# RECURSIVE REASONING WITH TINY NETWORKS

*Vincent Van Schependom (s251739), Malthe Bresler (s214631)*
*Jacob Corkill Nielsen (s204093), Sara Maria Bjørn Andersen (s202186)*

## ABSTRACT

In recent years the evolution of transformers in Natural Language Processing has involved larger and larger models, with exponential growth in parameters and required computation. Even then, these Large Language Models (LLMs) struggle with complicated tasks requiring thorough reasoning. The Tiny Recursive Model (TRM) proposes another approach for these tasks, by using a small, static neural network that iteratively refines a solution in latent space. The scope of this project is to explain how this model works, contrasting its "static mapping" approach with standard auto-regression in LLMs. We reproduce the results on the author's Sudoku dataset. Next, we apply the TRM to our own CSP puzzle, namely $N$-queens. We also attempt to test the architectures limits on 3 different kinds of problems, by applying it to a sequential planning task (Towers of Hanoi), as well as Math Arithmetic and Chess Move Prediction. We successfully validate the authors' findings on Sudoku, and find that the TRM works well as a static solver (straight from input to solution), leading to impressive results on unambiguous $N$-queens puzzles (98% accuracy) and quite good chess predictions (51.1% Top-5 accuracy). It has potential for non-CSP problems, too.

## 1. TINY RECURSIVE MODELS

This project considers a method proposed by Jolicoeur-Martineau called the Tiny Recursive Model (TRM) [1]. The TRM proposes a simplified approach to reasoning that moves away from the biological hierarchies of the Hierarchical Reasoning Model (HRM), which is inspired by the hierarchy of the brain (Wang et al. [2]). Instead of using two separate networks operating at different frequencies, TRM utilizes a single "tiny" network to iteratively refine a solution. The architecture relies on a recursive mechanism that progressively improves a latent reasoning vector and a candidate solution.

### 1.1. Problem Formulation and Variables

Unlike LLMs that predict the next token $t_{i+1}$ given $t_0..t_i$, the TRM takes an input state $x$ – representing the entire problem context (e.g., a full grid) – and iteratively refines it in the latent space, while maintaining a representation of reasoning in latent space; moving towards a *complete* solution.

The TRM abandons the biological metaphors of the HRM in favor of a clearer dual state representation. The current solution $y$ is the explicit, embedded candidate answer (previously denoted as $z_H$ in HRM). The latent reasoning variable $z$, on the other hand, is a hidden feature vector that stores computational history and constraints, analogous to a "Chain-of-Thought" in embedding space (denoted as $z_L$ in HRM).

The TRM also avoids the need for multiple networks by using a single network $f_\theta$ to handle both latent reasoning and solution updates, since moving variables between two latent spaces meaningfully requires the spaces to be identical.

### 1.2. A Single Recursion Process

The core of the TRM is the *Recursion Process*. Unlike standard recurrent networks that simply unroll over time, the TRM separates "thinking" from "answering". A single recursion process consists of two phases defined by the hyperparameter $n$:

1. **Latent Reasoning:** The model updates the latent variable $z$ repeatedly for $n$ iterations. In each iteration, the model considers the input $x$, the current solution $y$, and the previous reasoning $z$:

$$z \leftarrow f_\theta(x + y + z) \quad \text{for } i = 1 \dots n \tag{1}$$

   This step allows the model to perform $n$ steps of computation without committing to a new answer.

2. **Solution Refinement:** After $n$ latent steps, the model updates the proposed solution $y$ exactly once using the refined reasoning $z$:

$$y \leftarrow f_\theta(y + z) \tag{2}$$

   Crucially, this step does not directly observe $x$, forcing the model to rely on the reasoning encoded in $z$.

### 1.3. Deep Supervision & Effective Depth

To solve complex tasks, the TRM employs *Deep Supervision*, which is the "outer loop" of the model: the tuple $(y, z)$ is updated for $N_{\text{sup}}$ steps. We set this hyperparameter to $N_{\text{sup}} = 16$, as in the original paper.

To simulate a very deep network without the memory cost of Backpropagation Through Time (BPTT), TRM introduces the recursion depth hyperparameter $T$. Inside one supervision step, the model runs the full recursion process (defined above) $T$ times. For the first $T-1$ times, the recursion process is run *without gradients* to prime the variables $y$ and $z$. For the $T$-th time, the recursion process is run with gradients enabled.

This technique, distinct from the 1-step gradient approximation used in HRM, allows the model to backpropagate through a full recursion process ($n$ latent updates + 1 solution update) while benefiting from the depth of $T \times (n+1)$ total evaluations within each supervision step.

### 1.4. Adaptive Computational Time (ACT)

While $N_{\text{sup}}$ defines the maximum number of steps, the TRM uses Adaptive Computation Time (ACT) to decide when to stop thinking. Unlike HRM, which required a costly secondary forward pass for Q-learning, TRM learns a halting probability $q_{\text{halt}}$ via a simple Binary Cross-Entropy loss on the output head. If the value indicates high confidence during training, the loop breaks, and the model moves to the next data sample, which is critical for training efficiency and model generalization. The TRM thus learns to stop as soon as it finds the solution. Interestingly, while ACT is vital for training, the authors do *not* use it to stop early during testing/inference, where they perform the full $N_{\text{sup}} = 16$ steps.

### 1.5. Architecture Variants: Attention vs. MLP

The TRM architecture allows for flexibility in the *mixing* mechanism used within the network, which is responsible for exchanging information across the sequence dimension, allowing different parts of the input (e.g., distant grid cells) to communicate and influence each other's representations and is implemented based on the context length $L$.

The first variant uses standard multi-head self-attention. It is designed for tasks with large context lengths ($L \gg D$), such as the $30 \times 30$ grids found in the Maze-Hard and ARC-AGI datasets. The paper demonstrates that this variant is necessary for tasks requiring long-range spatial reasoning on larger maps.

The second variant (TRM-MLP) replaces the self-attention layer with a multi-layer perceptron applied across the sequence dimension. This approach is computationally cheaper and proved superior for tasks with small, fixed context lengths ($L < D$), such as Sudoku ($9 \times 9$).

## 2. DATASET STRUCTURE

A critical component of training on many *variants* of puzzles is the data hierarchy. The datasets are organized into three levels. Firstly, *groups* are the atomic unit for Train/Test splitting. They prevent data leakage by bundling *puzzles*, which are variations of a group generated via augmentation. For Sudoku, this includes digit permutations, band shuffling, and transposition. For Maze, this includes the 8 dihedral symmetries (rotations and flips). Of course, an equivalent representation of a base puzzle, on which we train the TRM, cannot be present in the test set. Groups prevent this data leakage. Lastly, within each puzzle, there are *examples*, which are actual Input/Label tensor pairs. Before training/inference, the data is flattened. For example, a simple $2 \times 2$ puzzle state might be represented as an integer vector like `[1, 3, 0, 5]`, where `0` represents an empty cell or padding, and other integers represent grid values.

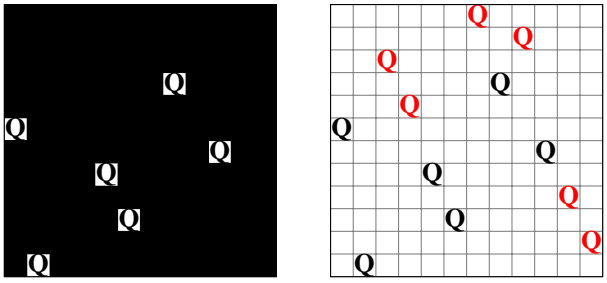
## 3. EXPERIMENT SETUPS

### 3.1. Suitability of Static Constraint Tasks

We determined that Sudoku and Maze are optimal fits for the TRM architecture, as they represent static "in-filling" tasks under global constraints. In Sudoku, the architecture's recursive cycles enable constraint propagation across the grid; for example, a digit placed at $(0,0)$ restricts valid possibilities at $(8,8)$. Similarly, Maze solving requires handling non-local dependencies where a barrier at coordinate $(6,7)$ may invalidate a pathing decision at $(5,5)$. The TRM's recursive latent updates ($z$) effectively allow the model to simulate backtracking and path verification internally before committing to a final static output $y$.

### 3.2. $N$-Queens

Like Sudoku, the $N$-Queens puzzle is a Constraint Satisfaction Problem (CSP) requiring the propagation of constraints across the board.




(a) Input (50.0% Masked) (b) Target (Red = to be inferred)

**Fig. 1**: Unambiguous $N$-queens input and output pair.

We selected a board size of $N = 12$ because it offers a non-trivial solution space of size $|\mathcal{S} = 14\,200|$, preventing simple memorization. For each of these solutions, we created a mask that covered 30% to 70% of the 12 queens, which we gave to the TRM as input $x$. The label $y$ was the unmasked board with all $N = 12$ queens placed in a correct way.

Standard $N$-Queens is a one-to-many problem (a single board configuration may allow multiple valid queen placements). Training the TRM (or any other deterministic model with Cross-Entropy loss) on one-to-many data causes the model to learn the average of all valid targets, resulting in a low-confidence, "blurry" output. That's why we created a "unambiguous" variant as well, where we strictly enforced a one-to-one mapping: for every generated mask, we utilized a backtracking solver to verify uniqueness. Any puzzle yielding multiple valid solutions was discarded in this variant, ensuring the TRM is trained strictly on function approximation rather than generative modeling.

For both puzzle variants, we trained the TRM using an MLP head, as suggested by the author in Section 4.5 of the original paper for puzzles of limited dimensionality (here, $D = 144$). Furthermore, we used a batch size of 512 and a learning rate of $1 \times 10^{-4}$. We use the same hyperparameters (except for the mixer variant) for *all* of our experiments going forward.

### 3.3. The Towers of Hanoi Failure

We attempted to extend the TRM to the sequential planning domain by testing it on the Towers of Hanoi. We generated datasets mapping the current state to the next optimal *action* and, alternatively, the next optimal *state*. While the model achieved $100\%$ training accuracy on setups with 3–6 disks, it failed to generalize, achieving $0\%$ accuracy on test cases with 7–9 disks. We attribute this failure to two primary factors.

Firstly, our data encoding mapped disk $N$ to input channel $N$. During training, the model observed zero activation in channels 7, 8, and 9. Consequently, during testing, the model encountered inputs in these previously "dead" channels. As the TRM (specifically the MLP-Mixer variant) learns position-dependent weights and is not permutation invariant, it lacked the mechanism to apply learned logic to these novel feature spaces.

Secondly, Hanoi is inherently a trajectory task where the solution is an exponential sequence of moves $(2^N - 1)$. The TRM is designed as a fixed-point solver that converges on a static global solution. Ideally, the model would output the full sequence, but the exponential output size hinders fitting the solution into a fixed grid. By constraining the TRM to predict only the *next step*, we underutilized its recursive capacity; the optimal next move is often an $\mathcal{O}(1)$ lookup operation that does not benefit from the deep latent reasoning required to solve global constraints.

### 3.4. Math Arithmetic

To further test the generalization capabilities of the TRM beyond spatial constraint satisfaction tasks, we introduced a symbolic reasoning task: math arithmetic. Unlike visual puzzles, where constraints propagate through a 2D topology, arithmetic requires hierarchical processing of operations (e.g., resolving parentheses) and sequential calculation.

**Fig. 2**: A chess board.

We generated a synthetic dataset of $100\,000$ training examples and $10\,000$ test examples using a recursive generation process. Each sample consists of a nested mathematical expression involving integers up to 20 and four operators: addition, subtraction, multiplication, and modulo $(+,-,*,\%)$. The expressions are generated with a maximum recursion depth of 4, resulting in strings such as $((12 + 4) * 3) = 48$.

In standard autoregressive modeling, the loss is often calculated over the *entire* sequence. However, as the TRM is trained as a static solver (mapping input $x$ directly to solution $y$), we formulated the task as a "fill-in-the-blank" problem. The *input* tensor contains the expression followed by the equals $(=)$ sign, with the answer masked (padded with zeros). Conversely, the *label* tensor masks the entire expression, leaving only the result. This ensures that the Cross-Entropy loss is calculated exclusively on the *answer* digits, forcing the model to perform the calculation internally within its latent $z$, rather than memorizing the input sequence. We hypothesize that the TRM's recursive iterations effectively act as a computational scratchpad, allowing the tiny network to resolve inner parentheses and intermediate values before committing to the final static integer output.

### 3.5. Chess Move Prediction

To test the generalization and limits of the TRM even further, we also tested chess move prediction. While this can be considered a puzzle, it needs slight abstraction to fit the TRM, as it doesn't fit the general type of "filling-in" CSP. It does, however, fit well with the heavy reasoning nature of the model. In Chess Move Prediction, the model receives an encoded *Forsyth-Edwards-Notation* (FEN) chess position and must predict a move in *Long Algebraic Notation* (LAN).

Long Algebraic Notation describes a *move* via a four character string. For example, consider the chess board in Figure 2; moving the left-most pawn two squares ahead, from `a2` to `a4`, is represented as `a2a4` in LAN.

Forsyth-Edwards-Notation, on the other hand, is used to represent the state of the *board*, row by row. Row indentation is marked by `/`. In each row, the letters `p`, `n`, `b`, `r`, `q`,

k individually represent the pieces (n being knight). Uppercase pieces are white, lowercase pieces are black. Numbers represent consecutive empty fields. After the board comes the additional information about whose turn it is, castling availability, the clock and the amount of half-moves. The chess board in Figure 2 is represented as:

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/
        RNBQKBNR w KQkq - 0 1
```

The model vocabulary includes the FEN characters, the mask token, and all possible LAN moves (64×64), totaling 4181 tokens. Inputs consist of the full FEN string and the masked target move. To capture the complex spatial relations between pieces, we employed the attention architecture with Rotary Positional Embeddings (RoPE). The dataset consists of 200,000 examples sampled from *Lichess* (A.3), split into 90% training and 10% validation sets. Model performance is reported using top-$k$ accuracy, specifically for $k \in \{1, 3, 5\}$.

## 4. RESULTS

To make sure that the author's code worked as explained in the paper, we reproduced the results obtained on *one* of the datasets, namely Sudoku Extreme, with the rest of our efforts going to our own experiments introduced above. Using the author's hyperparameters, we achieved a final test accuracy of $84.3\,\%$, a little bit shy of the papers $87.4\,\%$. This difference can be due to the maximum run-time of 24 hours on DTU HPC, which was not quite enough to perform the full amount of epochs specified by the author. However, as seen in Appendix B, the TRM seems to have converged. This, together with the obtained accuracy of 84.3%, is sufficient to conclude that the architecture runs as described in the paper on this dataset.

We summarize the results from our own CSP and generalization experiments in Table 1.

**Table 1**: TRM Performance our own experiments. $N_s$ denotes the average number of deep supervision steps in training.

| **Experiment** | **Accuracy** | $N_s$ |
|---|---|---|
| $N$-queens (Multiple Solutions) | $40\,\%$ | 10 |
| $N$-queens (Single Solution) | $97\,\%$ | 5 |
| Math Arithmetic | $43.2\,\%$ | 8 |
| Chess Move Prediction (Top-1) | $11.2\,\%$ | 14 |
| Chess Move Prediction (Top-3) | $34.0\,\%$ | 14 |
| Chess Move Prediction (Top-5) | $51.1\,\%$ | 14 |

## 5. DISCUSSION

Generally, the TRM shows impressive problem-solving skills on many hard problems. $N$-Queens represents a problem with a massive state space; especially for $N = 12$. Interestingly, there is a large discrepancy between the accuracy of the unambiguous and multi-solution versions. As the TRM thinks of the *whole* solution, rather than placing one queen at a time, it might have trouble with conflicting queen positions when multiple solutions are possible.

The result from our Math Arithmetic experiment highlights a moderate to good understanding the hierarchy in simple math expressions, something that LLMs often handle by invoking code generation.

Finally, for the Chess Move Prediction problem, the TRM was able to pick high quality moves a lot of the time. It is difficult to determine a 'correct' chess move. The best moves in the dataset were originally generated with the state-of-the-art chess engine *Stockfish*, and as such are not necessarily the *perfect* moves, but simply the tree search's best guess. While this makes it harder to quantify the quality of moves, the top-5 and top-3 metric help us showcase how often a move is contained in the top answers. Right now, we simply predict tokens from all possible moves, but for future work, it would be interesting to see how often the model predicts illegal moves. This would test how well the model learns the rule-set, simply based on positions and best moves.

The number of deep supervision steps $N_s$ indicates how 'easy' the TRM considers the problem. If it needs many recursion steps in the latent space, the problem is hard to solve, while few supervisions indicate that it finds a solution rather quickly. This turned out to be an additional tool to help against overfitting, helping to show when the model is memorizing examples rather than solving them.

We used our test / training split of $10\%$ and $90\%$ to produce our results. Since we did not tune our hyper-parameters to optimize our results (all experiments were run with the same hyperparameters), we did not find it necessary to have a separate validation set and test set – especially because of limited GPU time on the DTU HPC.

## 6. CONCLUSION

After briefly explaining the workings for the Tiny Recursive Model (TRM), we initially investigated how well we could recreate the results of the paper. We were able to obtain a Sudoku accuracy of $84.3\%$, just shy of the papers $87.4\%$, most likely due to 24h train limit. We then continued to investigate performance for a CSP, namely $N$-Queens, as well as other types of puzzles, like the Towers of Hanoi, Math Arithmetic and Chess Move Prediction to test the limits of the TRM. $N$-Queens showed impressive results in case of puzzles with a single solution, achieving an accuracy of 97%. The Towers of Hanoi did not work well due to the next-state prediction nature, as opposed to complete-solution refinement, as in CSP problems. Math Arithmetic showed a moderate understanding of simple mathematical expressions. Finally, for Chess Move Prediction, the TRM was able to consistently pick high quality moves. Overall, the TRM – as is – is promising for deterministic CSPs and has potential to be used for other types of reasoning tasks.

## 7. REFERENCES


[1] Alexia Jolicoeur-Martineau, "Less is More: Recursive Reasoning with Tiny Networks," 10 2025.

[2] Guan Wang, Jin Li, Yuhao Sun, Xing Chen, Changling Liu, Yue Wu, Meng Lu, Sen Song, and Yasin Abbasi Yadkori, "Hierarchical Reasoning Model," 8 2025.


## DECLARATION OF USE OF GENERATIVE AI

This declaration **must** be filled out and included as the **final page** of the document. The questions apply to all parts of the work, including research, project writing, and coding.

- I/we have used generative AI tools: yes

If you answered *yes*, please complete the following sections. List the generative AI tools you have used:

- ChatGPT by OpenAI
- Gemini by Google
- Claude by Anthropic
- Microsoft Copilot inside VSCode

Describe how the tools were used:

**What did you use the tool(s) for?** We mostly used the tools in the coding process. More precisely with regards to debugging and understanding the codebase and fixing bugs along the way, along with speeding up repetitive coding tasks.

**At what stage(s) of the process did you use the tool(s)?** It was mainly used in the initial phase for getting the code up and running, both on our local computers, but also on HPC.

**How did you use or incorporate the generated output?** We used code snippets (for repetitive tasks, such as structure of datasets) directly. Otherwise, we just used it to gain information.

# Appendix

## A. RESOURCES

### A.1. Project code

github.com/schependom/DTU_deep-learning-project

- `README.md` with all implementation details (how the datasets were created, how we set up the model on DTU HPC, etc.)
- The TRM code, forked and adapted based on the original repo.
- Our DTU HPC jobscripts.
- This report and the synopsis.

### A.2. Original (author) repository

github.com/SamsungSAILMontreal/TinyRecursiveModels

### A.3. *Lichess* open database

https://database.lichess.org/evals

## B. SUDOKU EXTREME

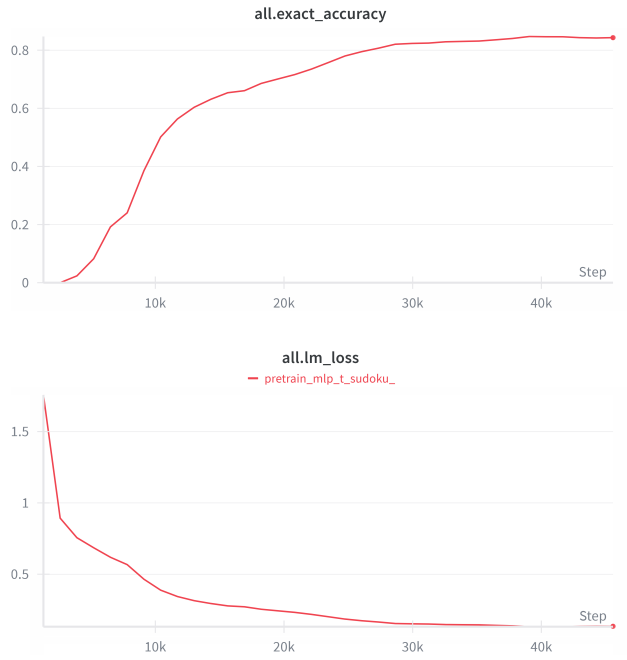



**Fig. 3**: Test accuracy (top) and test loss (bottom) for the Sudoku Extreme dataset.