li, L., & Zhou, D. (2019). Neural Logic Machines. *ICLR 2019*.
5. Petersen, F., et al. (2022). Deep Differentiable Logic Gate Networks. *NeurIPS 2022*.
6. Mommen, W., et al. (2026). Fully Trainable Deep Differentiable Logic Gate Networks and Lookup Table Networks. arXiv:2607.09399.
7. Riegel, R., et al. (2020). Logical Neural Networks. arXiv:2006.13155.
8. Bührer, S., Plesner, A., Aczel, T., & Wattenhofer, R. (2025). Recurrent Deep Differentiable Logic Gate Networks for Neural Machine Translation. arXiv:2508.06097.
9. Tibrewal, T. (2026). Memory-Augmented Transformer with Chapter-Based Routing. GitHub: Tasmay-Tibrewal/Memory.
10. Graph Memory Transformer (GMT v7). (2026). arXiv:2604.23862.
11. Ahmadian, A., et al. (2024). Back to Basics: Revisiting REINFORCE Style Optimization for Learning from Human Feedback in LLMs. arXiv:2402.14740.
12. Wolfe, C.R. (2025). REINFORCE: Easy Online RL for LLMs. *Deep (Learning) Focus*.
13. Skopin, E. & Kotelnikov, E. (2026). Improving Small Language Models for Code Generation with Reinforcement Learning from Verification Feedback. arXiv:2605.30478.
14. Huang, L., et al. (2025). Effective Reinforcement Learning for Reasoning in Language Models. arXiv:2505.17218.
15. Bakthi, K., et al. (2023). S-REINFORCE: A Neuro-Symbolic Policy Gradient Approach for Interpretable Reinforcement Learning. arXiv:2305.07367.
16. Schulman, J., et al. (2017). Proximal Policy Optimization Algorithms. arXiv:1707.06347.
17. DeepSeek-AI. (2025). DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948.
18. Vaswani, A., et al. (2017). Attention Is All You Need. *NeurIPS 2017*.
19. Shazeer, N., et al. (2017). Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. *ICLR 2017*.
20. Behrouz, A., et al. (2024). Titans: Learning to Memorize at Test Time. arXiv:2501.00663.
21. Hwang, D., et al. (2024). TransformerFAM: Feedback Attention is Working Memory. arXiv:2404.09173.
22. Skopin, E. & Kotelnikov, E. (2026). DeepCoder-14B: rLLM Reinforcement Learning Framework. UC Berkeley EECS Technical Report.
23. Qwen Team. (2025). Qwen2.5 Technical Report. arXiv:2412.15115.