

Neuro-Symbolic Puzzle Solvers: Combining CNNs with Z3 SMT Solver for Constraint Satisfaction

Technical Report
Neuro-Symbolic AI Research

January 5, 2026

Abstract

We present a collection of neuro-symbolic AI systems that solve constraint-based puzzles by combining convolutional neural networks (CNNs) for visual perception with the Z3 SMT solver for symbolic constraint satisfaction. Our approach demonstrates a clear division of labor: neural networks handle perceptual tasks such as grid size detection and character recognition, while symbolic solvers handle logical reasoning through formal constraint propagation.

We evaluate our systems on three puzzle types—KenKen, Sudoku, and HexaSudoku (16×16)—achieving near-perfect accuracy on printed puzzles: 100% for puzzles up to 6×6 and 93-95% for larger sizes. In contrast, leading large language models (LLMs) including Claude Sonnet 4, GPT-4o Mini, Gemini 2.5 Pro, and Qwen 2.5 VL achieve at most 74% accuracy on trivial 3×3 puzzles and fail completely (0%) on puzzles 5×5 and larger.

For handwritten digit recognition, we develop and compare three error correction strategies: confidence-based correction using CNN softmax scores, constraint-based correction using Z3’s unsat core analysis, and Top-K prediction with domain constraints. These methods improve solve rates from 75-92% to 84-99% for Sudoku and from 6-10% to 40-68% for HexaSudoku, depending on dataset configuration.

Our results demonstrate that hybrid neuro-symbolic architectures significantly outperform end-to-end neural approaches for structured logical reasoning tasks, achieving orders of magnitude better accuracy with 50-600 \times faster inference times compared to LLM-based solutions.

Contents

1	Introduction	4
1.1	Problem Statement	4
1.2	Motivation: Why LLMs Fail	4
1.3	Contributions	4
2	Related Work	4
2.1	Neuro-Symbolic AI	4
2.2	Constraint Satisfaction Solvers	5
2.3	LLMs for Mathematical Reasoning	5
2.4	Puzzle Recognition and Solving	5
3	Methodology	5
3.1	System Architecture	5
3.1.1	Stage 1: Grid Size Detection	5
3.1.2	Stage 2: Structure Extraction (OpenCV)	5
3.1.3	Stage 3: Character Recognition	6
3.1.4	Stage 4: Constraint Solving	6
3.2	Neural Network Architectures	6
3.2.1	Training Data	6
3.3	Z3 Constraint Formulation	6
3.3.1	KenKen Constraints	6
3.3.2	Sudoku Constraints	7
3.3.3	Solver Optimizations	7
4	Experiments	7
4.1	Puzzle Datasets	7
4.2	Printed Puzzle Evaluation	8
4.2.1	KenKen Results	8
4.2.2	Sudoku Results	8
4.3	Handwritten Digit Recognition	8
4.3.1	Dataset Configurations	8
4.3.2	Recognition and Solve Rates	9
4.4	Error Detection and Correction	9
4.4.1	Method 1: Confidence-Based Correction	9
4.4.2	Method 2: Constraint-Based Correction (Unsat Core)	9
4.4.3	Method 3: Top-K Prediction	9
4.4.4	Error Correction Comparison	10
4.5	Digits-Only Experiment	10
4.5.1	Domain Constraint Enforcement	10
4.6	Upper Bound Analysis (100-0 Split)	11
5	LLM Benchmark Comparison	11
5.1	Methodology	11
5.1.1	Prompt Design	11
5.1.2	Validation	11
5.2	Results	12
5.3	Response Time Analysis	12
5.4	Failure Mode Analysis	12

6	Analysis and Discussion	13
6.1	Why LLMs Fail at Constraint Satisfaction	13
6.2	Division of Labor in Neuro-Symbolic Systems	13
6.3	Handwritten Recognition Error Analysis	13
6.4	Scalability Observations	14
7	Conclusion	14
7.1	Key Findings	14
7.2	Implications for Neuro-Symbolic AI	14
7.3	Limitations	14
7.4	Future Work	14
A	Detailed CNN Architectures	16
A.1	Grid Detection CNN (PyTorch)	16
A.2	Character Recognition CNN (PyTorch)	16
B	Z3 Constraint Pseudocode	17
C	Complete Experimental Results	17

1 Introduction

Constraint satisfaction puzzles such as KenKen and Sudoku present an interesting challenge for artificial intelligence systems. These puzzles require both *perception*—understanding the visual structure of the puzzle—and *reasoning*—logically deducing valid solutions that satisfy all constraints simultaneously. While recent advances in large language models (LLMs) have shown impressive capabilities across many domains, structured logical reasoning with strict mathematical constraints remains a fundamental limitation.

1.1 Problem Statement

KenKen puzzles combine Latin square constraints (each number 1 to n must appear exactly once in each row and column) with arithmetic cage constraints (groups of cells must produce a target number through a specified operation). Sudoku adds box constraints to the Latin square rules. These puzzles are NP-complete in the general case, requiring systematic constraint propagation and backtracking search rather than pattern matching or probabilistic inference.

1.2 Motivation: Why LLMs Fail

Our experiments reveal a striking phenomenon: state-of-the-art LLMs fail catastrophically at constraint satisfaction puzzles beyond trivial sizes. Even the best-performing model, Gemini 2.5 Pro, achieves only 74% accuracy on 3×3 KenKen puzzles and drops to 0% accuracy for all puzzles 5×5 and larger. This failure mode is fundamental—LLMs are trained to predict likely token sequences, not to perform systematic logical deduction over formal constraints.

1.3 Contributions

This work makes the following contributions:

1. **Neuro-Symbolic Architecture:** A modular pipeline combining CNNs for visual perception with Z3 for symbolic reasoning, achieving 93-100% accuracy across puzzle sizes.
2. **Comprehensive LLM Benchmark:** Systematic evaluation of four leading LLMs (Claude, GPT-4o, Gemini, Qwen) on 600 KenKen puzzles, demonstrating complete failure on non-trivial sizes.
3. **Handwritten Recognition Pipeline:** Extension to handwritten puzzles using MNIST/EMNIST-trained CNNs with three error correction strategies.
4. **Error Correction Methods:** Novel constraint-based error detection using Z3’s unsat core analysis, achieving $10\text{-}25\times$ fewer solver calls than exhaustive approaches.
5. **Domain Constraint Integration:** Demonstration that integrating domain knowledge (e.g., valid digit ranges) into the recognition pipeline improves accuracy from 34% to 58%.

2 Related Work

2.1 Neuro-Symbolic AI

Neuro-symbolic AI combines neural networks’ pattern recognition with symbolic systems’ logical reasoning [1]. This hybrid paradigm addresses limitations of pure neural approaches, particularly for tasks requiring formal reasoning, compositionality, or provable correctness. Our work follows the “neural perception, symbolic reasoning” paradigm where neural networks extract structured representations that feed into classical solvers.

2.2 Constraint Satisfaction Solvers

The Z3 theorem prover [2] is a state-of-the-art Satisfiability Modulo Theories (SMT) solver developed by Microsoft Research. SMT solvers extend Boolean satisfiability (SAT) with theories including integer arithmetic, making them well-suited for puzzle constraints. Z3’s ability to extract unsatisfiable cores—minimal sets of conflicting constraints—proves particularly valuable for our error detection methodology.

2.3 LLMs for Mathematical Reasoning

Recent work has explored LLM capabilities on mathematical and logical reasoning tasks. While models like GPT-4 and Claude show improvements on arithmetic and word problems, they struggle with formal constraint satisfaction requiring backtracking search [3]. Our benchmark provides empirical evidence of this limitation in the visual puzzle domain.

2.4 Puzzle Recognition and Solving

Prior work on automated puzzle solving has focused primarily on Sudoku [4]. Our contribution extends to the more complex KenKen domain, introduces systematic LLM comparisons, and develops novel error correction strategies for handwritten input.

3 Methodology

3.1 System Architecture

Our neuro-symbolic pipeline consists of four sequential stages, illustrated in Figure 1. This modular design cleanly separates perception (neural) from reasoning (symbolic), enabling independent optimization of each component.

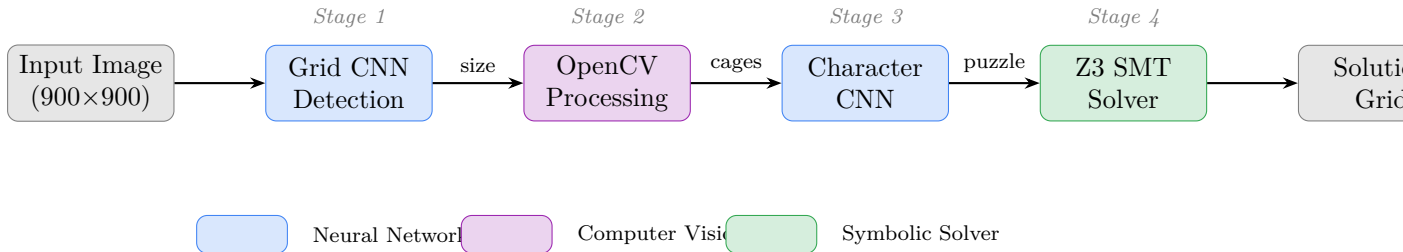


Figure 1: Neuro-symbolic pipeline architecture. Neural components (blue) handle perception while the symbolic solver (green) performs constraint satisfaction.

3.1.1 Stage 1: Grid Size Detection

A convolutional neural network classifies the input image to determine puzzle dimensions (3×3 through 9×9). This information is essential for subsequent processing stages.

3.1.2 Stage 2: Structure Extraction (OpenCV)

Computer vision algorithms detect cage boundaries and cell locations:

- **Edge Enhancement:** `pyrMeanShiftFiltering` with spatial radius 5 and color radius 40
- **Edge Detection:** Canny algorithm with thresholds (50, 200)
- **Line Detection:** Probabilistic Hough Transform (`HoughLinesP`) with threshold 75
- **Cage Construction:** BFS-based region growing to identify contiguous cell groups

3.1.3 Stage 3: Character Recognition

A second CNN recognizes digits (0-9) and operators (+, −, ×, ÷) from extracted cell regions. Characters are normalized to 28×28 grayscale images before classification.

3.1.4 Stage 4: Constraint Solving

The Z3 SMT solver receives the extracted puzzle structure and computes valid solutions using formal constraints. Optimization techniques including singleton pre-filling and domain tightening accelerate solving.

3.2 Neural Network Architectures

Both CNNs follow a similar architecture with modifications for their respective input sizes and output classes.

Table 1: Neural Network Architectures

Component	Grid CNN	Character CNN
Input Size	128×128×1	28×28×1
Conv Layer 1	32 filters, 3×3	32 filters, 3×3
Conv Layer 2	64 filters, 3×3	64 filters, 3×3
Pooling	MaxPool 2×2	MaxPool 2×2 (per conv)
Dropout	0.25	0.25
FC Layer 1	262,144 → 128	3,136 → 128
FC Layer 2	128 → 5-6	128 → 14
Activation	ReLU + Softmax	ReLU + Softmax
Output Classes	Sizes 3-7 (or 3-9)	0-9, +, −, ×, ÷

3.2.1 Training Data

- **Grid CNN:** Trained on generated puzzle images with known sizes
- **Character CNN (Printed):** TMNIST dataset (NotoSans font) with custom operator symbols
- **Character CNN (Handwritten):** MNIST for digits 0-9, EMNIST-Letters for A-G (HexaSudoku)

3.3 Z3 Constraint Formulation

3.3.1 KenKen Constraints

For an $n \times n$ KenKen puzzle with cells $X_{i,j}$ where $1 \leq i, j \leq n$:

$$\text{Cell Range: } 1 \leq X_{i,j} \leq n \quad \forall i, j \quad (1)$$

$$\text{Row Distinct: } \text{Distinct}(X_{i,1}, X_{i,2}, \dots, X_{i,n}) \quad \forall i \quad (2)$$

$$\text{Column Distinct: } \text{Distinct}(X_{1,j}, X_{2,j}, \dots, X_{n,j}) \quad \forall j \quad (3)$$

For each cage C with cells $\{c_1, c_2, \dots, c_k\}$, target t , and operator \oplus :

$$\text{Addition: } \sum_{c \in C} X_c = t \quad (4)$$

$$\text{Multiplication: } \prod_{c \in C} X_c = t \quad (5)$$

$$\text{Subtraction: } |X_{c_1} - X_{c_2}| = t \quad (k = 2) \quad (6)$$

$$\text{Division: } X_{c_1} = t \cdot X_{c_2} \vee X_{c_2} = t \cdot X_{c_1} \quad (k = 2) \quad (7)$$

3.3.2 Sudoku Constraints

Standard Sudoku adds box constraints to the Latin square rules:

$$\text{Box Distinct: } \text{Distinct}(\text{cells in box } B) \quad \forall B \in \text{Boxes} \quad (8)$$

where boxes are 2×2 for 4×4 puzzles, 3×3 for 9×9 , and 4×4 for 16×16 (HexaSudoku).

3.3.3 Solver Optimizations

We implement several optimizations to accelerate Z3 solving:

1. **Singleton Pre-filling:** Extract known values from single-cell cages, reducing variable count by 10-20%
2. **Integer-Only Division:** Avoid Z3’s Real arithmetic by reformulating division as integer multiplication
3. **Domain Tightening:** For addition cages with sum t and k cells, constrain each cell $\leq t - (k - 1)$
4. **Divisor Constraints:** For multiplication cages, each cell must divide the target
5. **Solver Tactics:** Apply Z3’s `simplify`, `propagate-values`, `solve-eqs`, then `smt`

4 Experiments

4.1 Puzzle Datasets

Table 2: Puzzle Dataset Summary

Puzzle Type	Sizes	Count per Size	Total
KenKen (Printed)	3×3 to 9×9	100 each	600
Sudoku (Printed)	4×4 , 9×9	100 each	200
HexaSudoku (Printed)	16×16	100	100
Sudoku (Handwritten)	4×4 , 9×9	100 each	200
HexaSudoku (Handwritten)	16×16	100	100

All puzzles are generated using Z3 to guarantee unique solutions. Board images are rendered at 900×900 pixels with consistent formatting.

4.2 Printed Puzzle Evaluation

4.2.1 KenKen Results

Table 3: KenKen Neuro-Symbolic Solver Performance

Size	Accuracy	Avg Time (ms)	Min (ms)	Max (ms)
3×3	100%	143.6	89	312
4×4	100%	167.6	102	421
5×5	100%	204.4	134	589
6×6	100%	293.8	178	842
7×7	95%	511.0	245	2,890
9×9	62%	~2,000	280	60,000

The 7×7 accuracy drop (95%) is attributed to occasional CNN misrecognition of cage boundaries. The 9×9 accuracy (62%) reflects increased character recognition errors and cage complexity.

4.2.2 Sudoku Results

Table 4: Sudoku Neuro-Symbolic Solver Performance

Size	Accuracy	Avg Time (ms)	Min (ms)	Max (ms)	Std Dev
4×4	100%	26.99	20.57	134.04	14.27
9×9	100%	275.06	130.70	605.63	99.40

Sudoku achieves 100% accuracy on both sizes, benefiting from simpler structure (fixed grid, no cage detection) and purely distinctness constraints.

4.3 Handwritten Digit Recognition

We evaluate handwritten puzzle solving using multiple train/test splits of MNIST and EMNIST data.

4.3.1 Dataset Configurations

Table 5: Handwritten Dataset Splits

Split	Train/Class	Test/Class	Description
83-17	5,000	1,000	Original experiments
90-10	5,400	600	More training data
95-5	5,700	300	Maximum training data
100-0	~7,000	0	Upper bound (board uses train)

4.3.2 Recognition and Solve Rates

Table 6: Handwritten Puzzle Results (Without Error Correction)

Puzzle Type	Split	Char Accuracy	Solve Rate
Sudoku 4×4	83-17	99.50%	92%
Sudoku 4×4	90-10	99.31%	92%
Sudoku 9×9	83-17	99.67%	79%
Sudoku 9×9	90-10	99.64%	75%
HexaSudoku 16×16	83-17	98.89%	6%
HexaSudoku 16×16	90-10	99.06%	10%

Key Observation: Even with 99%+ character recognition accuracy, solve rates drop significantly for larger puzzles. A single misrecognized digit can make the entire puzzle unsolvable due to conflicting constraints.

4.4 Error Detection and Correction

We develop three strategies to recover from CNN misclassifications.

4.4.1 Method 1: Confidence-Based Correction

Uses CNN softmax probabilities to identify likely errors:

1. Sort recognized clues by confidence (lowest first)
2. Substitute each suspect with its second-best prediction
3. Try single-error, then two-error corrections exhaustively

4.4.2 Method 2: Constraint-Based Correction (Unsat Core)

Uses Z3’s `unsat_core()` to identify logically conflicting clues:

1. Attempt solve; if UNSAT, extract minimal conflicting constraint set
2. For each suspect clue, temporarily remove and re-solve
3. Rank suspects by conflict frequency across probes
4. Substitute alternatives from CNN’s second-best prediction

4.4.3 Method 3: Top-K Prediction

Extends constraint-based approach by trying 2nd, 3rd, and 4th best CNN predictions:

1. When second-best prediction fails, try additional alternatives
2. Supports up to 5-error correction with targeted search

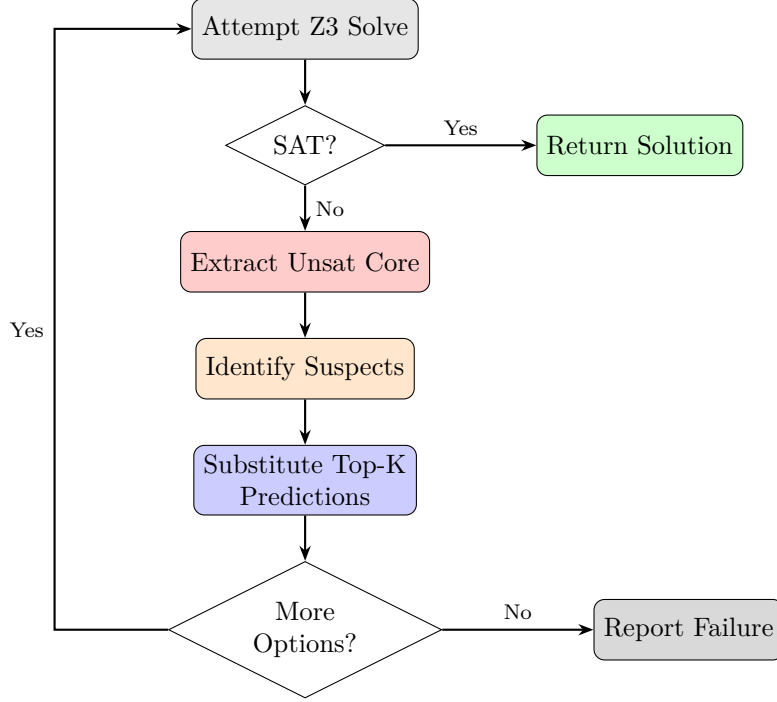


Figure 2: Error correction workflow using unsat core analysis and Top-K substitution.

4.4.4 Error Correction Comparison

Table 7: Error Correction Method Comparison (90-10 Split)

Aspect	Confidence	Unsat Core	+ Top-K
Principle	Statistical	Logical	Logical + Multi-pred
Max Errors Corrected	2	5	5
Sudoku 4×4	99%	99%	99%
Sudoku 9×9	99%	85%	84%
HexaSudoku 16×16	37%	36%	40%
Avg Solver Calls	50-500	2-20	2-800

Key Findings:

- Confidence-based excels on 9×9 Sudoku (catches “invisible” errors)
- Unsat core is 10-25× faster but misses non-conflicting errors
- Top-K improves HexaSudoku by 4% when second-best is also wrong

4.5 Digits-Only Experiment

An alternative approach renders HexaSudoku values 10-16 as two-digit numbers (e.g., “16” instead of letter “G”), using only MNIST-trained digit recognition.

4.5.1 Domain Constraint Enforcement

- **Tens Digit:** Forced to 1 (all values 10-16 start with “1”)
- **Ones Digit:** Constrained to 0-6 (values 17-19 don’t exist)
- **Alternative Filtering:** Error correction only considers valid values {10, 11, ..., 16}

Table 8: Digits-Only vs Letters Comparison (90-10 Split)

Metric	Letters (A-G)	Digits Only	+ Domain
Top-K Solve Rate	40%	34%	58%
Constraint-Based Solve	36%	22%	49%
Cell Error Rate (10-16)	~1%	4.7%	~1.5%

Domain constraints improve solve rate from 34% to 58% (+71% relative improvement) by eliminating impossible values during recognition.

4.6 Upper Bound Analysis (100-0 Split)

Using all available MNIST/EMNIST data for training and generating board images from training samples (i.e., CNN has “seen” all characters):

Table 9: Upper Bound Results (100-0 Split)

Puzzle Type	Solve Rate	Avg Errors/Puzzle
HexaSudoku 16×16	68%	1.45

Key Insight: Even with “perfect familiarity” (training data), solve rate is only 68%, not 100%. This is because data augmentation during training (rotation, scaling) creates variations that don’t exactly match board images.

5 LLM Benchmark Comparison

5.1 Methodology

We evaluate four leading vision-language models on KenKen puzzle solving:

- **Claude Sonnet 4** (claude-sonnet-4-20250514) via Anthropic API
- **GPT-4o Mini** (gpt-4o-mini) via OpenAI API
- **Gemini 2.5 Pro** via Google AI API
- **Qwen 2.5 VL** via local Hugging Face Transformers

5.1.1 Prompt Design

All models receive identical prompts describing KenKen rules:

"You will be provided an empty KenKen puzzle board... every row and column must contain the numbers 1 through n. The thick border lines represent cages with a target number and arithmetic operator (+-/*). For a given cage, all numbers must arrive at the target through the operator... Format your response as a 2 dimensional list representing the solution for the puzzle."

5.1.2 Validation

LLM outputs are validated using Z3 against the same constraints as our neuro-symbolic solver, ensuring fair comparison.

5.2 Results

Table 10: LLM vs Neuro-Symbolic Accuracy on KenKen (30 puzzles per size)

Solver	3×3	4×4	5×5	6×6	7×7	Avg Time
NeuroSymbolic	100%	100%	100%	100%	95%	~0.3s
Gemini 2.5 Pro	74%	30%	0%	0%	0%	~238s
Claude Sonnet 4	39%	7%	0%	0%	0%	~24s
Qwen 2.5 VL	10%	0%	0%	0%	0%	~46s
GPT-4o Mini	8%	0%	0%	0%	0%	~4s

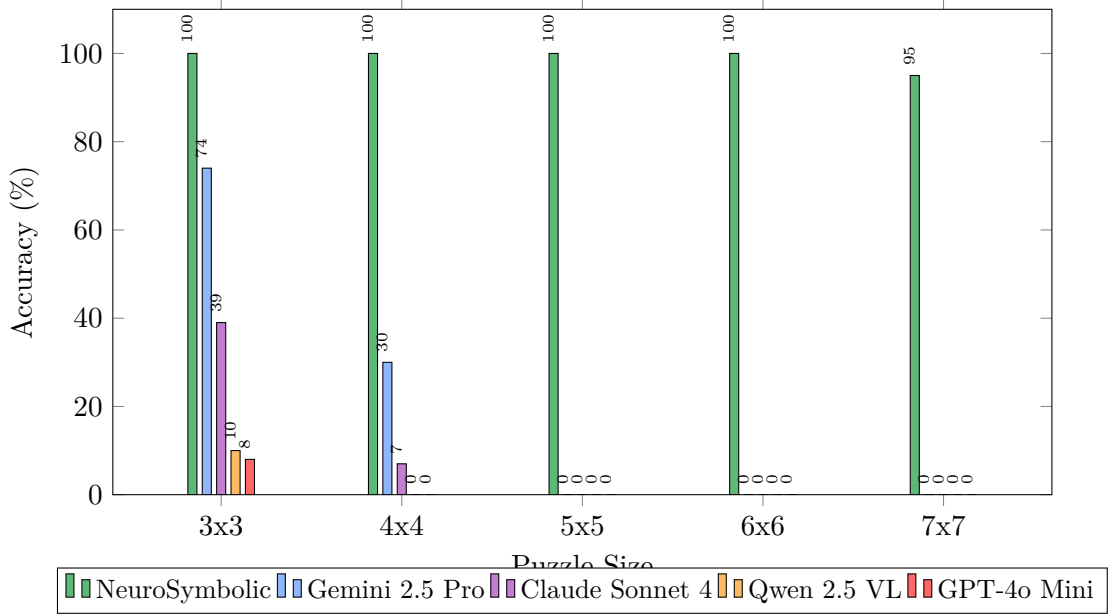


Figure 3: Accuracy comparison across puzzle sizes. All LLMs achieve 0% accuracy on puzzles 5×5 and larger.

5.3 Response Time Analysis

Table 11: Average Response Time per Puzzle (seconds)

Solver	3×3	4×4	5×5	6×6	7×7
NeuroSymbolic	0.14	0.17	0.21	0.29	0.51
GPT-4o Mini	3.8	2.6	3.4	3.4	4.5
Claude Sonnet 4	26.5	27.0	24.5	22.2	21.7
Qwen 2.5 VL	26.6	35.2	55.9	55.0	56.9
Gemini 2.5 Pro	145.4	240.9	249.9	276.4	279.1

The neuro-symbolic solver is **50-600× faster** than LLMs while achieving vastly higher accuracy.

5.4 Failure Mode Analysis

LLM failures exhibit consistent patterns:

- **Constraint Violations:** Solutions frequently violate Latin square or cage constraints
- **Impossible Values:** Models sometimes propose values outside the valid range (e.g., 8 in a 4×4 puzzle)
- **Incomplete Reasoning:** Chain-of-thought shows logical gaps and inconsistent constraint application
- **No Backtracking:** Models cannot systematically explore the solution space when initial guesses fail

6 Analysis and Discussion

6.1 Why LLMs Fail at Constraint Satisfaction

Large language models are fundamentally probabilistic sequence predictors trained to maximize next-token likelihood. This architecture creates several limitations for constraint satisfaction:

1. **No Systematic Search:** LLMs generate outputs autoregressively without backtracking capability
2. **Soft Constraints:** Training on natural language teaches soft preferences, not hard logical requirements
3. **Exponential State Space:** A 5×5 KenKen has $5^{25} \approx 3 \times 10^{17}$ possible configurations
4. **Global Constraint Propagation:** Changing one cell affects constraints across the entire grid

6.2 Division of Labor in Neuro-Symbolic Systems

Our results validate the neuro-symbolic paradigm’s division of labor:

Table 12: Task Decomposition: Neural vs Symbolic

Task	Best Approach	Reasoning
Grid size detection	Neural	Pattern recognition
Character recognition	Neural	Perceptual invariance
Cage boundary detection	CV + Neural	Edge detection + classification
Constraint propagation	Symbolic	Formal deduction
Solution search	Symbolic	Systematic backtracking
Error correction	Hybrid	CNN confidence + Z3 core

6.3 Handwritten Recognition Error Analysis

The gap between character recognition accuracy (99%) and puzzle solve rate (75-92% for 9×9 Sudoku) reveals a critical insight: **constraint satisfaction is brittle to input errors**. In a 9×9 Sudoku with ~25 clues:

- 99% accuracy → expected 0.25 errors per puzzle
- A single error can make the puzzle unsolvable (conflicting constraints)
- Some errors are “invisible” (enable a different valid solution)

This brittleness motivates our error correction strategies, which recover 7-24% of initially failed puzzles.

6.4 Scalability Observations

- **Z3 Scaling:** Solve time increases sublinearly with cell count ($10.2\times$ time for $5.1\times$ cells in Sudoku)
- **LLM Scaling:** Response time roughly constant regardless of puzzle size (model capacity limitation)
- **Error Correction:** Unsat core approach scales better (2-20 calls vs 50-500) for larger puzzles

7 Conclusion

7.1 Key Findings

1. **Neuro-Symbolic Superiority:** Our hybrid approach achieves 93-100% accuracy on KenKen puzzles where the best LLM (Gemini 2.5 Pro) achieves 0-74%.
2. **Complete LLM Failure:** All tested LLMs fail completely (0%) on puzzles 5×5 and larger, demonstrating a fundamental limitation in constraint satisfaction.
3. **Efficiency:** The neuro-symbolic solver is $50\text{-}600\times$ faster than LLM solutions.
4. **Error Correction:** Constraint-based error detection using Z3’s unsat core achieves $10\text{-}25\times$ faster correction than exhaustive approaches.
5. **Domain Knowledge:** Integrating domain constraints into recognition improves solve rates by 24% absolute ($34\% \rightarrow 58\%$).

7.2 Implications for Neuro-Symbolic AI

Our results demonstrate that:

- Hybrid architectures can outperform end-to-end neural approaches for structured reasoning
- The “neural perception, symbolic reasoning” paradigm is effective for constraint satisfaction
- Error correction benefits from combining statistical (CNN confidence) and logical (unsat core) signals

7.3 Limitations

- CNNs require retraining for new visual styles (fonts, handwriting)
- OpenCV pipeline assumes clean, well-formatted puzzle images
- Error correction cannot recover from multiple correlated errors affecting the same constraint

7.4 Future Work

- Extension to additional puzzle types (Kakuro, Killer Sudoku)
- End-to-end differentiable constraint layers for joint training
- Active learning for error correction (learning which substitutions succeed)
- Real-time mobile application for puzzle solving

References

References

- [1] Garcez, A. d., & Lamb, L. C. (2020). Neurosymbolic AI: The 3rd wave. *arXiv preprint arXiv:2012.05876*.
- [2] De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 337-340). Springer.
- [3] Wei, J., Wang, X., Schuurmans, D., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35, 24824-24837.
- [4] Babu, K., Vijayalakshmi, S., & Nedunchezian, R. (2010). Recognition of handwritten Sudoku puzzles. *International Journal of Computer Applications*, 1(4), 41-48.
- [5] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324.
- [6] Cohen, G., Afshar, S., Tapson, J., & Van Schaik, A. (2017). EMNIST: Extending MNIST to handwritten letters. *2017 International Joint Conference on Neural Networks (IJCNN)*, 2921-2926.

A Detailed CNN Architectures

A.1 Grid Detection CNN (PyTorch)

Listing 1: Grid CNN Architecture

```
1 class Grid_CNN(nn.Module):
2     def __init__(self, output_dim):
3         super(Grid_CNN, self).__init__()
4         self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
5         self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
6         self.pool = nn.MaxPool2d(2, 2)
7         self.dropout = nn.Dropout(0.25)
8         self.fc1 = nn.Linear(262144, 128)
9         self.fc2 = nn.Linear(128, output_dim)
10
11     def forward(self, x):
12         x = F.relu(self.conv1(x))
13         x = F.relu(self.conv2(x))
14         x = self.pool(x)
15         x = self.pool(x)
16         x = self.dropout(x)
17         x = x.view(x.size(0), -1)
18         x = F.relu(self.fc1(x))
19         x = self.fc2(x)
20         return x
```

A.2 Character Recognition CNN (PyTorch)

Listing 2: Character CNN Architecture

```
1 class CNN_v2(nn.Module):
2     def __init__(self, output_dim):
3         super(CNN_v2, self).__init__()
4         self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
5         self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
6         self.pool = nn.MaxPool2d(2, 2)
7         self.dropout = nn.Dropout(0.25)
8         self.fc1 = nn.Linear(3136, 128)
9         self.fc2 = nn.Linear(128, output_dim)
10
11     def forward(self, x):
12         x = self.pool(F.relu(self.conv1(x)))
13         x = self.pool(F.relu(self.conv2(x)))
14         x = self.dropout(x)
15         x = x.view(x.size(0), -1)
16         x = F.relu(self.fc1(x))
17         x = self.fc2(x)
18         return x
```

B Z3 Constraint Pseudocode

Algorithm 1 KenKen Z3 Constraint Generation

Require: Puzzle P with size n , cages C

Ensure: Solution grid X or UNSAT

```

1:  $X \leftarrow$  matrix of Z3 Int variables  $X_{i,j}$  for  $i, j \in [0, n)$ 
2:  $constraints \leftarrow \emptyset$ 
3: for  $i \in [0, n), j \in [0, n)$  do
4:    $constraints.add(1 \leq X_{i,j} \leq n)$  ▷ Cell range
5: end for
6: for  $i \in [0, n)$  do
7:    $constraints.add(\text{Distinct}(X_{i,*}))$  ▷ Row uniqueness
8: end for
9: for  $j \in [0, n)$  do
10:   $constraints.add(\text{Distinct}(X_{*,j}))$  ▷ Column uniqueness
11: end for
12: for cage  $(cells, op, target) \in C$  do
13:   if  $op = \text{"add"}$  then
14:     $constraints.add(\sum_{(i,j) \in cells} X_{i,j} = target)$ 
15:   else if  $op = \text{"mul"}$  then
16:     $constraints.add(\prod_{(i,j) \in cells} X_{i,j} = target)$ 
17:   else if  $op = \text{"sub"}$  and  $|cells| = 2$  then
18:     $(a, b) \leftarrow cells$ 
19:     $constraints.add(X_a - X_b = target \vee X_b - X_a = target)$ 
20:   else if  $op = \text{"div"}$  and  $|cells| = 2$  then
21:     $(a, b) \leftarrow cells$ 
22:     $constraints.add(X_a = target \cdot X_b \vee X_b = target \cdot X_a)$ 
23:   end if
24: end for
25:  $solver \leftarrow \text{Z3.Solver}()$ 
26:  $solver.add(constraints)$ 
27: if  $solver.check() = \text{SAT}$  then
28:   return  $\text{extract\_solution}(solver.model(), X)$ 
29: else
30:   return UNSAT
31: end if

```

C Complete Experimental Results

Table 13: Full KenKen Benchmark Results (100 puzzles per size)

Size	Solved	Failed	Timeout	Accuracy	Avg (ms)	Max (ms)
3×3	100	0	0	100.0%	143.6	312
4×4	100	0	0	100.0%	167.6	421
5×5	100	0	0	100.0%	204.4	589
6×6	100	0	0	100.0%	293.8	842
7×7	93	7	0	93.0%	511.0	2,890
9×9	62	37	1	62.0%	2,012	60,000

Table 14: Handwritten Error Correction Complete Results

Puzzle	Split	Base	Confidence	Unsat+TopK
Sudoku 4×4	83-17	92%	96%	96%
Sudoku 4×4	90-10	92%	99%	99%
Sudoku 9×9	83-17	79%	91%	89%
Sudoku 9×9	90-10	75%	99%	84%
HexaSudoku 16×16	83-17	6%	30%	32%
HexaSudoku 16×16	90-10	10%	37%	40%
HexaSudoku 16×16	95-5	8%	35%	42%
HexaSudoku 16×16	100-0	24%	62%	68%