Beyond Pattern Recognition——A New Paradigm Uniting Symbolicism and Connectionism

Preamble

- **Title:** Beyond Pattern Recognition——A New Paradigm Uniting Symbolicism and Connectionism
- **Author:** Ball Lightning
- **Abstract:** This research challenges the conventional view of neural networks as mere statistical pattern recognizers, proposing and validating a new paradigm capable of awakening their endogenous capacity for exact reasoning. By employing programmatically generated ideal data and a parallel, non-autoregressive solving framework, this paradigm transforms standard neural networks from probabilistic mimics into deterministic rule executors. I powerfully demonstrate the universality of this paradigm through its success across a series of tasks, including symbolic rule execution, algorithmic fitting, visual reasoning, physics simulation, and interpretability. Furthermore, I systematically evaluate multiple representative architectures (MLP, CNN, RNN, UNET, Diffusion) on the task of multi-step evolution of a 1D cellular automaton, confirming that various non-Transformer models possess similar capabilities. This finding provides strong evidence that the ability for precise algorithmic execution is an inherent potential of connectionist systems, not the patent of a specific architecture. Through the **Neural Sculpting Paradigm**, this work, for the first time, systematically demonstrates that standard neural networks possess the intrinsic potential to execute arbitrary algorithms with precision, opening up a novel path toward building highly reliable and interpretable artificial general intelligence systems.**Code is available at:** https://github.com/ball-lightning6/neural-sculpting-paradigm.

Main Body

Act I: The Reasoning Gap of Modern AI

Deep neural networks have achieved tremendous success in pattern recognition tasks. However, in domains requiring precise, multi-step logical reasoning, their propensity for "hallucinations" and logical inconsistencies remains a fundamental bottleneck. This limitation curtails the application of AI in high-reliability fields such as science and mathematics. This paper posits that the root of this predicament is not an inherent flaw in neural networks, but a fundamental mismatch between their architectural inductive biases and the intrinsic structure of the tasks.

Based on this core insight, I propose and validate a novel, concise, and powerful paradigm for solving-based reasoning—which I term the **Neural Sculpting Paradigm**. By abandoning autoregression in favor of single-step, structured output solving, and by using

programmatically generated ideal data, this paradigm systematically transforms a standard [2] Transformer from a statistical mimic into a precise rule executor. My extensive experiments across a range of tasks spanning symbolic rules, algorithm learning, visual reasoning, physics simulation, and interpretability not only verify the universal effectiveness of this paradigm but also, through experiments like semantic shuffling, prove that the model learns abstract algebraic structures rather than superficial symbolic patterns. My work provides a simple, universal, and scalable new path for building trustworthy, interpretable AI systems capable of high-order reasoning.

Act II: A Brief History of Neuro-Symbolic Computing

The fusion of the powerful learning capabilities of neural networks with the rigorous reasoning abilities of symbolic systems, known as "neuro-symbolic computing," has long been the **Holy Grail** of artificial intelligence. Previous explorations have primarily followed two paths.

The first path is "architectural modification." This line of work attempts to imbue models with reasoning capabilities by designing specialized network architectures that mimic symbolic computation processes. Its pioneering explorations can be traced back to the [6] Neural Turing Machine, which introduced external memory modules, and its successor, the [7] Differentiable Neural Computer. This idea was further extended in later developments like Neural Module Networks [11] and Graph Neural Networks [10]. The common feature of these works is the attempt to enhance neural networks by "designing specialized hardware for reasoning." However, these specialized architectures often struggle with transferability to new task domains and face challenges of complex design.

The second path is "logic embedding." This line of work seeks to "embed" external symbolic logic rules into the learning process of a neural network in a differentiable form. Whether it's the [8] Neural Theorem Provers that transform rules into loss function constraints via fuzzy logic, or [9] DeepProbLog which treats a neural network as a component of a probabilistic logic program, the core idea is to use an "external" logical system to "guide" or "constrain" an "internal" neural network.

However, both of the aforementioned paths share a common, underlying "hybrid" philosophy, treating "learning" and "reasoning" as two heterogeneous capabilities that need to be "glued" together externally.

In fundamental contrast, my work discovers that reasoning is a "native, dormant" ability of the Transformer architecture. It does not need to be "mixed"; it needs to be "awakened." I found that through a novel and simple training paradigm—the combination of "solver-style output" and "ideal data"—a standard, general-purpose Transformer architecture can spontaneously and systematically learn to execute precise symbolic rules. My contribution lies not in designing a more complex "model," but in discovering a more correct "method" to unlock the immense potential that already exists within the model.

Act III: The Neural Sculpting Paradigm

The new paradigm I have discovered does not center on designing a novel model architecture. Instead, by fundamentally reconstructing the learning environment and task definition, it systematically demonstrates the powerful intrinsic reasoning capabilities of neural networks. The forms of problems this reasoning ability applies to are not limited to symbolic rules, algorithmic problems, visual reasoning, physics simulation, and interpretability. To summarize it in one sentence, it is a class of problems with well-defined transformation relationships, where for a corresponding input, there exists a mechanically executable symbolic process that transforms it into a unique, precise output.

All my main experiments are based on the standard Transformer architecture and its variants in the image domain, such as [3] ViT or [4] Swin Transformer. However, I have also conducted a few experiments demonstrating that networks like MLPs, RNNs, and CNNs possess similar capabilities, although perhaps not as powerful as the Transformer. This proves that this ability is not exclusive to the Transformer but is an inherent capability of neural networks.

This paradigm is built upon the following three synergistic core principles.

3.1. The Core Engine: The Transformer, Biased Towards "Abstract Relations"

Although other types of neural networks also possess this capability, experiments show that the Transformer is the most suitable for these kinds of reasoning problems, and its ability in this regard is the strongest. The reason may be that the Transformer's core self-attention mechanism fundamentally breaks the constraints of space. It allows any element in a sequence to interact directly and in parallel with all other elements, calculating the "strength of their relationship." This architecture is, by its very nature, profoundly "isomorphic" with the intrinsic structure of the task of "reasoning." It is not "looking" at the neighborhood of pixels; it is "understanding" the "relational graph" of the entire symbolic system. Experiments have found that Transformers, ViT, Swin Transformers, etc., all have the same capabilities in the domains listed earlier: symbolic rules, algorithmic problems, visual reasoning, physics simulation, interpretability, and so on. Therefore, I can posit that the classification of these problems is merely superficial; there may be no essential difference between them.

3.2. Output Paradigm: From "Sequential Mimicry" to "Parallel Solving"

The reasoning capabilities of current large models primarily rely on autoregressively predicting the next token. In contrast, I adopt a method that completely abandons autoregression, predicting the entire output in a single pass. For this class of reasoning problems, this method is more successful for the following reasons. The non-autoregressive approach makes the format of input and output data more pure and consistent, which is something the next-token prediction method inherently cannot achieve; the latter may be better suited for large language models. Moreover, this "word-by-word prediction" model has a fundamental flaw: error accumulation. Any minor error early in the sequence can be amplified in subsequent generation, leading to a complete collapse of the logical chain. In contrast, my "one-shot" format forces the model to perform global, synergistic computation and self-consistency constraints among all its internal "neurons" in a single forward pass.

This transforms the reasoning process from a fragile "linear chain of guesses" into a robust "parallel constraint solving" process.

3.3. Data Paradigm: From "Real-World Noise" to "Ideal Laws"

The traditional machine learning paradigm, often due to the nature of real-world data, is forced to distill knowledge from data fraught with noise, spurious correlations, and uncertainty. Such data often contains noise in both inputs and outputs, and there is no clear, precise transformation rule. Consequently, neural networks trained on such data can only exhibit pattern recognition capabilities. However, for learning precise, deterministic rules, this "dirty data" is itself a powerful "source of interference."

My paradigm adopts a diametrically opposed "data philosophy." I use programmatic scripts to generate massive amounts of logically pure "ideal data." In this data, there is only a single, necessary, and precise logical relationship between the input and output, without any statistical "shortcuts" or "coincidental patterns." This high "signal-to-noise ratio" learning environment drastically reduces the difficulty of learning. It leaves the model with no choice but to learn the one, unchanging "signal" hidden behind all samples—that is, the underlying rule itself. My extensive experiments (see Chapter 4) demonstrate that under this paradigm, the model only needs to cover a negligible fraction of the total input space to achieve perfect generalization across the entire problem space.

Additionally, the training data must share the same distribution as the input space (or at least, this seems to be the most convenient approach at present). I achieve this condition in my dataset generation scripts through random sampling. This paradigm cannot generalize outside the input space. In other words, since the training data occupies a very small portion of the entire input space, the model, after convergence, can generalize precisely to the entire vast remaining input space. This is analogous to generalization in past paradigms.

3.4. Paradigm Interpretation: Sculpting Precise Rules with Gradient Descent

In summary, my paradigm can be understood as a philosophical practice of using a connectionist process to infinitely approximate a symbolic ideal. It is like a sculptor facing a rough block of marble (a randomly initialized neural network). The "ideal data" provides him with countless perfect "photographs" of the "Statue of David" from every angle; the "solver-style output" loss function tells him what "discrepancies" still exist between his current work and that "ideal"; and "gradient descent" is the miraculous "chisel" in his hand, which, with every strike, chips away a small piece of excess stone in the right direction.

Through millions of tiny, precise "chisels," the perfect "Statue of David," representing the "precise rule," naturally "emerges" from the "continuous, chaotic" block of neural network stone. And for problems that cannot converge perfectly, this paradigm can always compress the uncertainty of the output.

In essence, the paradigm I've discovered is simply using connectionist gradient descent to fit a precise rule. This is an extremely natural way to connect connectionism and symbolism. I

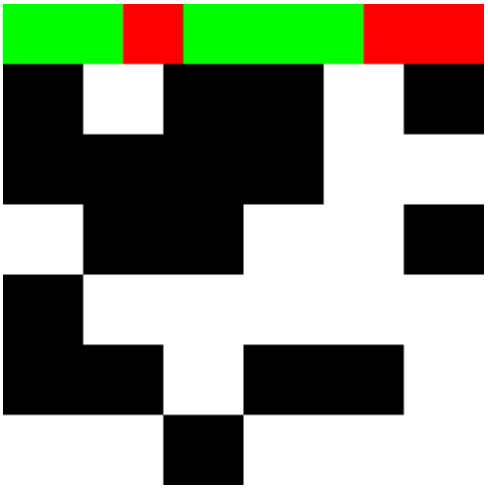
summarize this paradigm as being multi-modal and capable of end-to-end learning for almost any transformation rule that has a precise definition.

3.5. Model Description

Symbolic/Algorithmic Reasoning — qwen2_0.5b [21] fine-tuned with LoRA + custom lm_head, or later replaced with a small Transformer trained from scratch.
Image-to-Symbol Reasoning — ViT/Swin Transformer
Multi-modal/Image2Image — Swin Transformer + UNet
Text-to-Image Reasoning — A small Transformer followed by a UNet

Corresponding Code:
Multi-modal/Image2Image: train_image2image.py
Symbolic/Algorithmic Reasoning: train_tiny_transformer.py
Text-to-Image Reasoning: train_text2image.py / train_qwen2_text2image.py
Image-to-Symbol Reasoning: train_swin_image2text.py

For multi-modal reasoning, I am currently using image inputs and encoding symbolic inputs into the image to solve it, i.e., Swin Transformer + UNet. There should be other methods in the future. An example is as follows: the top layer of 8 green and red squares represents the rule of a cellular automaton, and the 6x6 grid of black and white cells below represents the initial state of a 36-cell 1D cellular automaton. This method has achieved good results in the task of predicting the state after applying the cellular automaton rule for two layers.



Additionally, MLPs, CNNs, and RNNs can complete simple tasks, and other models also have the potential to complete this type of task. However, I still use the Transformer as the standard model.

Act IV: Empirical Evidence Across Domains

4.1. Foundational Capability: Symbolic Rule Learning

This section aims to verify, at the most fundamental level, the core capability of this paradigm to learn precise and complex symbolic rules.

Train/Validation Split Ratio: Evaluation is generally performed at a frequency of about every 1000 steps to monitor the trend of eval loss. The validation set size is calculated based on this frequency to avoid excessively long evaluation times. The validation set typically constitutes less than 0.01 of the total dataset. Since the validation set, training set, and input space share the same distribution, the eval loss can reliably verify the output.
Model Training Hyperparameters: Almost all hyperparameters remain unchanged, except for sometimes increasing the batch size on simpler tasks to better utilize GPU memory. For some image-related reasoning tasks, the learning rate is halved if the loss becomes NaN.
Model Structure Variations: For tasks with symbolic outputs, the most important change is typically adjusting the number of labels for multi-label binary classification.
Input Space Estimation: For some tasks where the input space is difficult to calculate directly, I use a Monte Carlo-like method. I generate a large number of task-related random numbers (sufficient to define a single sample) and estimate the magnitude of the input space based on the number of duplicate samples.

4.1.1. Cellular Automata

Task Description: The model learns to predict and output the state of a 1D cellular automaton after a given number of transformations according to a given rule.
Input Format: A 30-character string consisting of "0"s or "1"s, representing the initial state of the cellular automaton.
Output Format: A multi-label binary classification format with 30 labels, equivalent to a 30-character 0/1 string, representing the state of the cellular automaton after transformation.
Loss: Mean Binary Cross-Entropy Loss
Training Configuration: NVIDIA 4090 GPU
Dataset Generation Script: generate_cellular_automata_1d.py
Dataset: ca_rule110_layer15_30.jsonl
Training Code: train_tiny_transformer.py
Training Set Size: 500,000
Input Space Size Estimation: 2^30 = 1,073,741,824
Training Set Size as a Percentage of Input Space: 0.0466%

Task-Related Design Philosophy and Discussion:

1.To rigorously test the ability of neural networks to go beyond pattern recognition and learn to execute precise symbolic rules, I chose the 1D cellular automaton as the primary experimental platform. This choice was not accidental but based on its unique, ideal properties that eliminate all ambiguity. First, a CA is a computational system driven entirely by local, deterministic rules. Its behavior does not depend on any statistical priors or semantic information from the real world, making it a perfect "sterile environment" for testing pure symbolic manipulation capabilities. Second, its vast state space (for a binary cellular automaton of width W, there are 2^W possible states) makes any form of "rote memorization" futile, thereby forcing the model to learn the rule itself rather than memorizing input-output pairs.
On this basis, I specifically chose Rule 110 as the core test rule. This is because Rule 110 is

not only a non-trivial rule that exhibits complex and seemingly chaotic behavior, but more critically, it has been proven to be Turing-complete. This means that, in principle, Rule 110 can simulate any computational process. Therefore, having a Transformer learn the evolution of Rule 110 is equivalent to testing whether the model possesses the potential to learn a form of universal computation. Successfully learning this rule would provide the most solid and convincing experimental evidence for the argument that "neural networks can internalize complex algorithms."
2.Some task parameters:
Width: 30
Rule: 110
Layers: 15

Experimental Results and Analysis: The evaluation loss converged to 0.000068. This extremely low evaluation loss, combined with the fact that the training set size is far smaller than the input space, strongly indicates that the neural network has learned the rule.

Final Convergence Result:

--- Step 477000 ---
Train Loss: 0.001766
Eval Loss: 0.000068

4.1.2. Probing the Nature of Learning: N-ary Addition and Semantic Shuffling

Task Description: The model learns to predict and output the sum of two N-ary numbers. The training results of semantic shuffling and position shuffling are compared.
Input Format: A 16-character string, where the first 8 characters represent one number and the last 8 characters represent another.
Output Format: A multi-label binary classification format with 33 labels, equivalent to a 33-bit binary string, representing the sum as a 33-bit binary number.
Loss: Mean Binary Cross-Entropy Loss
Training Configuration: NVIDIA 4090 GPU
Dataset Generation Script: generate_symbol_add_shuffle_dataset.py
Datasets:
adder_8bit_base16_train.jsonl — No semantic shuffling, no position shuffling
adder_8bit_base16_sem_shuffled_train.jsonl — Semantic shuffling, no position shuffling
adder_8bit_base16_pos_shuffled_train.jsonl — No semantic shuffling, position shuffling
adder_8bit_base16_sem_shuffled_pos_shuffled_train.jsonl — Semantic shuffling, position shuffling
Training Code: train_tiny_transformer.py
Training Set Size: 500,000
Input Space Size Estimation: 16^16 = 18,446,744,073,709,551,616 = 1.84e19
Training Set Size as a Percentage of Input Space: 2.71e-14

Task-Related Design Philosophy and Discussion:

1.The model learns to perform addition on two 8-digit hexadecimal numbers. To investigate the nature of what the model learns, I designed a "semantic shuffling" control experiment, which includes two types of shuffling. One uses randomly different symbols to represent the hexadecimal digits, and the other randomly shuffles the positions of the hexadecimal digits, no longer adhering to the format where the first 8 digits and last 8 digits represent the two numbers.
Reasons for this experimental design:

- Semantic Shuffling: To confirm that the Transformer learns the structure between symbols relative to the task, independent of what the specific symbols are.
- Position Shuffling: To confirm that the Transformer's ability to learn this task, and most tasks, does not depend on the arrangement of input symbols.
  2.Some task-related parameters and details:
  Base: 16 (hexadecimal)
  Number of digits for each number: 8
  Without semantic shuffle: `0123456789abcdef` are used as symbols for hexadecimal digits.
  With semantic shuffle: Any distinct visible ASCII characters are chosen as symbols for hexadecimal digits.
  Without position shuffle: The first 8 characters of the input string are one 8-digit hexadecimal number, and the last 8 are the other.
  With position shuffle: At the beginning of the dataset generation script, a position shuffle map is created and kept consistent throughout the entire dataset. The corresponding code is as follows:

```
if SHUFFLE_POSITIONS:
    position_map = list(range(INPUT_LEN))
    random.shuffle(position_map)
    POSITION_SHUFFLE_MAP = {from_idx: to_idx for from_idx, to_idx in
enumerate(position_map)}
```

Experimental Results and Analysis: The results in the table below decisively prove my core claim. In both "no semantic shuffle" and "with semantic shuffle" conditions, the model eventually converged to an extremely low loss level (<0.0001), with no significant difference in performance. This irrefutably demonstrates that the model learned the abstract algebraic structure underlying the addition operation, rather than superficial symbolic patterns.

Note: Different random data and training seeds can produce very different training loss curves, so a single comparison cannot be used to analyze training difficulty. However, across multiple attempts, the convergence of all four experimental setups was consistently observed.

As the training processes were relatively short, the complete training progress for all 4 experiments is listed below, represented by eval loss.

step	no sem shuffle, no pos shuffle	no sem shuffle, pos shuffle	sem shuffle, no pos shuffle	sem shuffle, pos shuffle
1000	0.534105	0.505196	0.511109	0.564834
2000	0.438036	0.263462	0.446237	0.497787
3000	0.347195	0.001700	0.365586	0.461389
4000	0.324581	0.001392	0.179969	0.395561
5000	0.254971	0.000123	0.063128	0.328988
6000	0.129849	0.000042	0.001083	0.247512
7000	0.066946	0.000024	0.000203	0.159110
8000	0.044260		0.000123	0.149018
9000	0.032533		0.000051	0.139221
10000	0.001005			0.103275
11000	0.000325			0.080524
12000	0.000099			0.052849
13000				0.022913
14000				0.002443
15000				0.002063
16000				0.000327
17000				0.000472
18000				0.000272
19000				0.000293
20000				0.000409
21000				0.000173
22000				0.000164
23000				0.000391
24000				0.000243
25000				0.000095

4.2. Advanced Capabilities: Algorithm Learning and Planning

4.2.1. Case Study: Trapping Rain Water

Task Description: Derived from the classic LeetCode algorithm problem, 42. Trapping Rain Water - LeetCode [20]

42. Trapping Rain Water

Hard | Topics | Companies

Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it can trap after raining.

Example 1:


```
Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: The above elevation map (black section) is represented by array
[0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.
```

Example 2:

```
Input: height = [4,2,0,3,2,5]
Output: 9
```

Constraints:

- `n == height.length`
- `1 <= n <= 2 * 10^4`
- `0 <= height[i] <= 10^5`

Input Format: A 30-character 0/1 string. Each consecutive 3-character 0/1 substring represents the height of a bar (0-7) as a 3-bit binary number, for a total of ten bars.
Output Format: A 30-character 0/1 string. Each consecutive 3-character 0/1 substring represents the amount of rainwater trapped by a bar (0-7) as a 3-bit binary number, for a total of ten bars.
Loss: Mean Binary Cross-Entropy Loss
Training Configuration: NVIDIA 4090 GPU
Dataset Generation Script: generate_trapping_rain_water_decoupled.py
Dataset: trapping_rain_water_decoupled_n10_b3_train.jsonl
Training Code: train_tiny_transformer.py
Training Set Size: 300,000
Input Space Size Estimation: 2^30 = 1,073,741,824 = 1.07e9
Training Set Size as a Percentage of Input Space: 0.0279%

Task-Related Design Philosophy and Discussion:

1.LeetCode is a natural repository for testing the algorithmic capabilities of this paradigm. I have tested many algorithm problems from here, and the paradigm can fit many of them. The "Trapping Rain Water" problem is categorized as "Hard" on LeetCode. Its solution cannot be simulated by any simple symbolic rule. As for the principle of how it is fitted, it

should be considered unknown for now. However, I believe the Transformer does not follow the typical human thought process of first parsing the format and then executing algorithmic steps one by one. It is impossible for it to know the meaning of each bit through any method, nor can it know the meaning of the task itself. It is simply performing a simple task: stochastic gradient descent according to the goal defined by the loss. This is why I believe this paradigm can be described as—using a connectionist method, gradient descent to approximate the discrete.
2.Regarding why the output is not the total amount of trapped water: I previously tried training directly with the binary representation of the total amount as the target, but the results were not good, and convergence was difficult. So, I adopted this decoupled format, which allowed for extremely rapid convergence. Here, I relieved the Transformer of the burden of summation, which is likely the reason for the model's quick convergence. Additionally, in another experiment I conducted, a task that serially mixed many simple operations also had difficulty converging. This is a very interesting phenomenon worthy of further study and will be discussed later.
3.Some task parameters:
Number of bars: 10
Number of binary digits to represent bar height and water amount: 3
Range of bar height / water amount: 0-7

Experimental Results and Analysis: After 16,000 training steps, the model's evaluation loss converged to 0.000059. This result indicates that the paradigm can effectively learn complex algorithms that require understanding global information (such as the highest walls to the left and right) and, through the decoupling of the output format, has achieved precise modeling of the problem's structure.

Final Convergence Result:

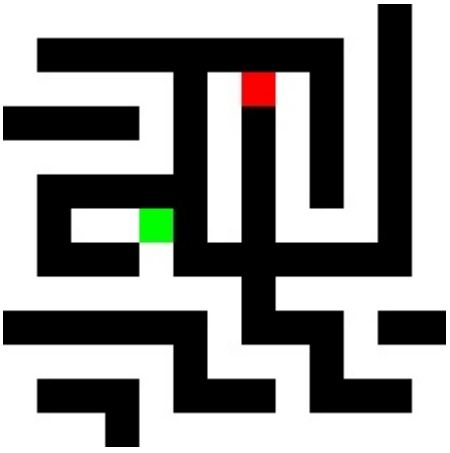
--- Step 16000 ---
Train Loss: 0.000570
Eval Loss: 0.000059

4.2.2. Dense Maze Pathfinding

Task Description: In a complex, dense maze designed for human players, the model learns to predict the "next step" direction (Up/Down/Left/Right) for the optimal path from any given position to the goal.
Input Format: A 169-character string consisting of "0", "1", "s", or "t", representing the initial state of the maze. Each consecutive 13 characters represents a row of the maze. '0' is open space, '1' is a wall, 's' is the start point, and 't' is the target point. There is exactly one 's' and one 't'. A border of walls is assumed around the 13x13 maze.
Output Format: 4-class classification
Loss: Cross-Entropy Loss for 4 classes
Training Configuration: NVIDIA 4090 GPU
Dataset Generation Script: generate_maze_dense.py

Dataset: maze_optimized_13_13_dataset.jsonl
Training Code: train_tiny_transformer.py
Training Set Size: 2,000,000
Input Space Size Estimation: Using a Monte Carlo method, 2,000,000 samples were generated with no repetitions. A rough estimate is 2^169 ≈ 7.48e50.
Training Set Size as a Percentage of Input Space: Approximately 2.673e-45

The image below shows a typical maze, with white for open space, black for walls, green for the start point, and red for the end point. This is a typical maze game suitable for humans, as opposed to maze generation algorithms that randomly place individual wall points.



Task-Related Design Philosophy and Discussion:

1.Based on my confidence in this paradigm's ability to fit algorithms, the initial design of this maze experiment used an algorithm that was equivalent to randomly generating wall points. However, practice showed that such mazes, while perhaps very easy for humans to solve at a glance (the shortest path is also the only viable path), have a number of branches and complexity that far exceed the current maze experiment setup.
2.The reason for choosing the next step direction of the shortest path is that this format is relatively simple to design. Directly outputting the shortest path would be difficult as its length is not known in advance, and even with other measures, it would make the task design less elegant. Another important reason is that this design implies another possibility: the maze task can be seen as choosing the best action in a certain state. This can be linked to reinforcement learning. Training state-action pairs in this way could give this paradigm the potential to complete reinforcement learning tasks, see the discussion on arc-agi-3 later. There are two methods for training: directly supervising the best action and using reinforcement learning to explore the best action. However, due to time and energy constraints, I have not completed the experiment of directly using reinforcement learning to explore the best action.
3.Some task parameters:
Width: 13
Height: 13

Experimental Results and Analysis: The evaluation loss converged to 0.000014, indicating that the model has demonstrated a very strong pathfinding capability. This proves that the paradigm can "amortize" a traditional "search" problem into the weights of a feedforward

network, achieving efficient "search-free planning." The extremely low evaluation loss means the model has learned to navigate similar mazes, and it also implies that on the scale of the length of the optimal path, the accuracy of the entire path will be very high.

Final Convergence Result:

--- Step 279000 ---
Train Loss: 0.000083
Eval Loss: 0.000014

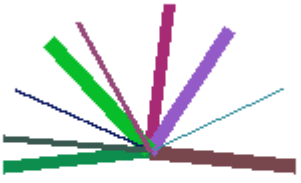
4.3. Reasoning from Images and Extracting Symbolic Output

This section will demonstrate the paradigm's powerful potential in handling multi-modal tasks, including decoding symbols from visual information.

4.3.1. Visual Symbol Decoding: Clock Dial Angle Recognition

Task Description: Image reasoning. Given an image of something like a clock dial, the model outputs the angular intervals occupied by line segments of different colors and thicknesses.
Input Format: 224x224 image
Output Format: Multi-label binary classification format with 36 labels, equivalent to a 36-character 0/1 string, indicating whether each interval is occupied by a line segment. The 36 intervals evenly divide the 360-degree angle, with each interval being 10 degrees.
Loss: Mean Binary Cross-Entropy Loss
Training Configuration: NVIDIA 4090 GPU
Dataset Generation Script: generate_line_angle_to_vector.py
Dataset: line_angle folder
Training Code: train_swin_image2text.py
Model Used: Swin Transformer
Training Set Size: 200,000
Input Space Size Estimation: 389,329,651
Training Set Size as a Percentage of Input Space: 0.0514%

Illustration:



Task-Related Design Philosophy and Discussion:

1.After demonstrating the algorithm fitting capability, I thought of the application of Transformers in images and wanted to test if image Transformers have similar reasoning abilities. I tested ViT on a checkerboard-related task, and the results were very good. After that, I designed this task with the intention of testing pure image understanding, rather than simply recognizing symbols in the image and then applying the already proven symbolic rule learning and algorithm fitting capabilities of the Transformer.
2.This task initially used the ViT model, which did not perform well, so I later switched to the currently used Swin Transformer.
3.To increase diversity, the color and thickness of the line segments were randomized.
4.Some task parameters:
Image Resolution: 224x224
Number of intervals: 36
Angle per interval: 10

Experimental Results and Analysis: The evaluation loss converged to 0.019183. While this loss does not represent perfect convergence, it demonstrates that the model can accurately "decode" abstract, symbolic angle information from raw pixels, showcasing a strong visual-symbol grounding capability.

Final Convergence Result:

--- Step 24000 ---
Train Loss: 0.016392
Eval Loss: 0.012753

4.4. Image Reasoning Capability

This section will demonstrate the paradigm's pure image reasoning capability, distinguishing it from symbolic rule learning and algorithm fitting capabilities.

4.4.1. Geometric Reasoning and Construction: Predicting a Triangle's Incircle

Task Description: The model's input is an image of a randomly generated green triangle. The task is to generate the image of its mathematically unique and correct red incircle.
Input Format: 224x224 image
Output Format: 224x224 image
Loss: MSELoss
Training Configuration: NVIDIA 4090 GPU
Dataset Generation Script: generate_triangle_to_incircle.py
Dataset: incircle_dataset folder
Training Code: train_image2image.py
Model Used: Swin Transformer + UNet
Training Set Size: 150,000
Input Space Size Estimation: Estimated to be approximately 2.8e+12 using a Monte Carlo method.
Training Set Size as a Percentage of Input Space: Approximately 5.4e-8

Task-Related Design Philosophy and Discussion:

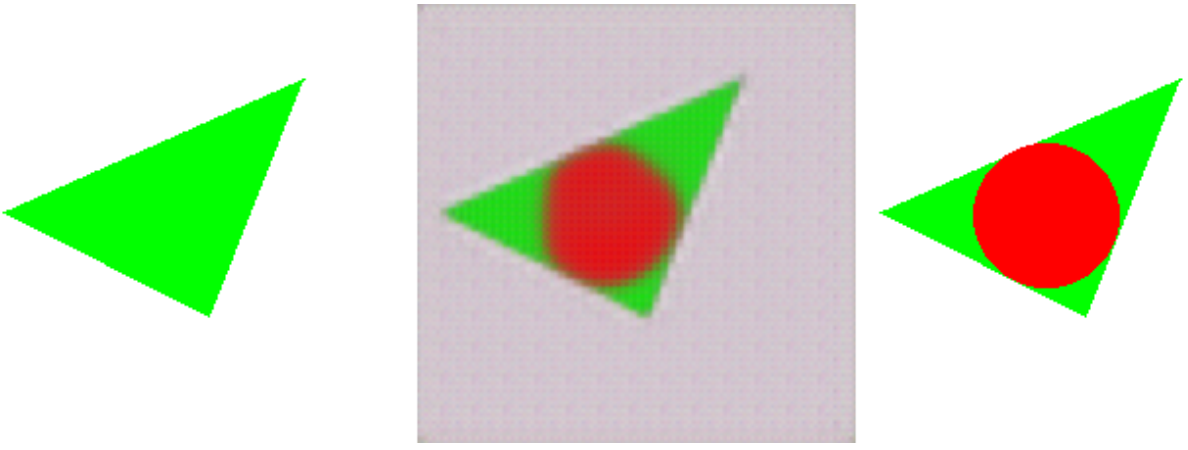
1.To further test the pure image reasoning capability of the Transformer, separating it from symbolic rule learning and algorithm fitting, I designed this incircle experiment. Of course, in theory, it's possible to recognize the positions of the 3 vertices, use a series of mathematical formulas to calculate the incircle's position, and then draw it. However, in practice, this is almost certainly not the actual process. Therefore, the purpose of this task is to prove the Transformer's image reasoning ability. Given the analog nature of images, this reasoning ability has great potential to be extended to larger domains, such as purely experimental observation data. However, it is easy to imagine that since real experimental observation data will inevitably have noise, it would devolve back to the previous pattern recognition deep learning paradigm. This issue will be discussed later.
2.The most astonishing thing about this task and the related series of tasks that follow is that they show, in a very intuitive form, the evolution of a "machine mind" during training. The way the model approaches and learns the rule, and the characteristics it exhibits in the process, are very similar to some form of intelligence. The visualization of the training process is presented as eval images.
3.Some task parameters:
Image Resolution: 224x224

Experimental Results and Analysis: The final training results show that the model has completely learned the rule of the incircle. This indicates that the model has image reasoning capabilities. In this particular case, the possibility of interpolating from other training set examples cannot be ruled out. However, other subsequent tasks do rule out this possibility, as those tasks cannot be explained by interpolation, and they use the exact same model and hyperparameter configuration. At the same time, the model's learning speed is very fast, providing a direct observation of the formation process of a machine mind.

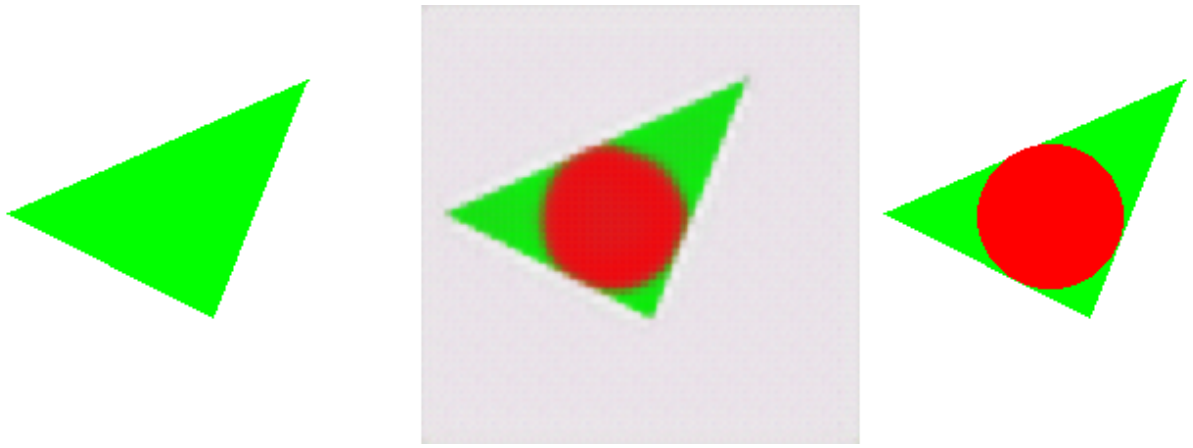
Experimental Process Showcase:

Left is the input image, right is the ground truth, and middle is the generated image.

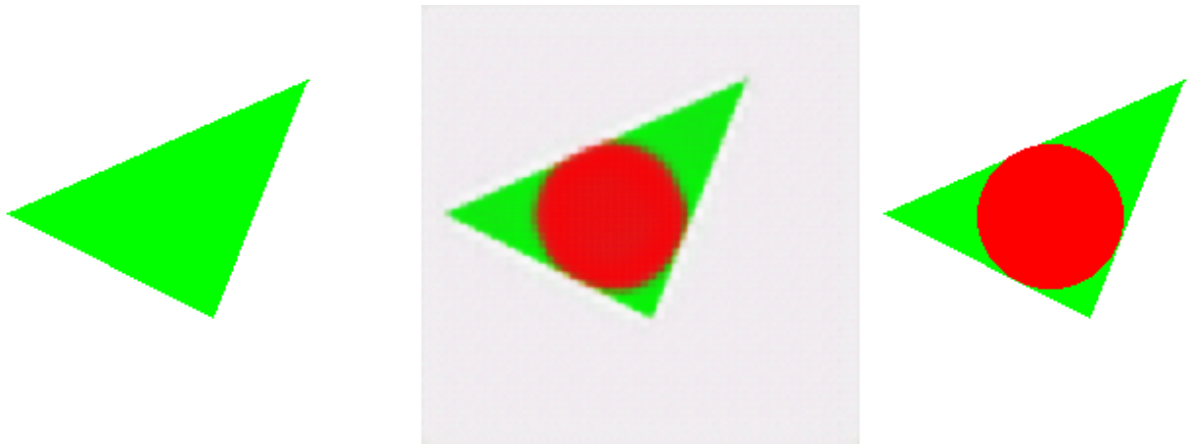
200 steps



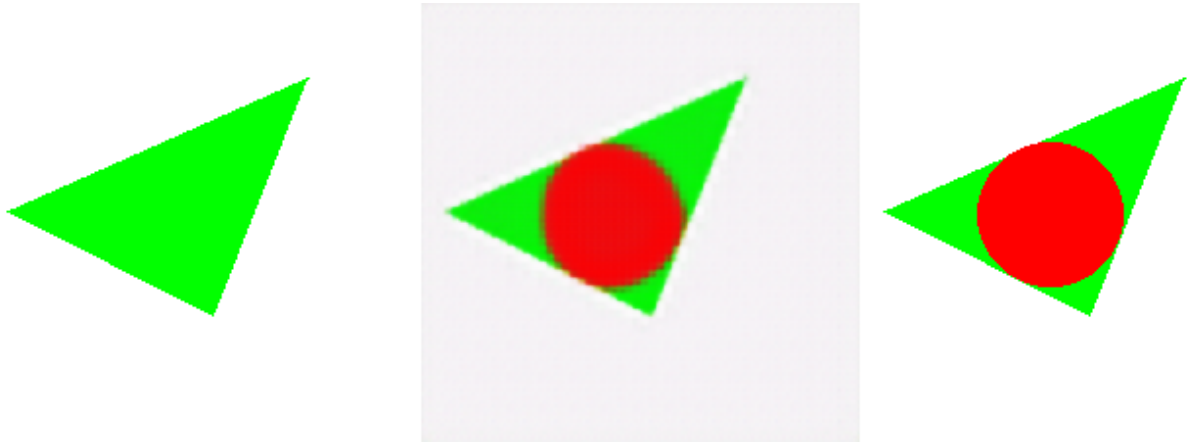
400 steps



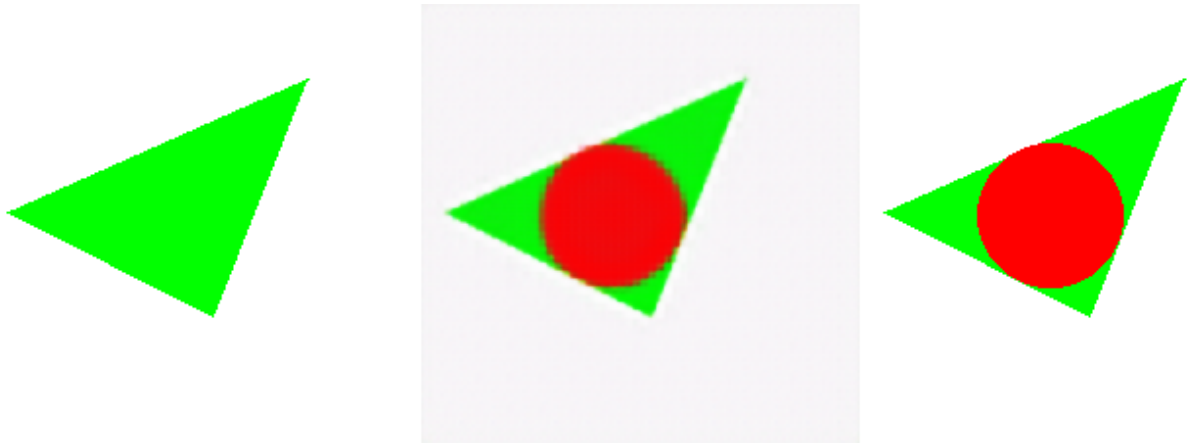
600 steps



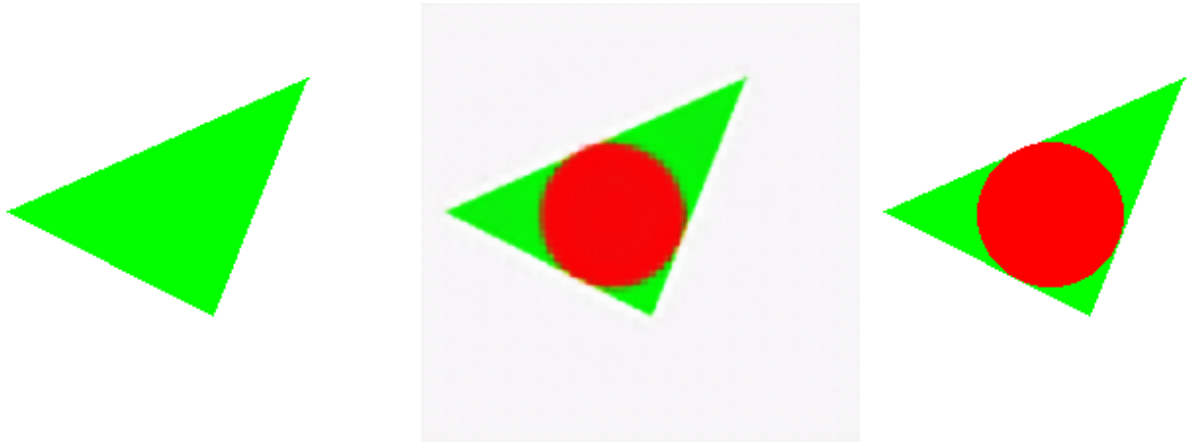
800 steps



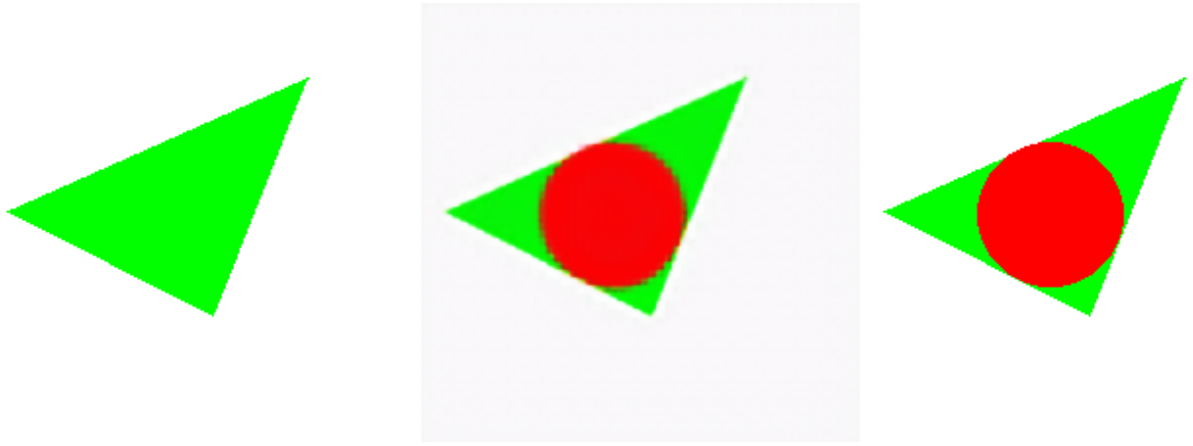
1000 steps



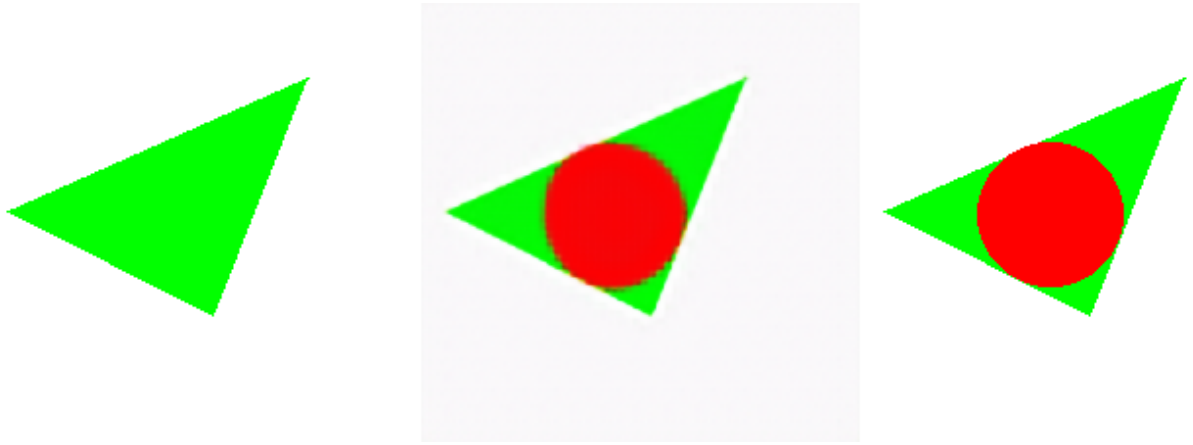
1200 steps



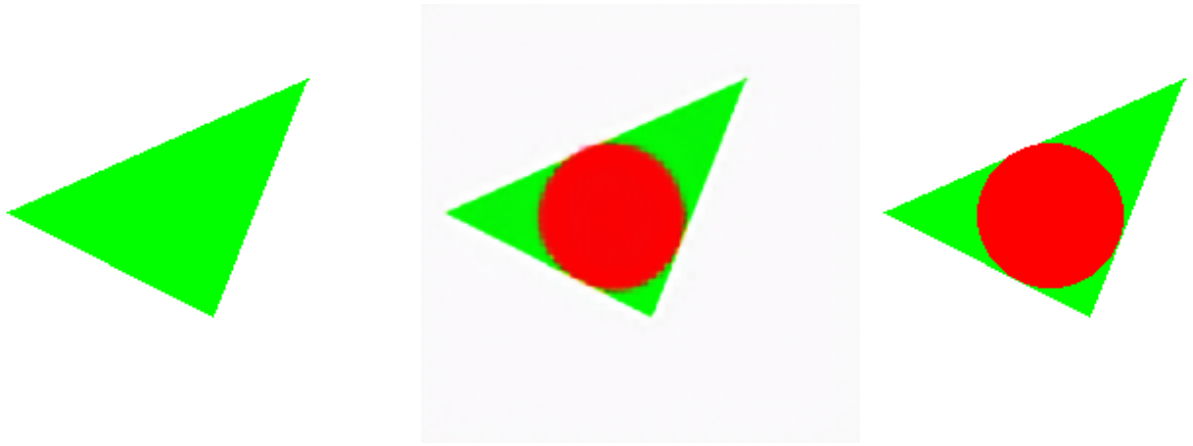
1400 steps



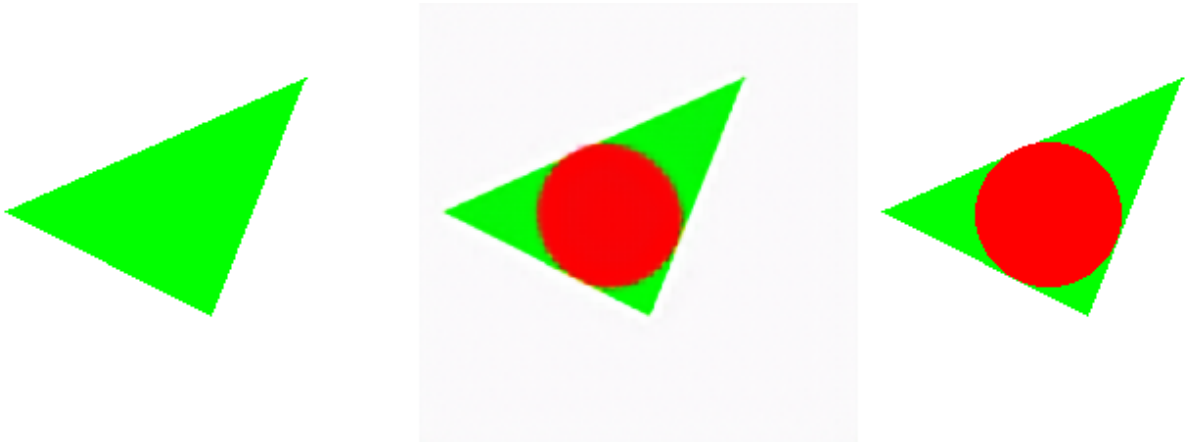
1600 steps



1800 steps

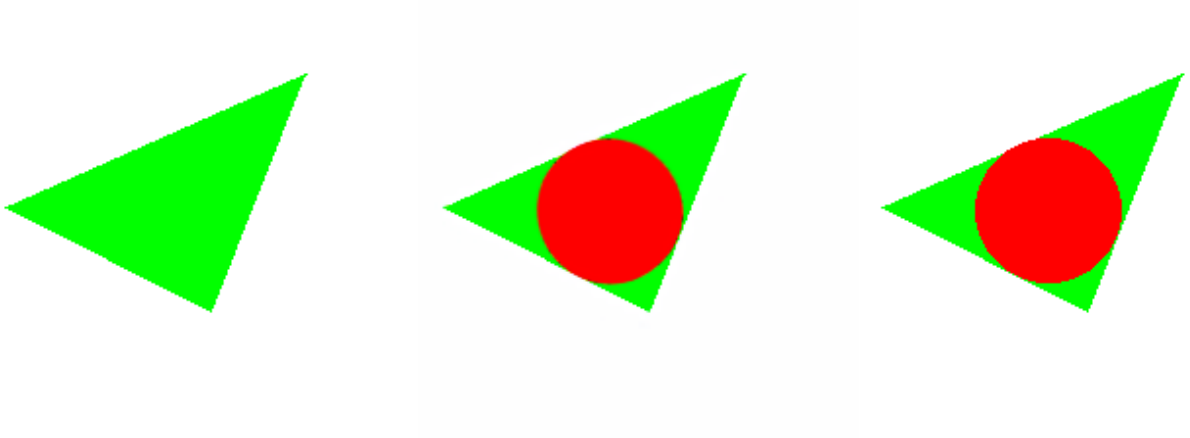


2000 steps



As the rate of change slows down, the final training result is shown directly.

14000 steps



4.4.2. Geometric Reasoning and Construction: Plane Tessellation

Task Description: The model's input is an image of a randomly generated green triangle. The output image is a plane tessellation of the green triangle, where green and red triangles are adjacent and alternate.
Input Format: 224x224 image, containing a randomly generated green triangle.
Output Format: 224x224 image, which is the plane tessellation result of the input image, with green and red triangles adjacent and alternating.
Loss: MSELoss
Training Configuration: NVIDIA 4090 GPU
Dataset Generation Script: generate_triangle_to_tessellation.py
Dataset: tessellation_dataset_224 folder
Training Code: train_image2image.py
Model Used: Swin Transformer + UNet
Training Set Size: 150,000
Input Space Size Estimation: Estimated to be approximately 2.71e10 using a Monte Carlo method.
Training Set Size as a Percentage of Input Space: Approximately 5.5e-6

Task-Related Design Philosophy and Discussion:

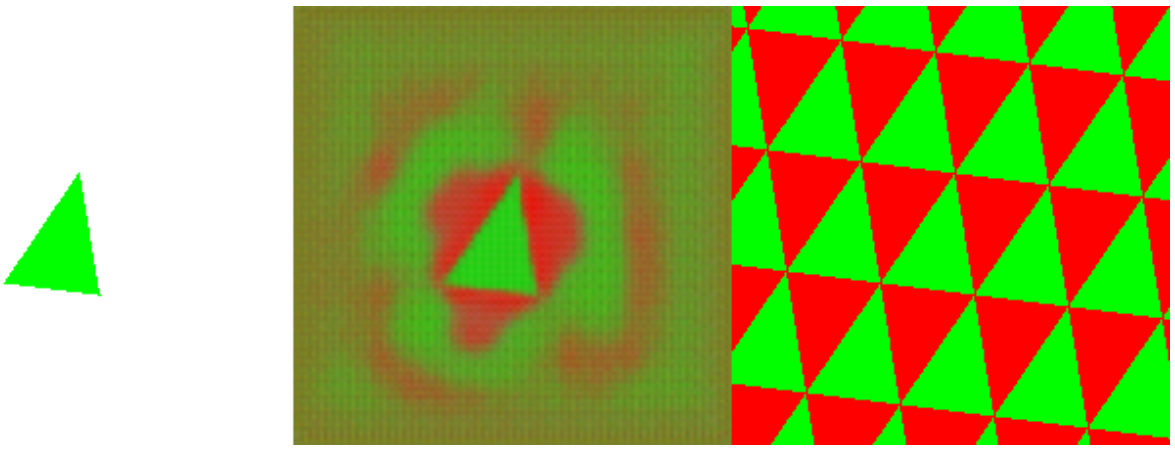
1.This task can completely rule out the possibility of solving the task through interpolation rather than learning the precise rule. The reason is that, far from the initial triangle in the plane tessellation, the errors from interpolation would gradually accumulate to the point where it would be impossible to fit the rule using a naive interpolation method.
2.To ensure the task difficulty is appropriate, the shape and size of the input green triangle were constrained. See the dataset generation script code for details.
3.Some task parameters:
Image Resolution: 224x224

Experimental Results and Analysis: The final training results show that the model has completely learned the rule of plane tessellation. This indicates that the model has image reasoning capabilities and largely rules out the possibility that the model is performing naive interpolation. At the same time, the model provides a direct observation of the formation process of a machine mind. In this example, a very clear learning process extending outwards from the input triangle is evident.

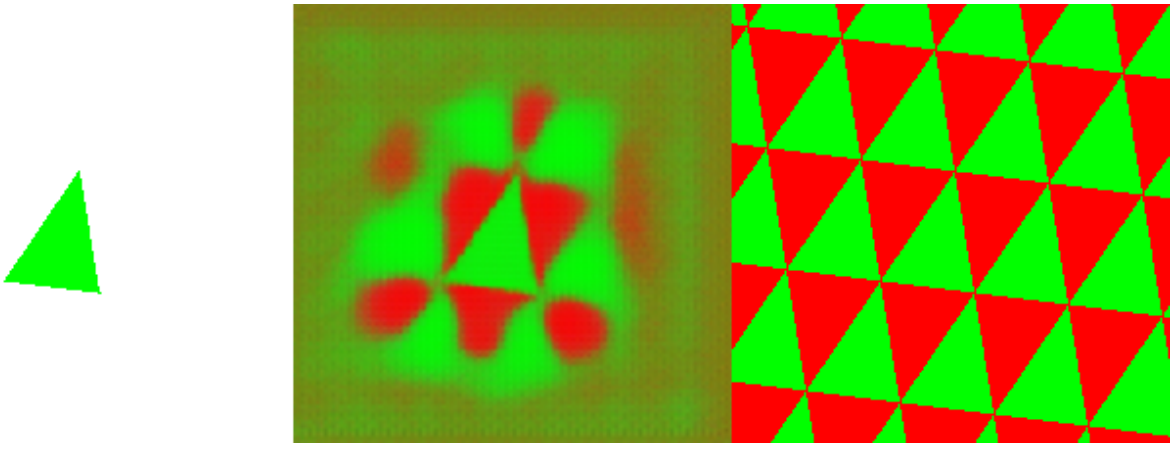
Experimental Process Showcase:

Left is the input image, right is the ground truth, and middle is the generated image.

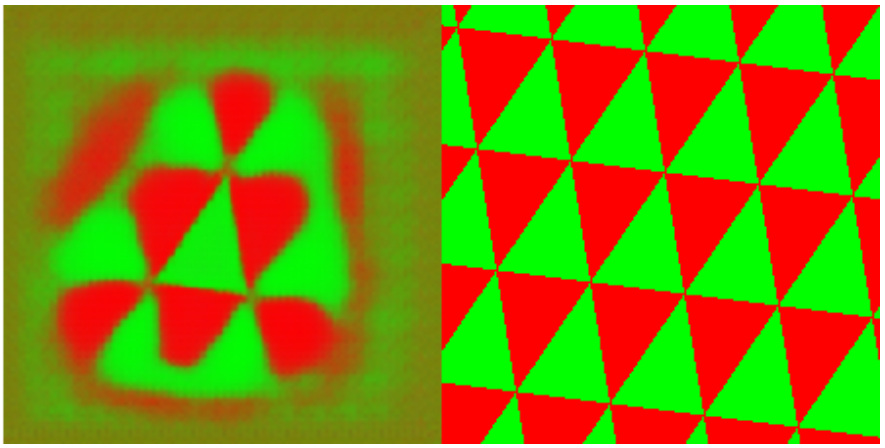
200 steps



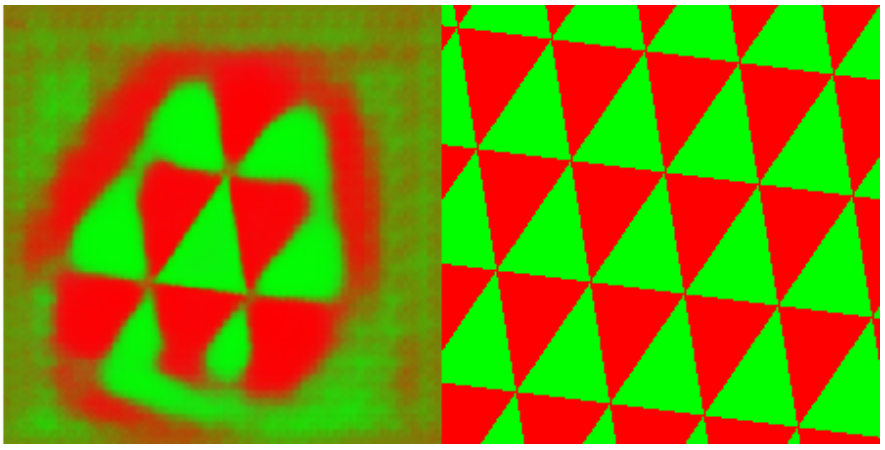
400 steps



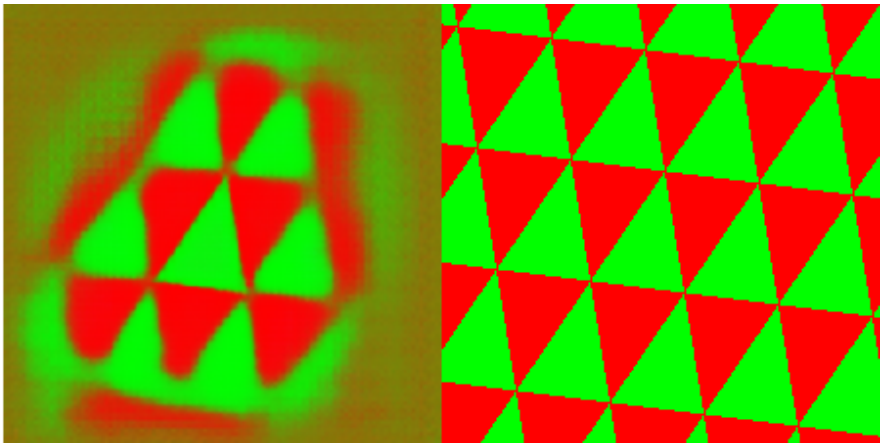
600 steps



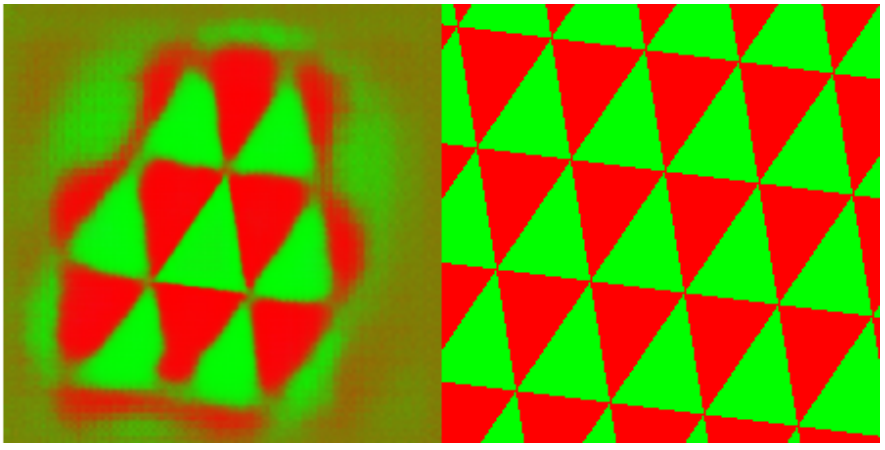
800 steps



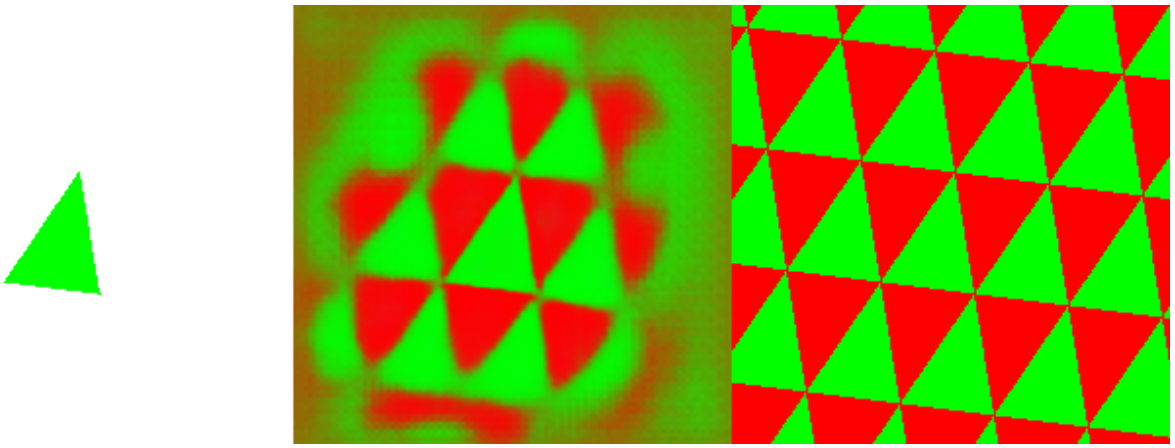
1000 steps



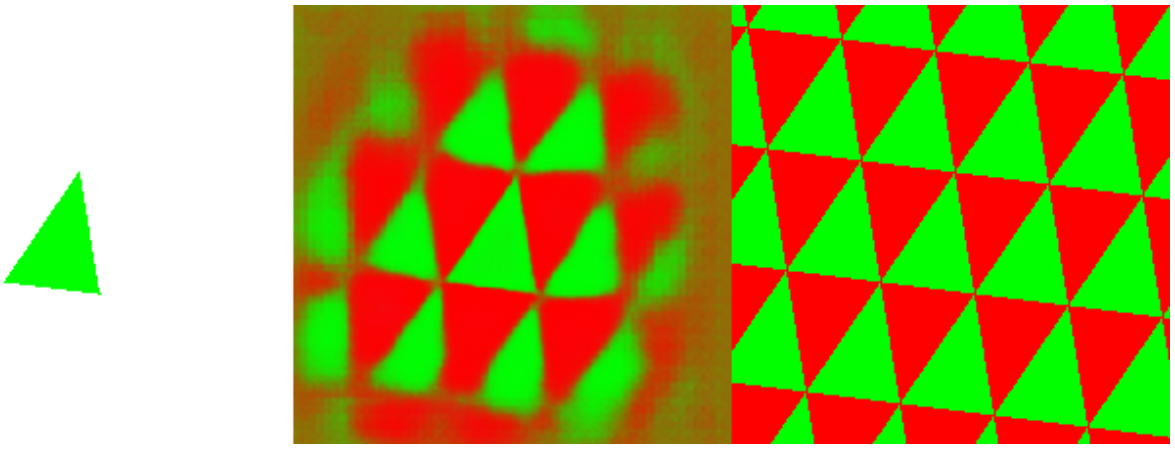
1200 steps



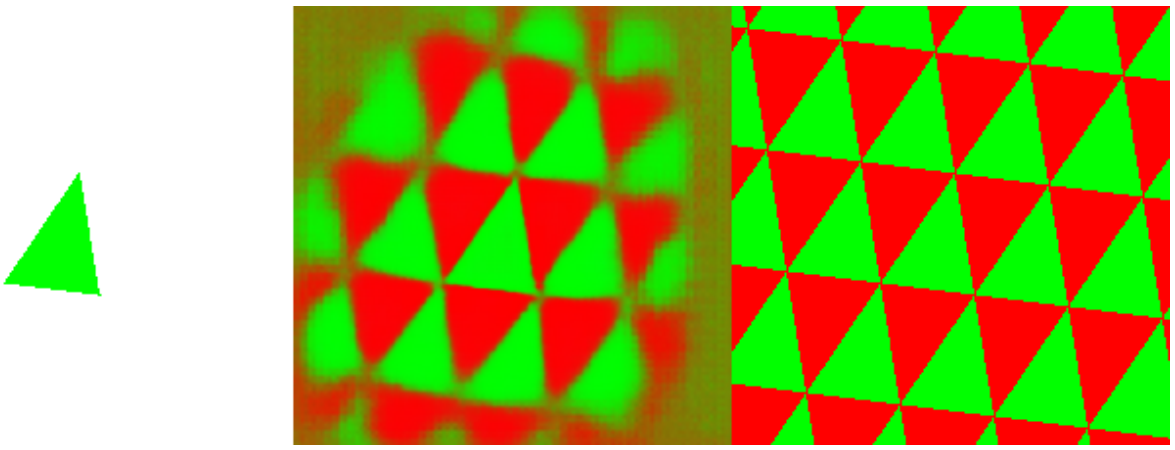
1400 steps



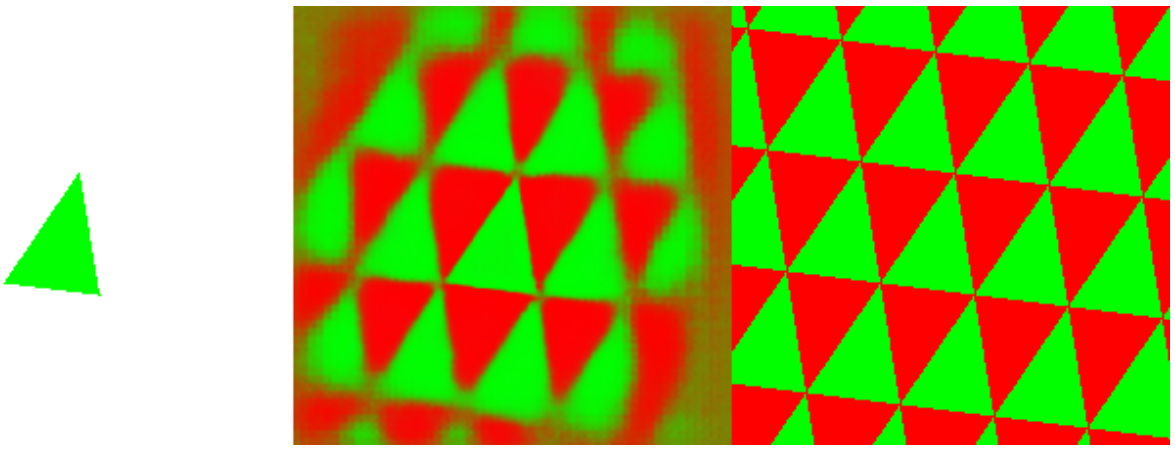
1600 steps



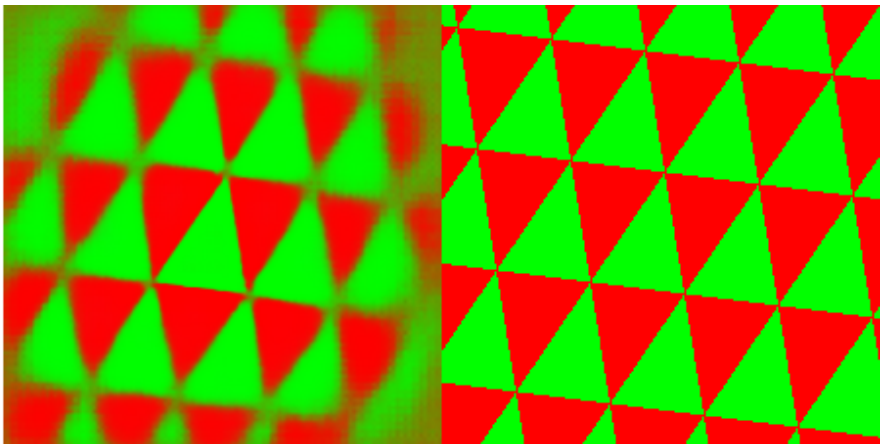
1800 steps



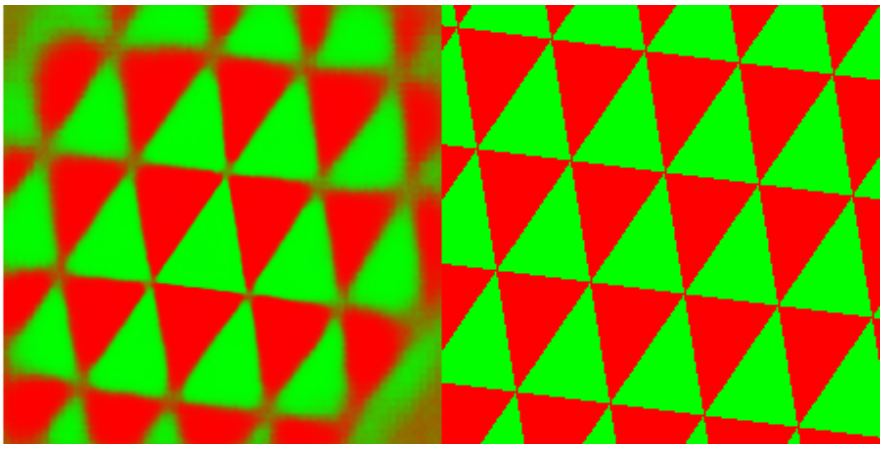
2000 steps



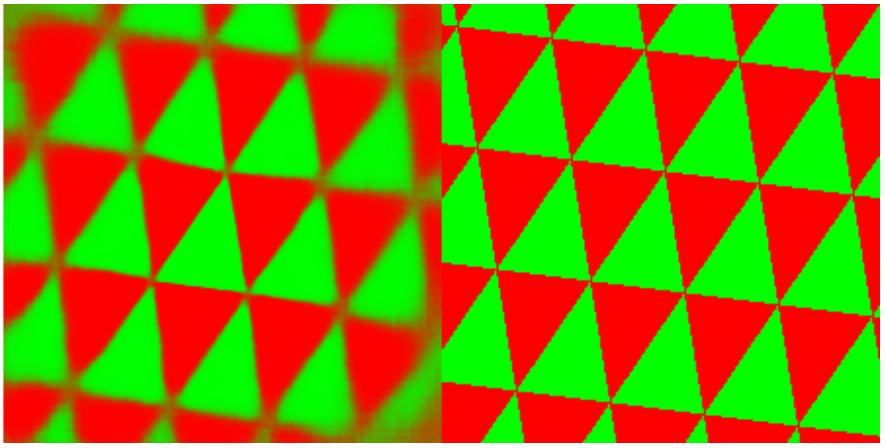
3000 steps



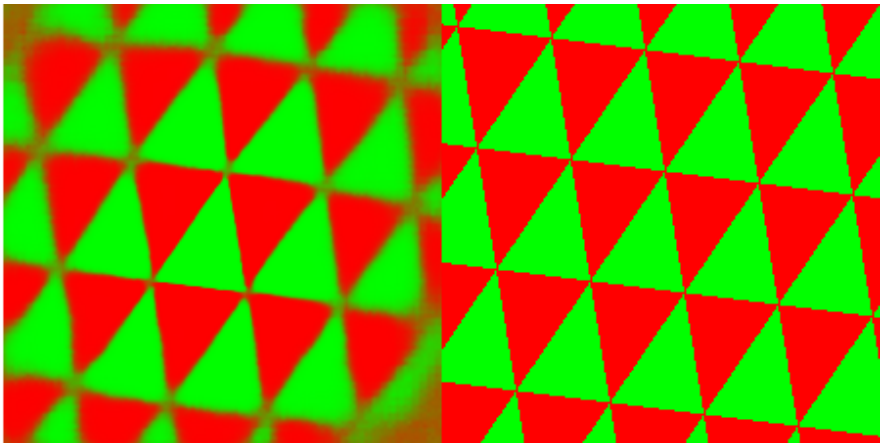
4000 steps



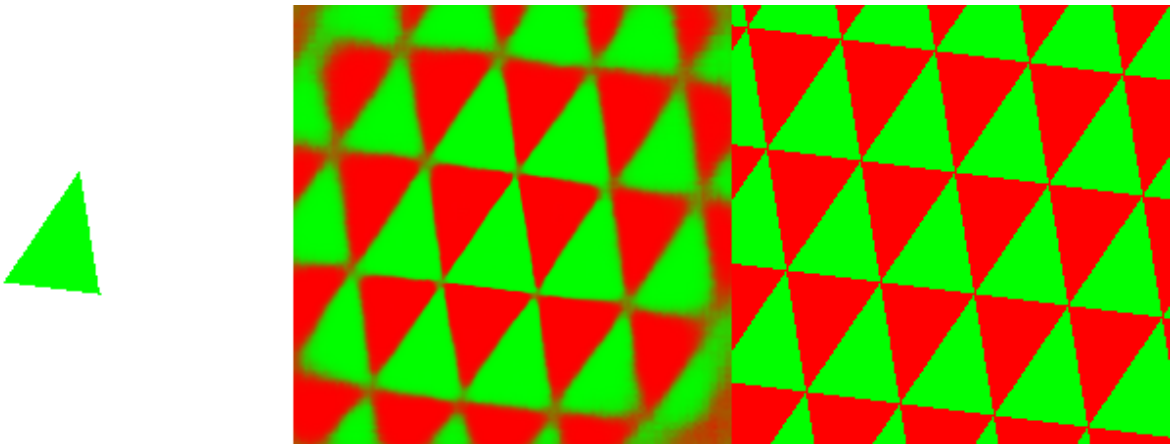
5000 steps



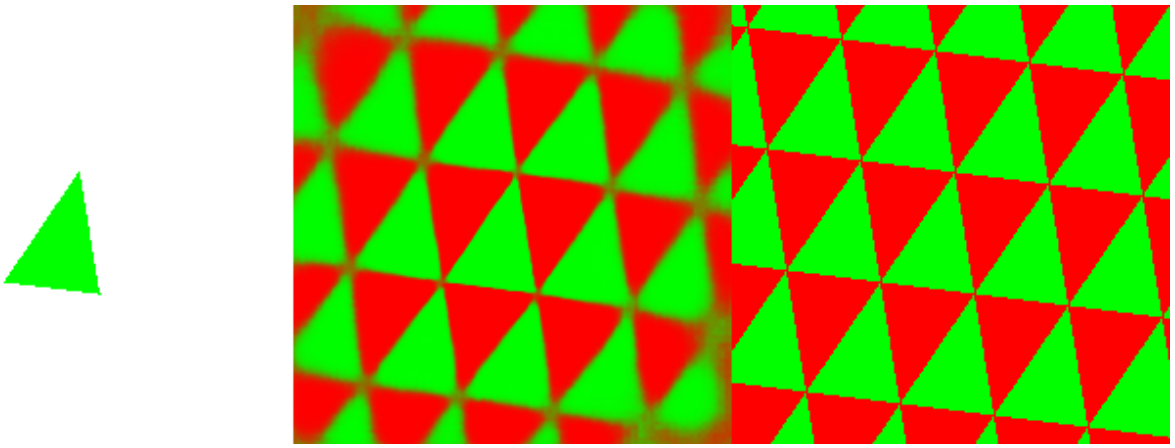
6000 steps



7000 steps

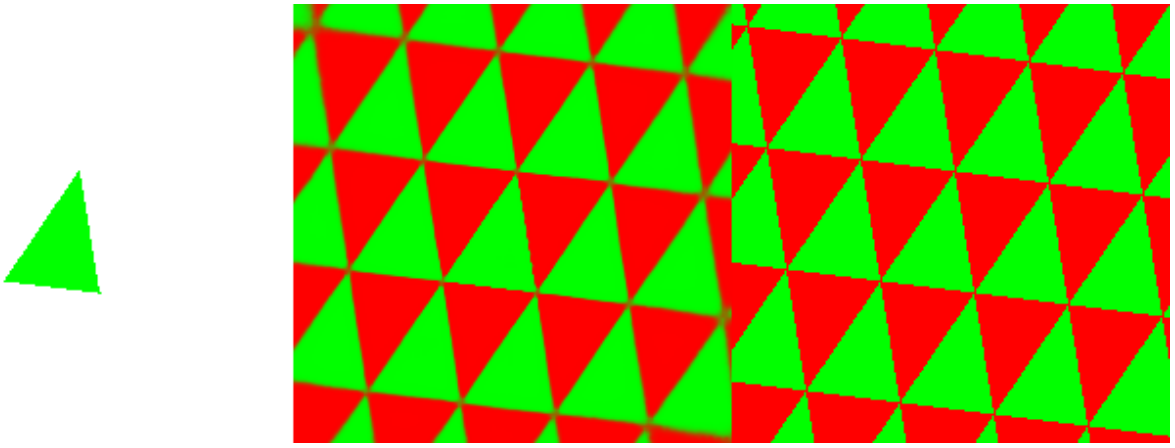


8000 steps



As the rate of change slows down, the final training result is shown directly.

150000 steps



4.5. Symbolic Reasoning to Image Generation

This section will demonstrate the paradigm's ability to process symbolic input, learn the corresponding rules, and generate the corresponding images.

4.5.1. Cellular Automata

Task Description: The model learns to predict and output the state of a 1D cellular automaton after a given number of transformations according to a given rule.
Input Format: A 36-character string consisting of "0"s or "1"s, representing the initial state of the cellular automaton.
Output Format: 240x240 image

Loss: MSELoss
Training Configuration: NVIDIA 4090 GPU
Dataset Generation Script: generate_cellular_automata_1d_to_grid_image.py
Dataset: ca_render_dataset_240 folder
Training Code: train_qwen2_text2image.py
Model Used: Transformer + UNet
Training Set Size: 300,000
Input Space Size Estimation: 2^36 = 68,719,476,736
Training Set Size as a Percentage of Input Space: 4.4e-6

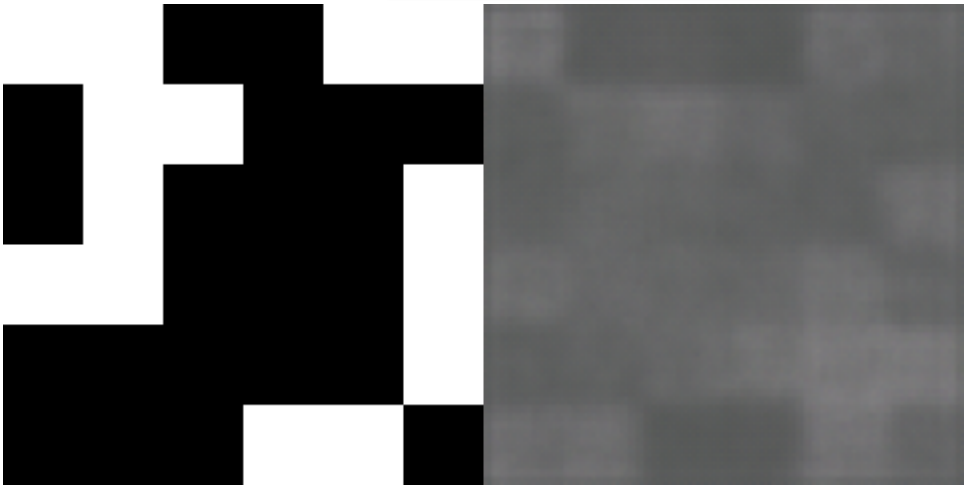
Task-Related Design Philosophy and Discussion:

1.This task is designed to test the Transformer's reasoning ability in symbolic-to-image generation. This mode is not text-to-image in the traditional sense; rather, the focus is on testing its reasoning capability.
2.In the output image, the 36-cell 1D cellular automaton state is arranged in a row-major order into a 6x6 checkerboard image, where black represents 1 and white represents 0.
3.Some task parameters:
Grid dimensions: 6x6
Grid cell size: 40x40 pixels
Image Resolution: 240x240
Cellular automaton rule: rule 110
Number of rule evolution layers: 3

Experimental Results and Analysis: The final training results show that the model has completely learned the rules of the cellular automaton. This indicates that this symbolic-to-image generation mode can also preserve the Transformer's reasoning ability. The model learns very quickly, providing a direct observation of the formation process of a machine mind. The entire learning process cannot be simply explained by some kind of logic, which further proves that the learning of this paradigm does not necessarily follow the logic that people might assume, but rather a more naive connectionist approximation of the discrete.

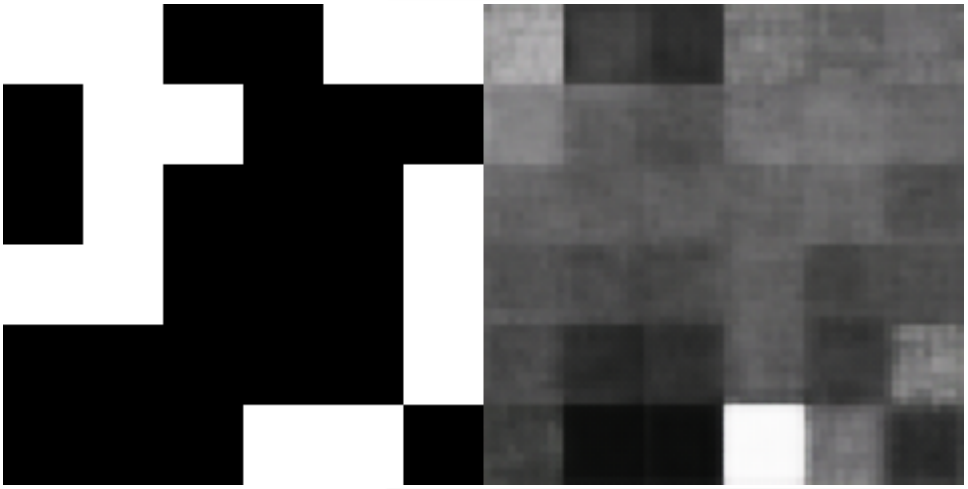
Left is the ground truth, right is the model-generated image.

100 steps, symbolic input: `111110011010001000111010000110`



200 steps, symbolic input: `11111001101000100011101000011O`

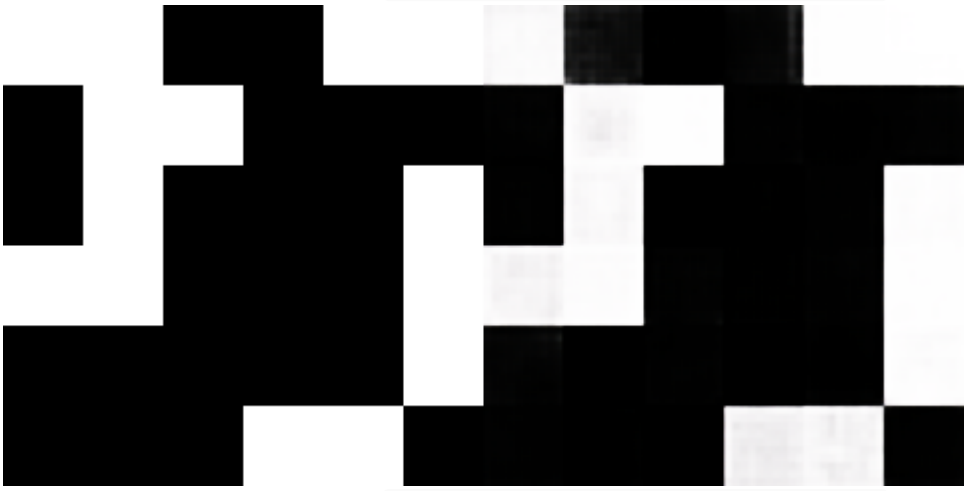
300 steps, symbolic input: `11111001101000100011101000011O`



400 steps, symbolic input: `11111001101000100011101000011O`

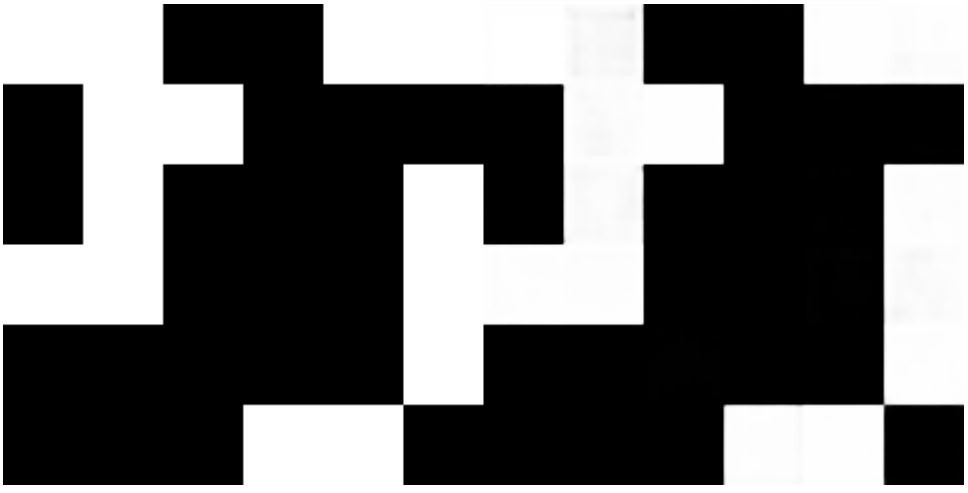
500 steps, symbolic input: `11111001101000100011101000011O`

600 steps, symbolic input: `11111001101000100011101000011O`



700 steps, symbolic input: `11111001101000100011101000011O`

800 steps, symbolic input: `11111001101000100011101000011O`



4.5.2 Rotating Cube

Task Description: The model learns to predict and output an image of a colored hexahedron after rotation, based on given rotation angles.
Input Format: A 24-character string of "0"s or "1"s, representing 3 rotation angles. Each consecutive 8-bit binary number represents one rotation angle.
Output Format: 256x256 image
Loss: MSELoss
Training Configuration: NVIDIA 4090 GPU
Dataset Generation Script: generate_cube_rotation_pillow_with_anchor.py
Dataset: cube_dataset_final_highlight folder
Training Code: train_qwen2_text2image.py
Model Used: Transformer + UNet
Training Set Size: 300,000
Input Space Size Estimation: 16,777,216
Training Set Size as a Percentage of Input Space: 1.79%

Task-Related Design Philosophy and Discussion:

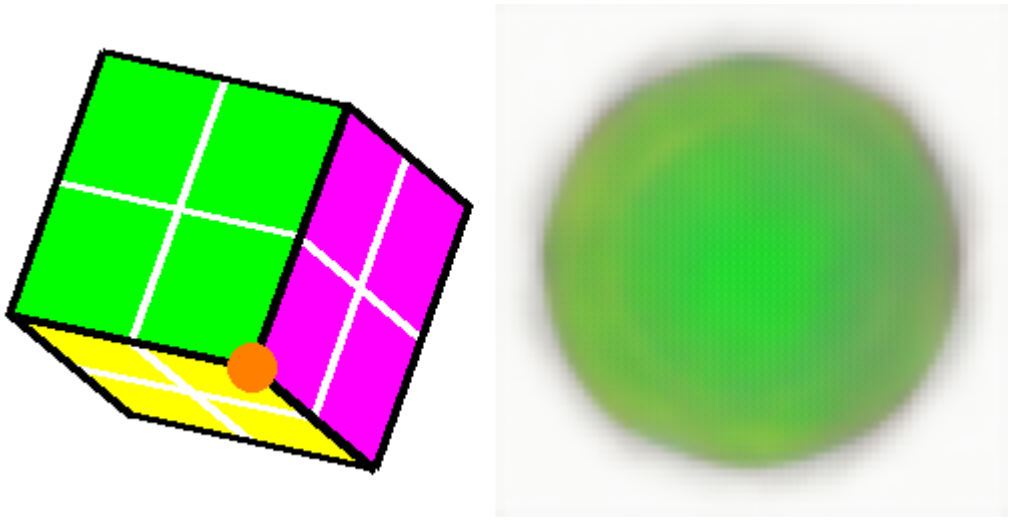
1.In this task, I encountered a specific difficulty: the model struggled to associate the vertices corresponding to the rotation angles with the symbolic input, leading to a failure to converge. The solution was to always highlight a specific vertex in the final step of generating the training set images, ignoring occlusion. After adopting this solution, the model was finally able to converge on this task. In another task, where I used symbols to indicate the position of a triangle and the output was a plane tessellation, the model faced a similar difficulty; it couldn't figure out which green triangle in the output image corresponded to the symbolic input. The solution was similar: I changed the color of the triangle indicated by the symbolic input to black in the output image, and everything worked normally. This seems to be a rather typical problem: the model has difficulty determining how to map symbolic input to which feature of the output image. The solutions should also be similar.
2.The white cross-shaped lines on each face are mainly for aesthetics but may also have some auxiliary training effect.
3.The 3 rotation angles in the symbolic input, namely pitch, yaw, and roll, can be referred to in the dataset generation code for specific correspondence.
4.I actually have another version of the model that does not use LoRA fine-tuning on qwen2-0.5b but is a Transformer with fewer total parameters than qwen2-0.5b but more LoRA parameters than qwen2-0.5b. Practice has shown that its learning performance is not as good as that of qwen2-0.5b. I consider this an interesting phenomenon.
5.Some task parameters:
Image Resolution: 256x256

Experimental Results and Analysis: The final training results show that the model has completely learned this task. This indicates that this symbolic-to-image generation mode can also preserve the Transformer's image reasoning ability. And since I used LoRA fine-tuning on qwen2-0.5b, it can be said that this image reasoning ability, at least the ability to generate

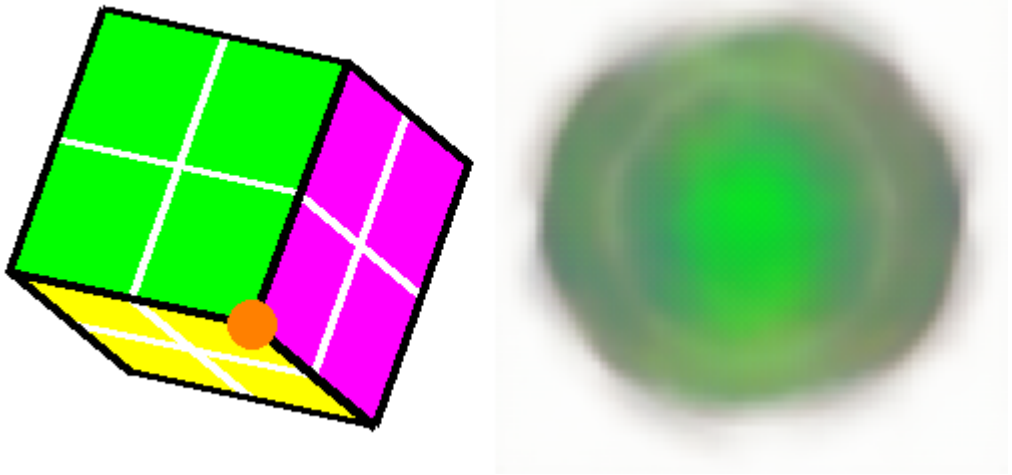
images, is not limited to image Transformers like Swin Transformer and ViT. The model's learning speed is not very fast, possibly because the task is relatively difficult. It provides a direct observation of the formation process of a machine mind. In this task, it can be seen that at the beginning, the model has not yet understood the rotation angles and simply learns to generate a circular shape and a rather uniform color fill. Later, the model gradually understands the concept of a rotating cube, and then slowly fills in the precise details.

Left is the ground truth, right is the model-generated image.

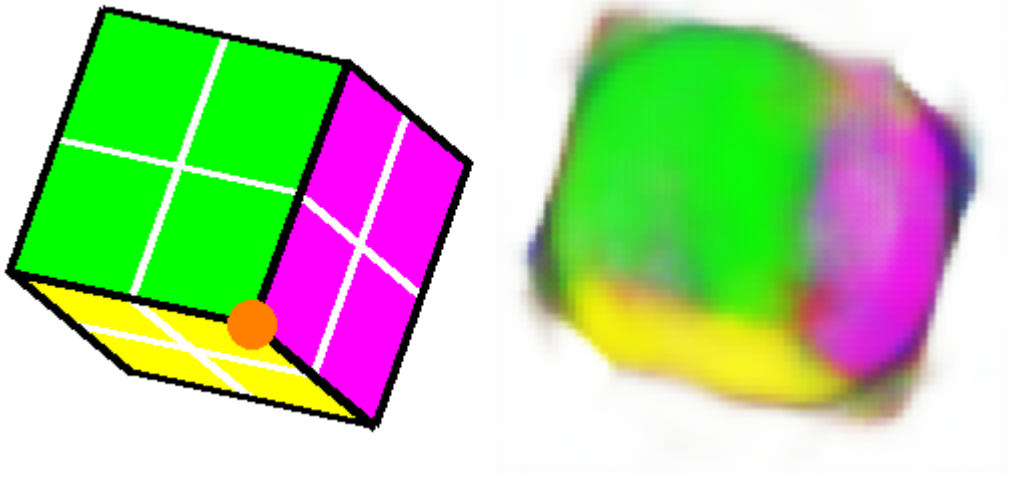
step 500, symbolic input: `100101101000100100110010`



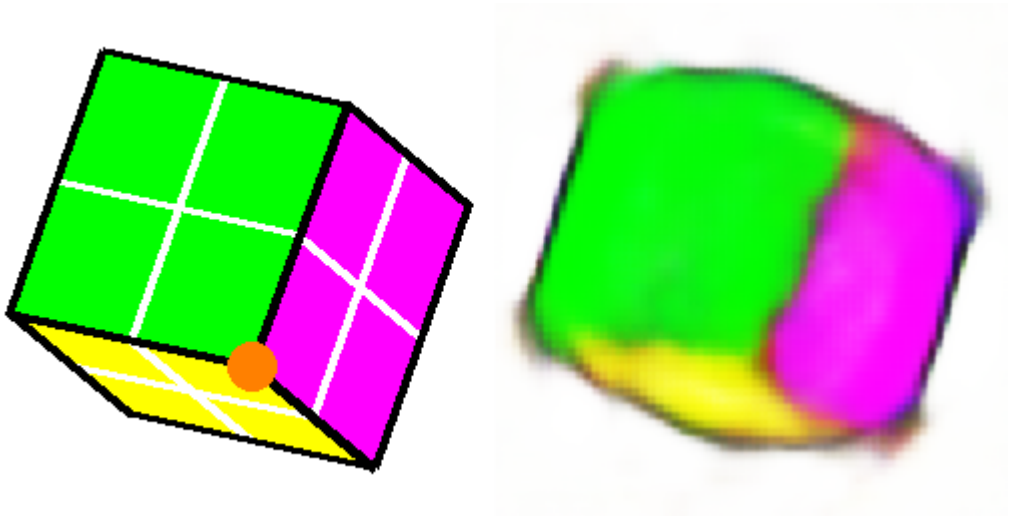
step 1000, symbolic input: `100101101000100100110010`



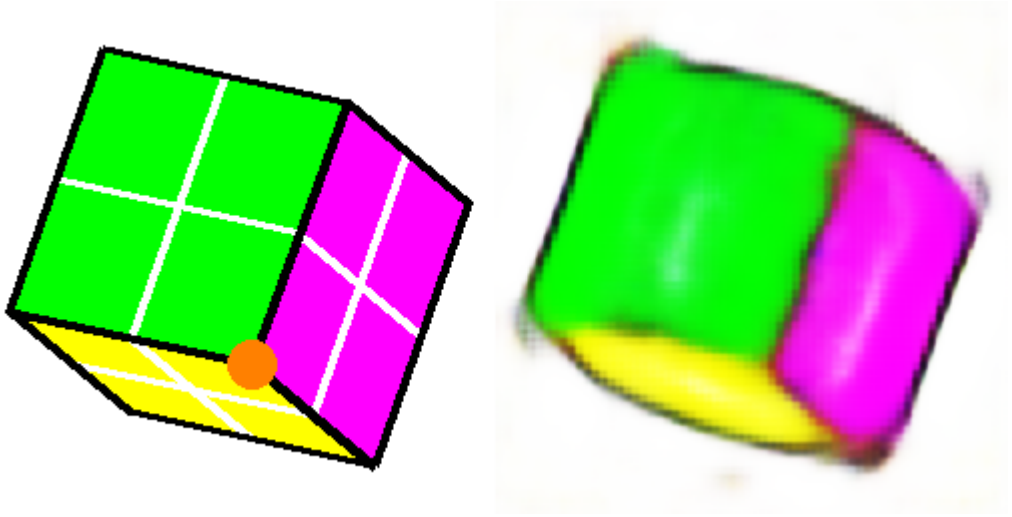
step 1500, symbolic input: `100101101000100100110010`



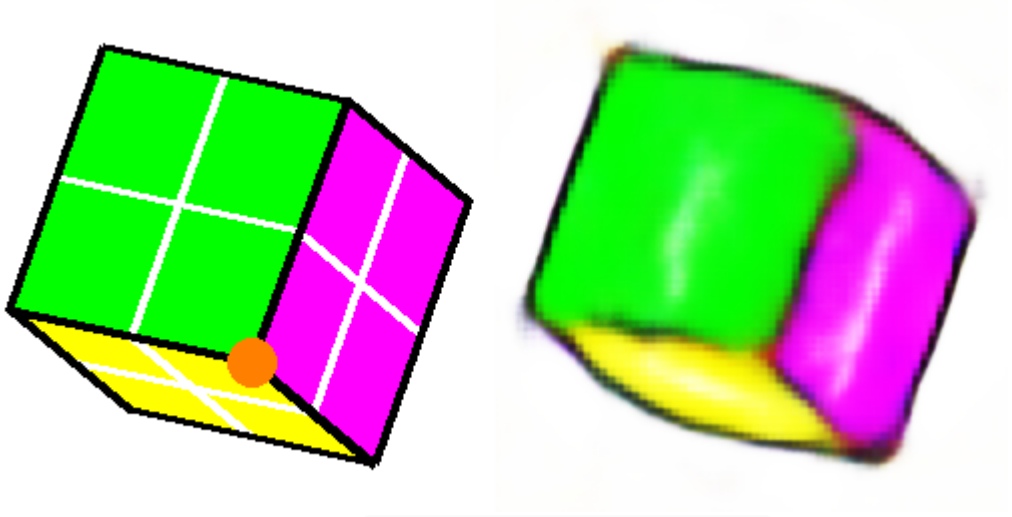
step 2000, symbolic input: `10010110100010010011001O`



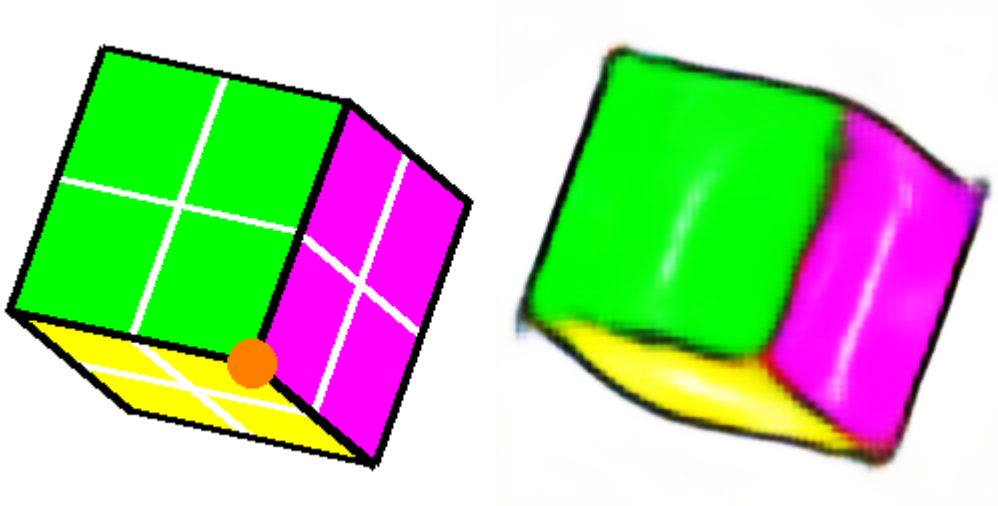
step 2500, symbolic input: `10010110100010010011001O`



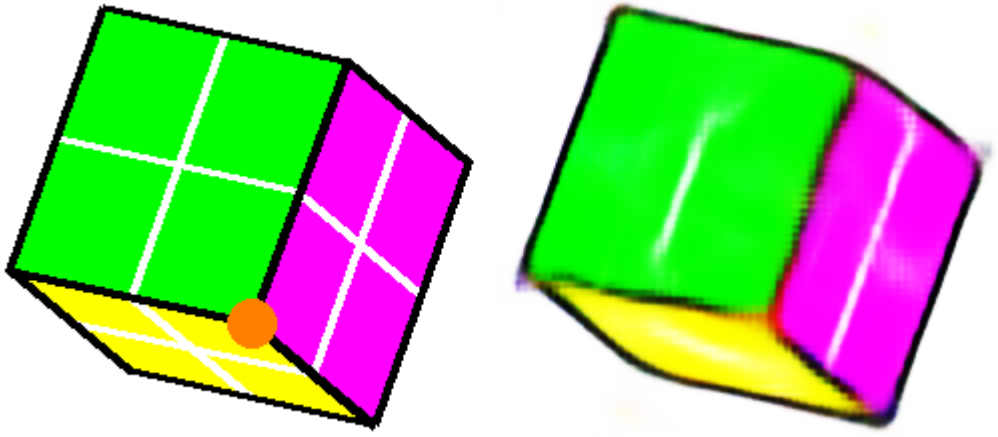
step 3000, symbolic input: `10010110100010010011001O`



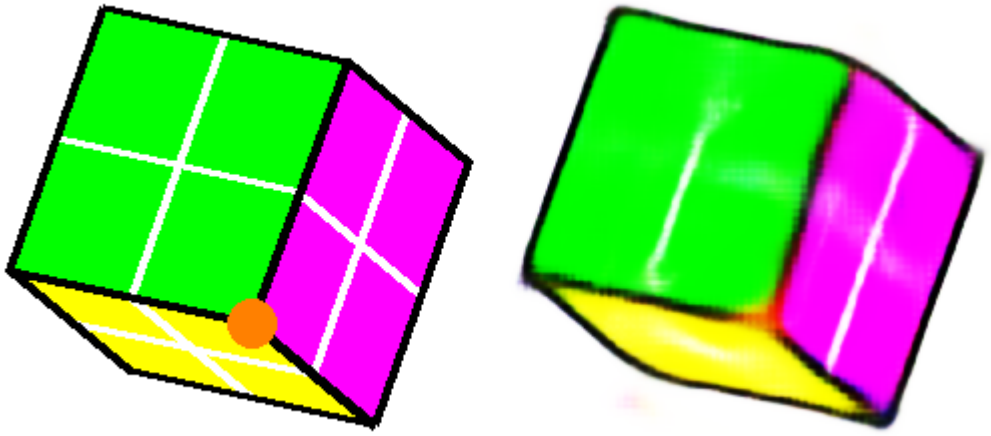
step 4000, symbolic input: `10010110100010010011001O`



step 6000, symbolic input: `10010110100010010011010`

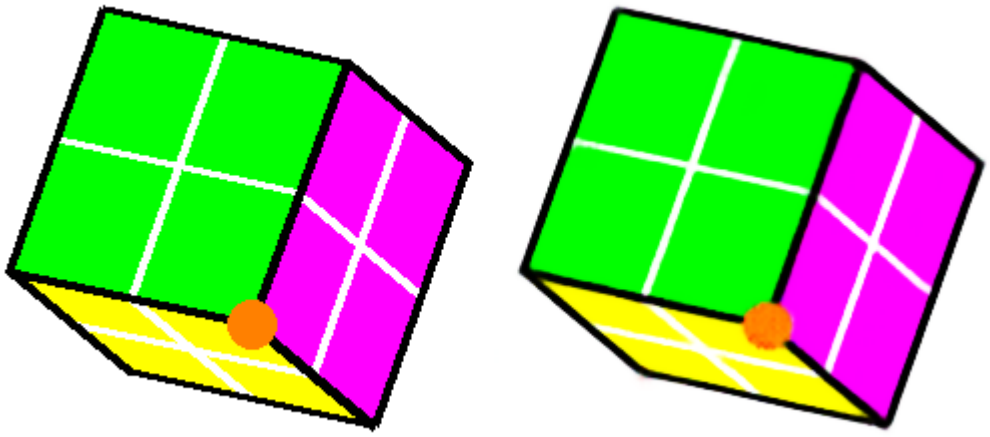


step 8000, symbolic input: `10010110100010010011010`



As the rate of change slows down, the final training result is shown directly.

step 133600, symbolic input: `10010110100010010011001O`



4.6. Physics Simulation

This section will demonstrate the paradigm's ability to simulate physical processes.

4.6.1 Catenary Curve

Task Description: Given the two endpoints of a catenary and another given point that the catenary passes through, the model learns to draw the catenary curve uniquely determined by the laws of physics (principle of minimum potential energy).
Input Format: 224x224 image
Output Format: 224x224 image
Loss: MSELoss
Training Configuration: NVIDIA 4090 GPU
Dataset Generation Script: generate_catenary_curve_from_points.py
Dataset: catenary_dataset_v5_CONSTRUCTIVE folder
Training Code: train_image2image.py
Model Used: Swin Transformer + UNet
Training Set Size: 150,000
Input Space Size Estimation: Estimated to be approximately 9.3e9 using a Monte Carlo method.
Training Set Size as a Percentage of Input Space: Approximately 1.61e-5

Task-Related Design Philosophy and Discussion:

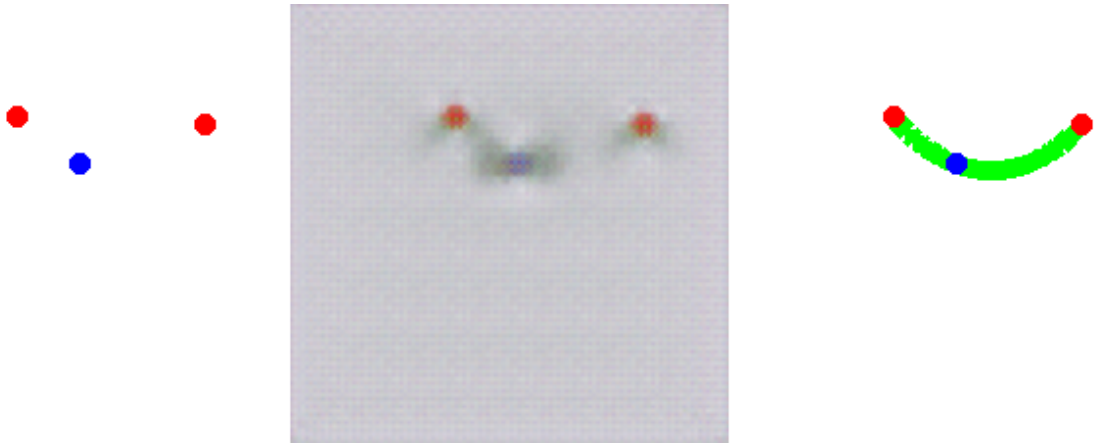
1.This task is to test the physics-related simulation capabilities of this paradigm, implemented through an image reasoning model.
2.Although there is a possibility of solving this problem through interpolation, the model and hyperparameter configuration used have also solved other problems that cannot be solved by interpolation, thus ruling out this possibility.
3.The reason for adding the constraint of a point the catenary passes through is mainly because setting a fixed length for the catenary or representing the length in some other way did not feel as good as this setup. Then, adding this constraint also increases the input space, further ruling out the possibility of memorization or interpolation.

4.Some task parameters:
Image Resolution: 224x224

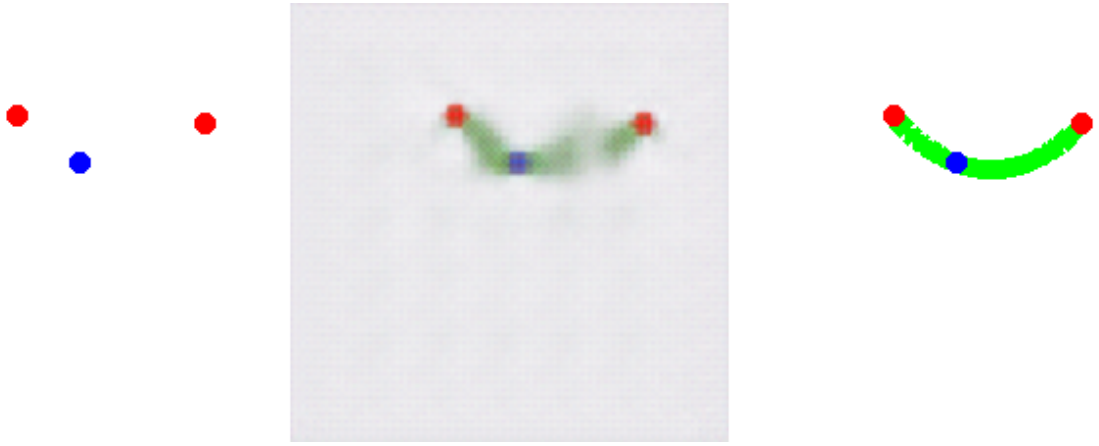
Experimental Results and Analysis: The final training results show that the model has completely learned the rules of the catenary curve. This indicates that the paradigm's learning ability has extended from pure mathematical and logical laws to simulating the fundamental physical laws of our real universe. However, the precision of this simulation needs further investigation. The model learns very quickly, providing a direct observation of the formation process of a machine mind. It can be seen that in this example, the learning process did not spend too much time on the trajectory of the catenary but mainly on optimizing the image output. It is also noted that the training set I generated has flaws, with aliasing on the output catenary images, while the model's output after convergence is relatively smooth. This phenomenon is worth further study.

Left is the input image, middle is the model-generated image, and right is the ground truth.

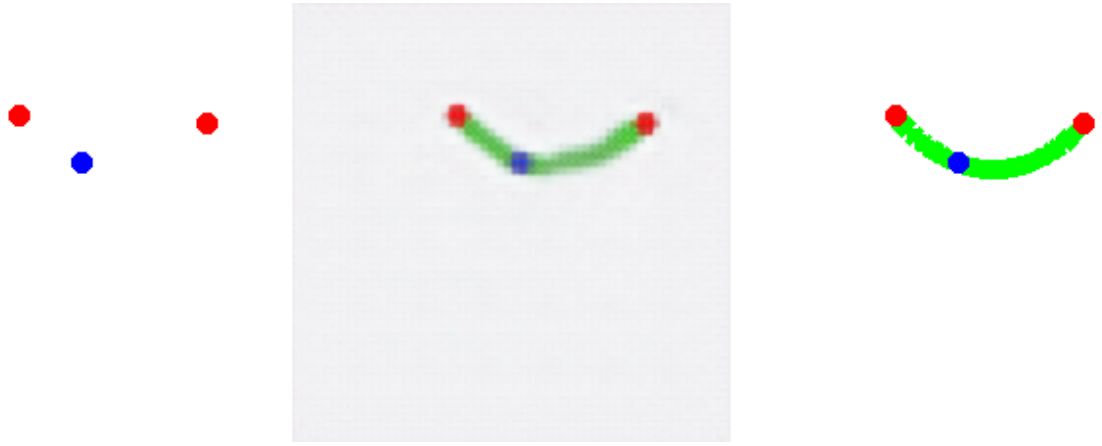
200 steps



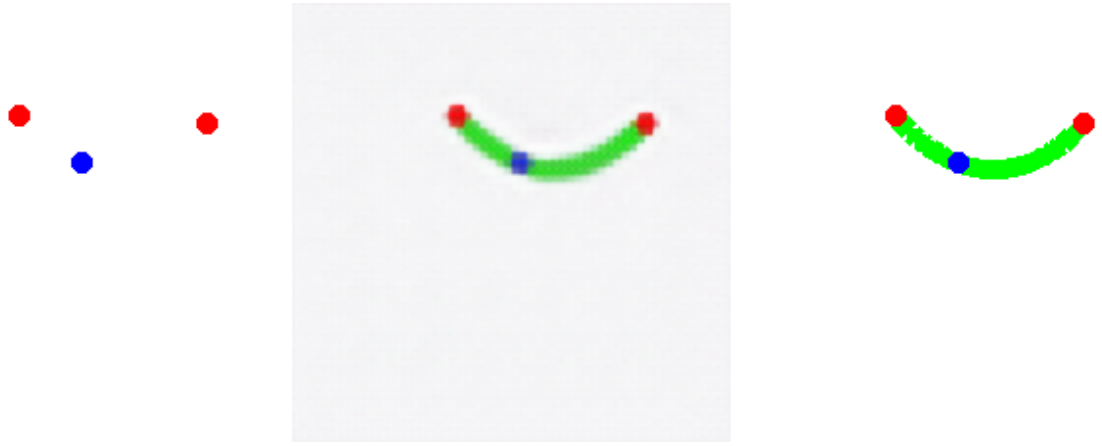
400 steps



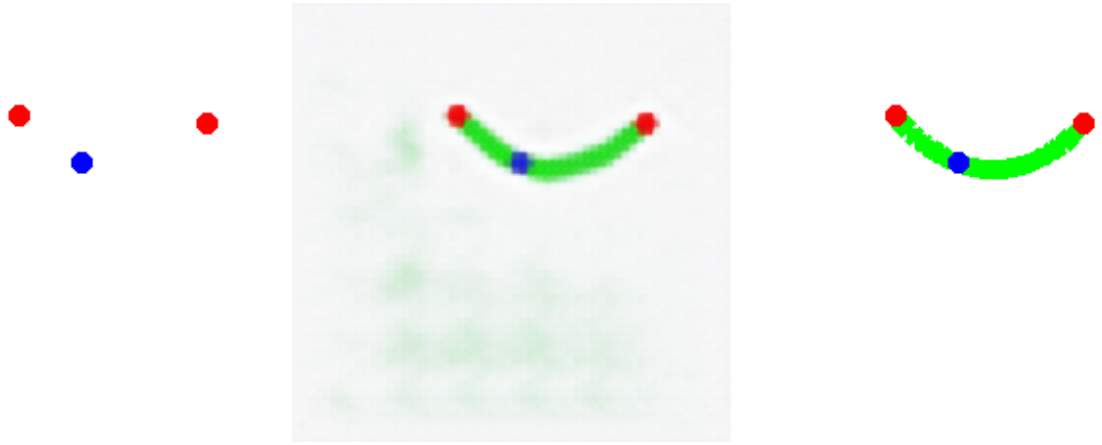
600 steps



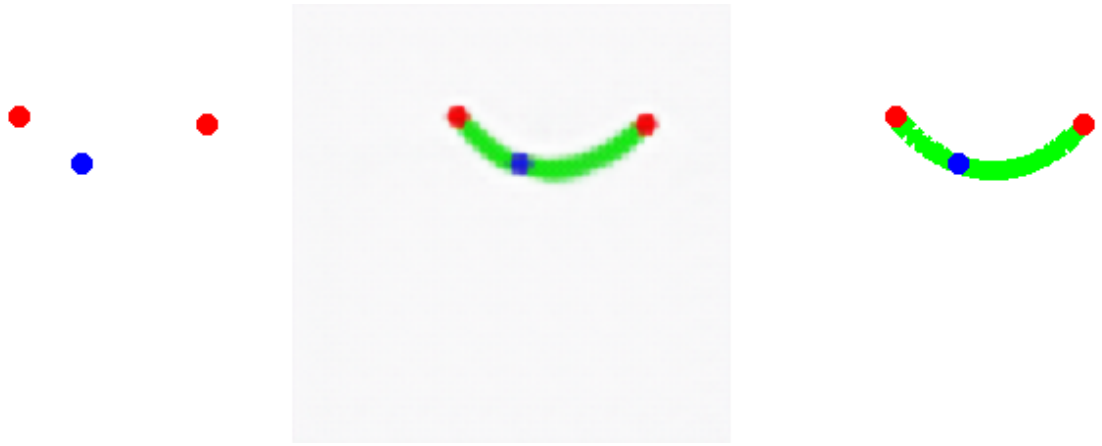
800 steps



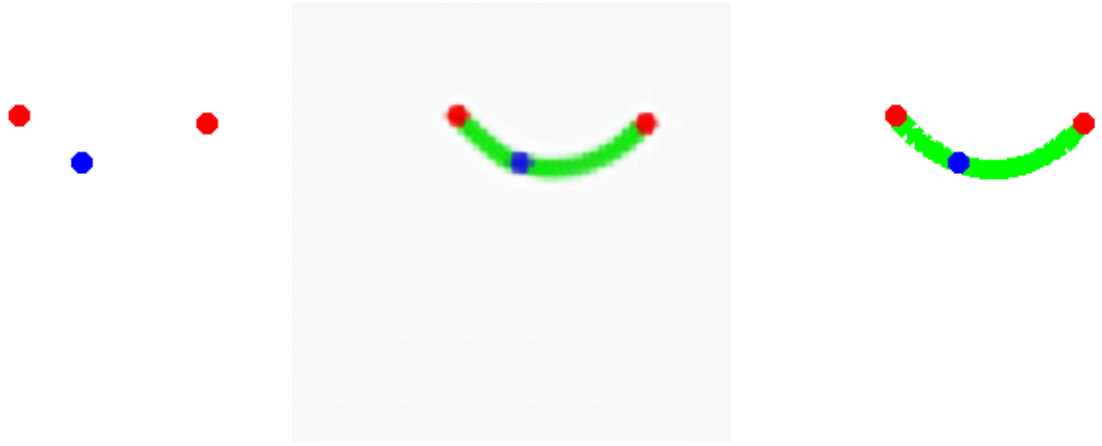
1000 steps



1200 steps



1400 steps



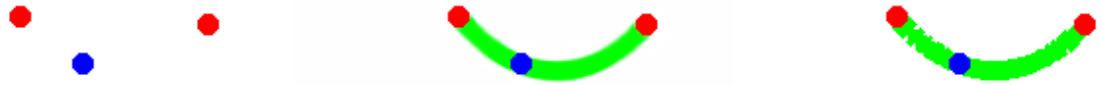
1600 steps

1800 steps

2000 steps

As the rate of change slows down, the final training result is shown directly.

19000 steps



4.6.2. Planetary Orbit Simulation

Task Description: Given the fixed position of a star, the initial position of a planet, its initial speed magnitude, and initial velocity direction, predict the planet's orbit.
Input Format: 224x224 image
Output Format: 224x224 image
Loss: MSELoss
Training Configuration: NVIDIA 4090 GPU
Dataset Generation Script: generate_orbital_path_from_initial_state.py
Dataset: orbital_dataset_256_separated_v3 folder
Training Code: train_image2image.py
Model Used: Swin Transformer + UNet
Training Set Size: 150,000
Input Space Size Estimation: Estimated to be approximately 2.8e11 using a Monte Carlo method.
Training Set Size as a Percentage of Input Space: Approximately 5.36e-7

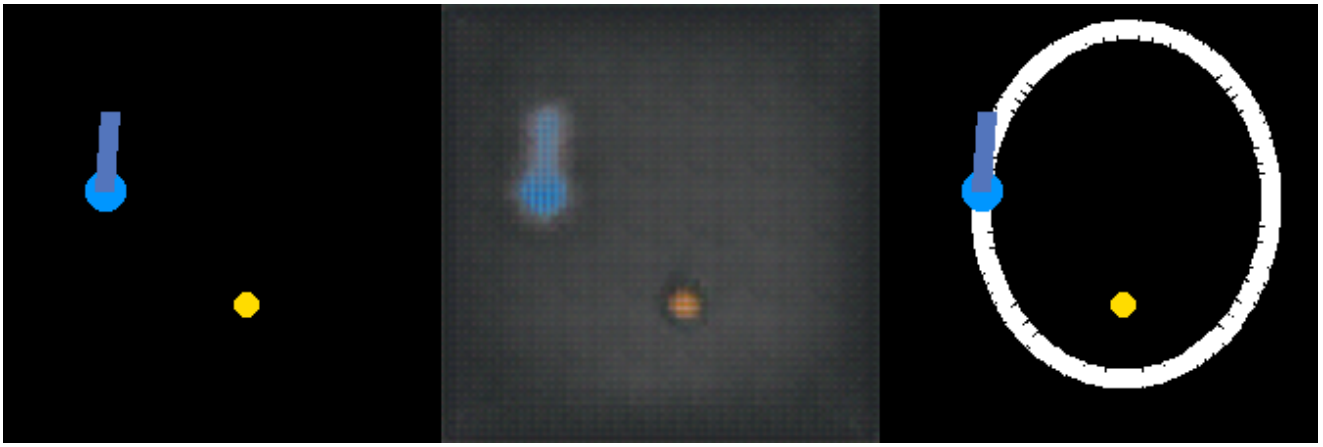
Task-Related Design Philosophy and Discussion:

1.This task is to further test the physics-related simulation capabilities of this paradigm, implemented through an image reasoning model.
2.Although there is a possibility of solving this problem through interpolation, the model and hyperparameter configuration used have also solved other problems that cannot be solved by interpolation, thus ruling out this possibility.
3.The fixed position of the star and the initial position of the planet are relatively easy to represent in the input image. For the initial speed magnitude and direction of the planet, I use the following method: a short ray segment is drawn from the planet's initial position, with its color representing the speed magnitude and its direction representing the velocity direction. For the correspondence between speed magnitude and color, please refer to the dataset generation script code.
4.Some task parameters:
Image Resolution: 224x224

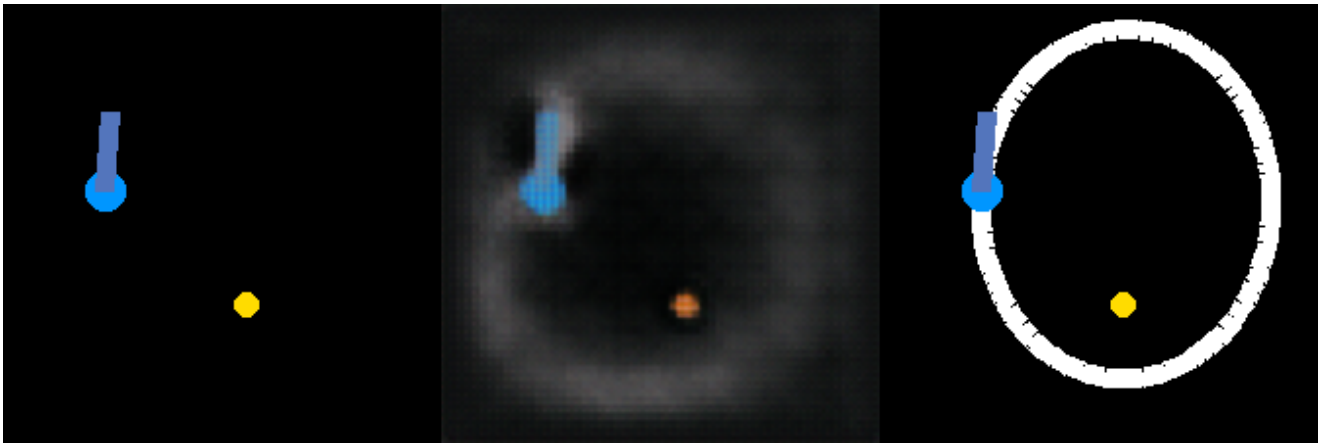
Experimental Results and Analysis: The final training results show that the model has completely learned the laws related to planetary orbits, but the precision of this simulation needs further investigation. The model provides a direct observation of the formation process of a machine mind. An interesting phenomenon emerged during the learning process of this task: since the orbit in the output image is set to white, during the learning process, especially in the early stages, this whiteness is diffused throughout the output image, reminiscent of a probability cloud in quantum mechanics.

Left is the input image, middle is the model-generated image, and right is the ground truth.

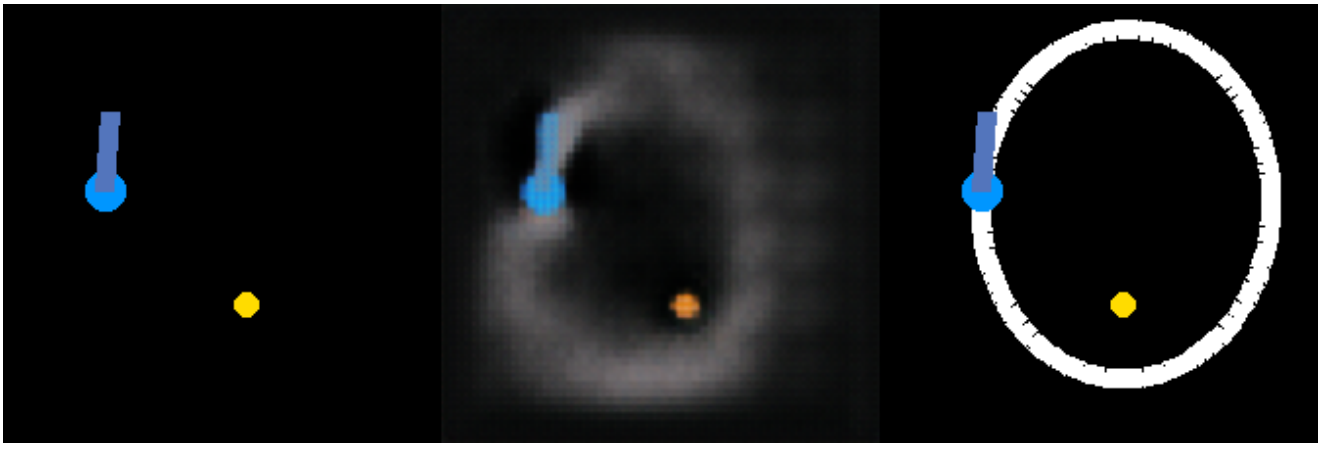
200 steps



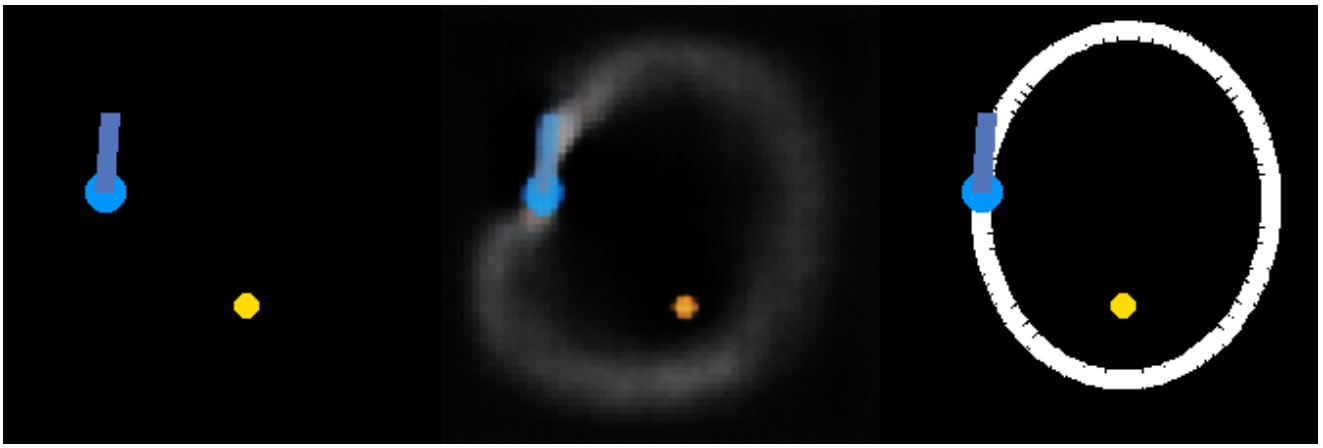
400 steps



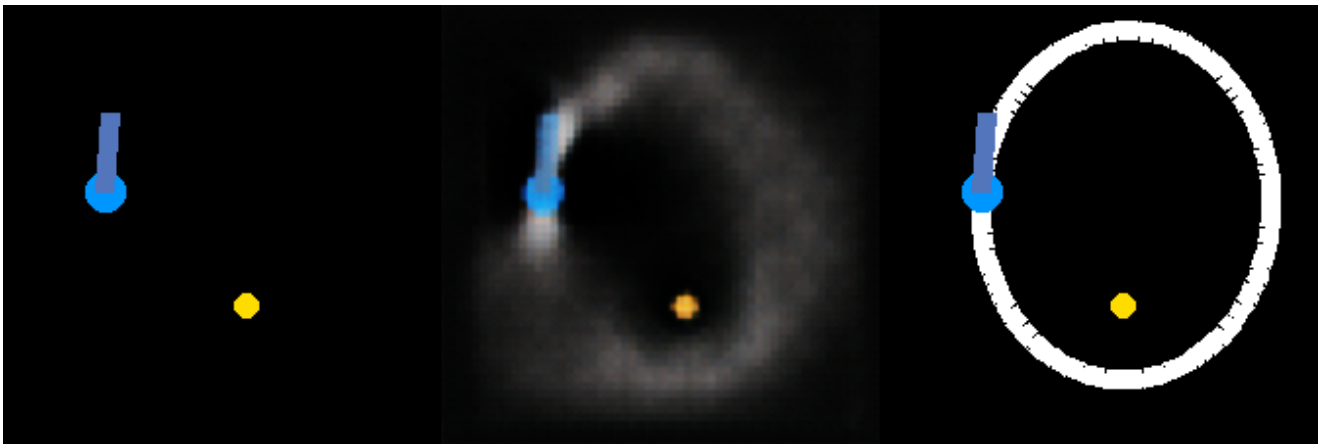
600 steps



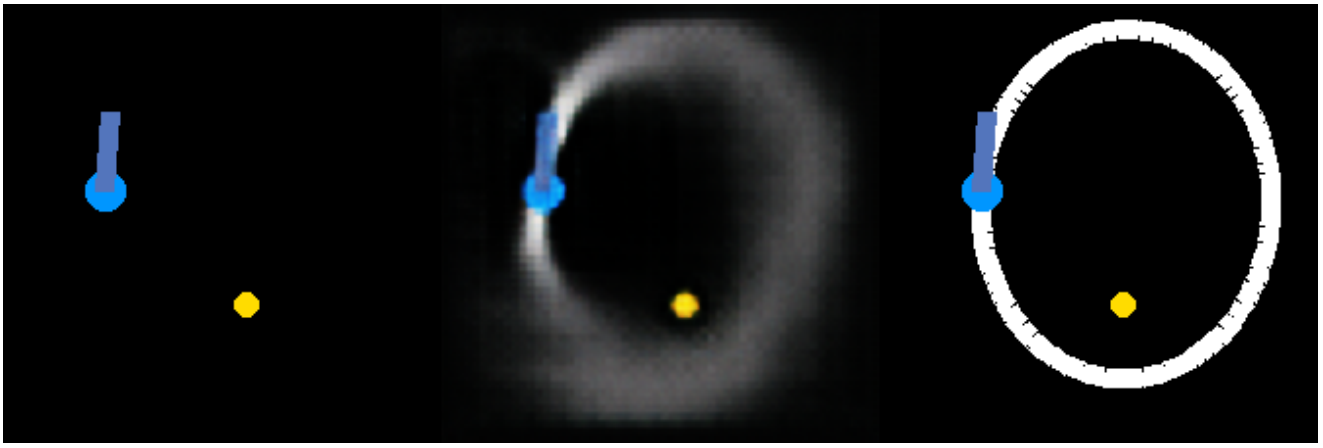
800 steps



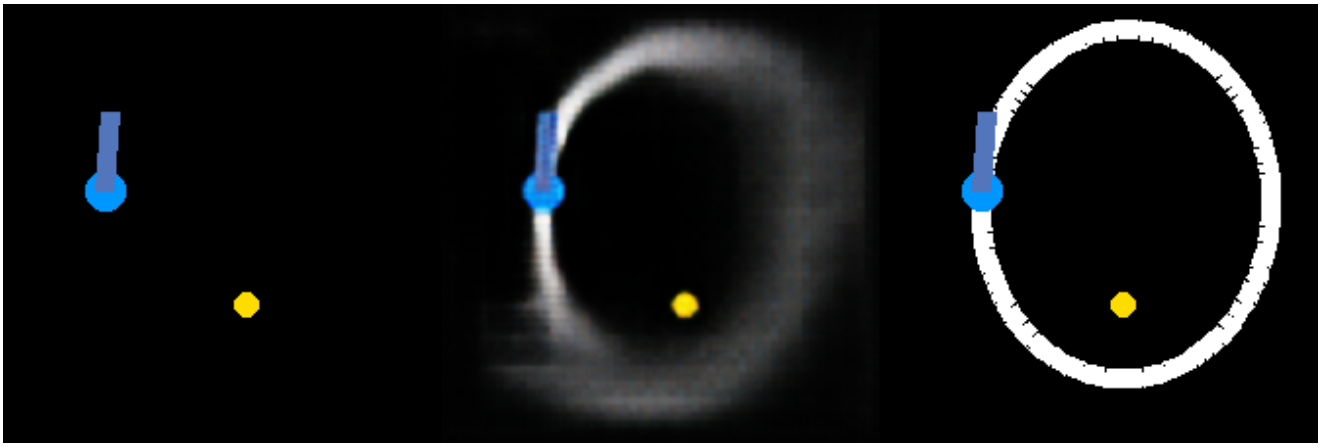
1000 steps



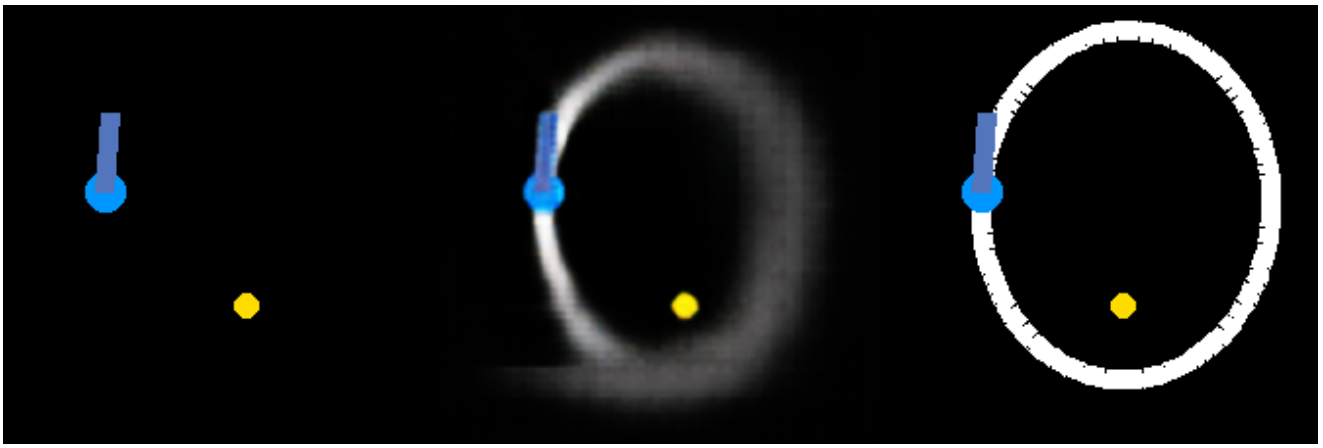
2000 steps



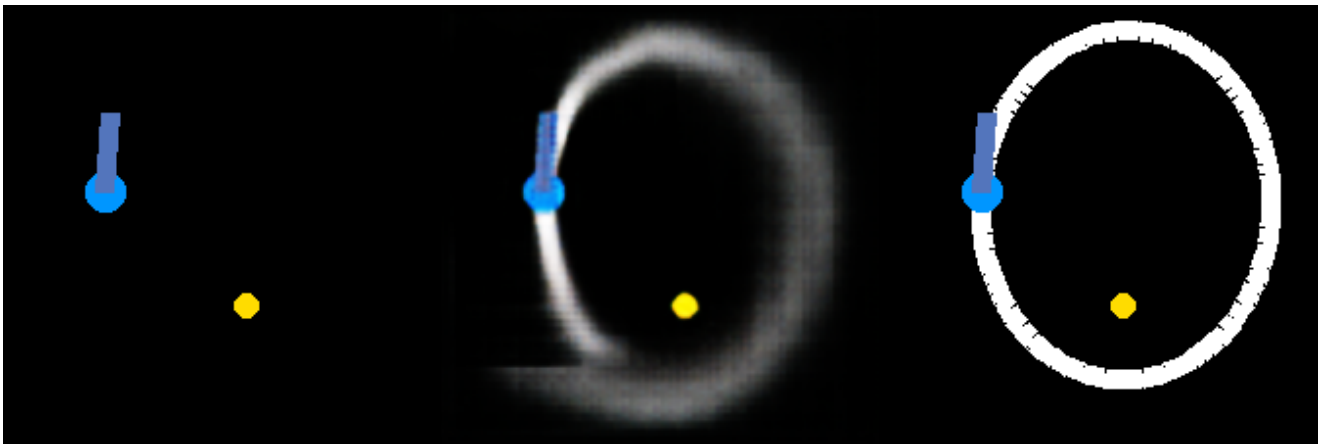
3000 steps



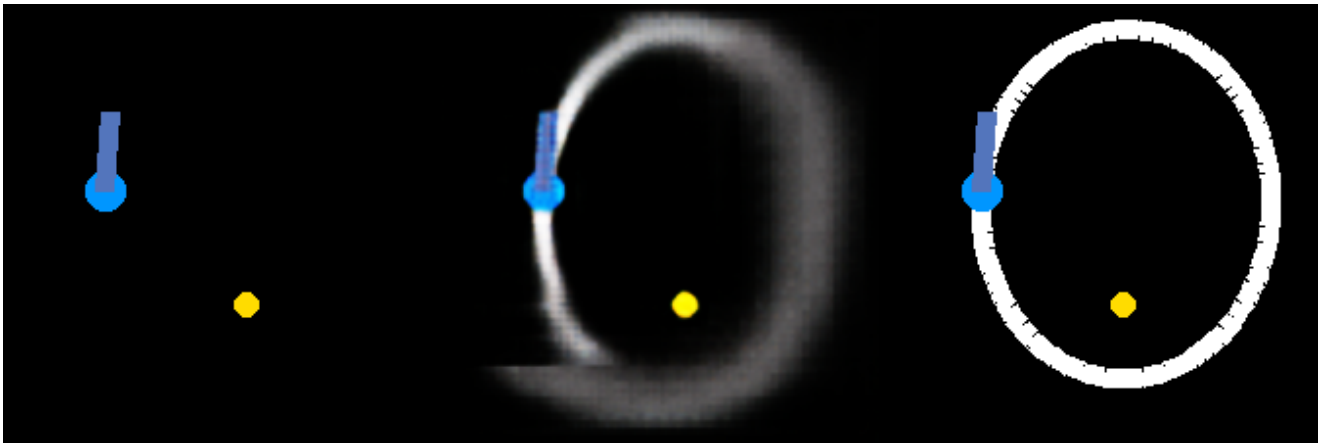
4000 steps



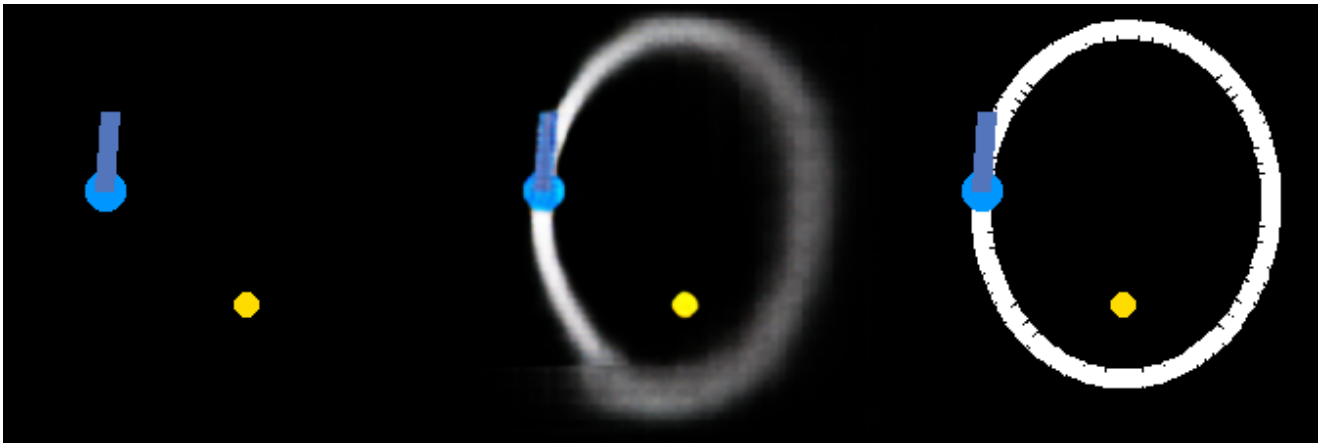
5000 steps



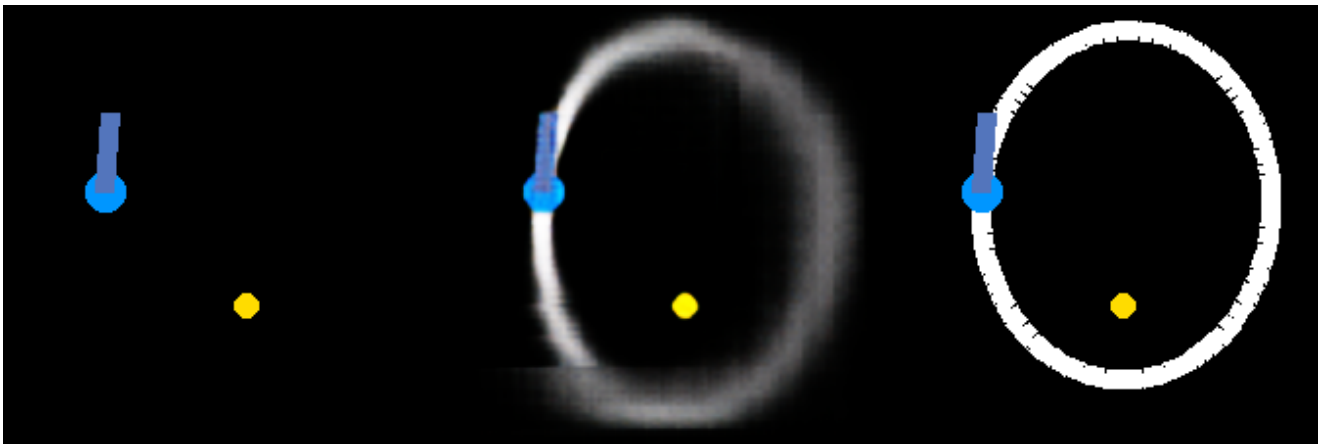
6000 steps



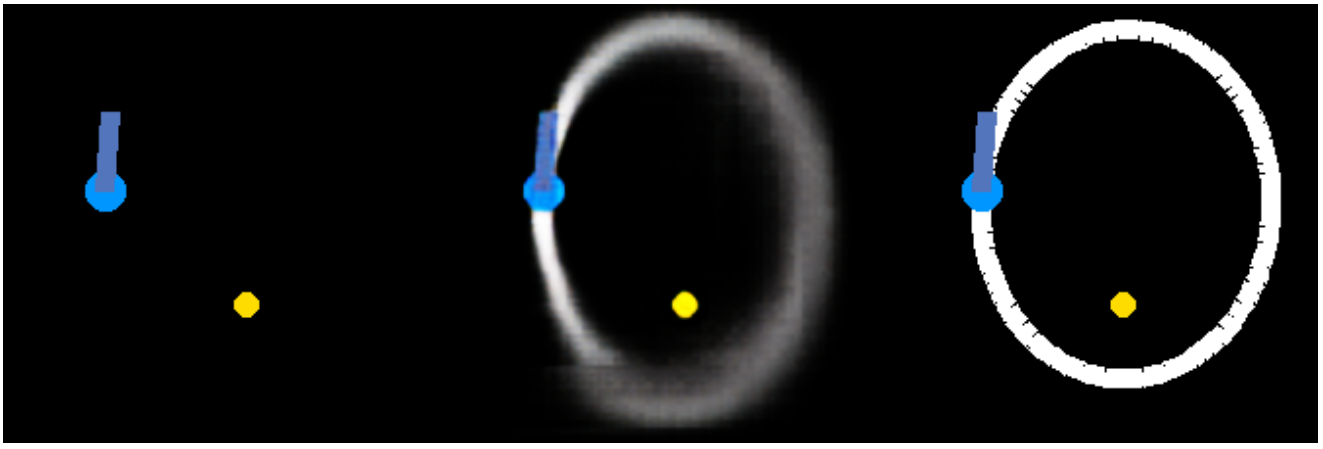
7000 steps



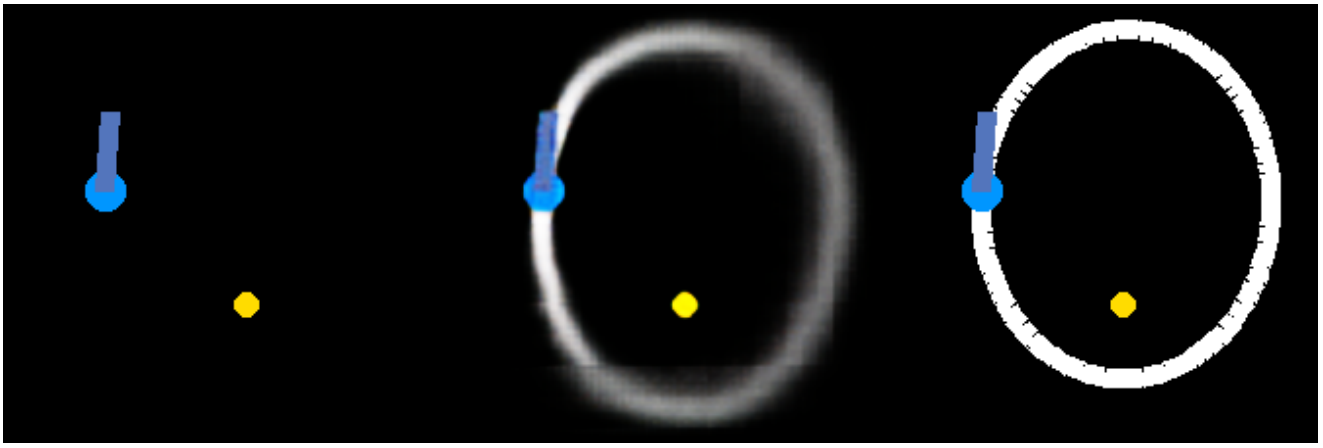
8000 steps



9000 steps

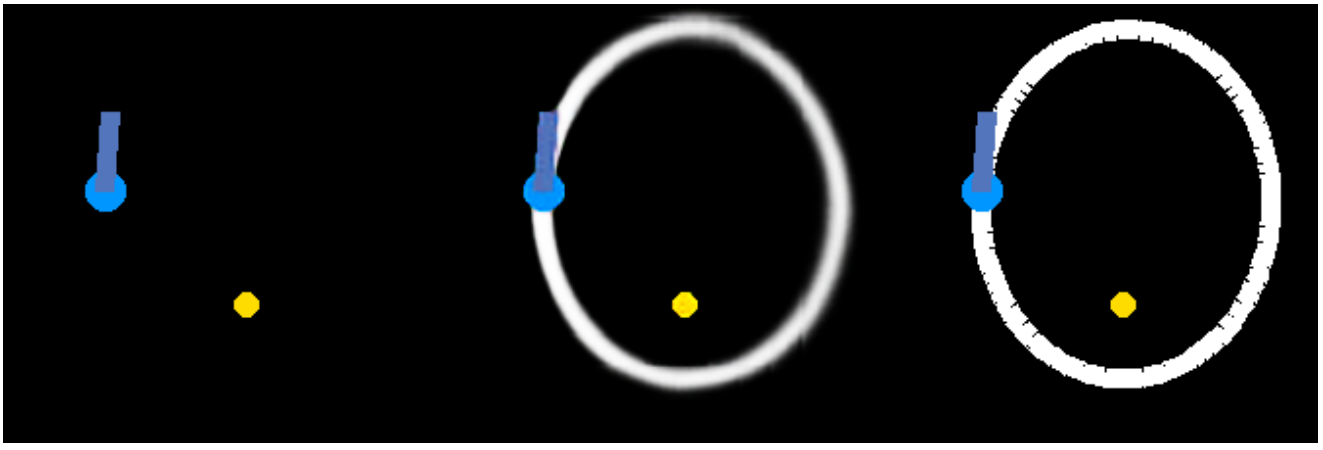


10000 steps



As the rate of change slows down, the final training result is shown directly.

156400 steps



4.7. ARC-AGI Related Tasks

This section will demonstrate the paradigm's ability to handle tasks from the [12,13] arc-agi-1/2 datasets.

Introduction to My Experimental Method:

1.The pattern in the arc-agi-1/2 datasets is to provide a very small number of examples, around 3, and require the model to grasp the game rules from these examples and generalize to solve new examples. I tried to use my method strictly following the arc-agi-1/2 requirements, but it was not feasible and resulted in severe overfitting. Therefore, I adopted a circuitous method. First, I manually observe the logic of a task. Through interactive programming with an LLM (like Gemini 2.5 Pro), I get assistance in writing and debugging the scripts to generate training data for these complex tasks. Then, I run the script to generate a large dataset, usually around 150,000-200,000 samples. Finally, I train on this dataset and seek generalization.
2.Because this experimental method requires a significant amount of time and effort, including manually observing the task logic and generating and modifying the dataset generation scripts, I only attempted it on 16 tasks: 8 from arc-agi-1 and 8 from arc-agi-2, selected randomly based on the principle that their training set generation scripts were relatively simple. The results were quite good. Most problems converged quickly, and a few required longer training but also converged. This universality is very shocking, considering I randomly selected 16 problems.
3.I used the image reasoning mode with the Swin Transformer + UNet model.
4.I adopted the colored image representation style from the official arc-agi-1/2 dataset website.
5.The horizontal and vertical dimensions of the grid were kept consistent throughout the dataset.
6.Generally, a dataset of 150,000 samples is sufficient for training, but due to time and energy constraints, I did not try to find the minimum number required.

4.7.1. Fluid Simulation

Task Description: From ARC Prize - Puzzle #1ae2feb7
Input Format: 224x224 image
Output Format: 224x224 image
Loss: MSELoss
Training Configuration: NVIDIA 4090 GPU
Dataset Generation Script: generate_arc_fluid_simulation.py
Dataset: arc_fluid_final_dataset folder
Training Code: train_image2image.py
Model Used: Swin Transformer + UNet
Training Set Size: 150,000
Input Space Size Estimation: Using a Monte Carlo method, 2,000,000 samples were run with no repetitions.
Training Set Size as a Percentage of Input Space: As above, it can be considered that the training set size is a very small fraction of the input space.

Task-Related Design Philosophy and Discussion:

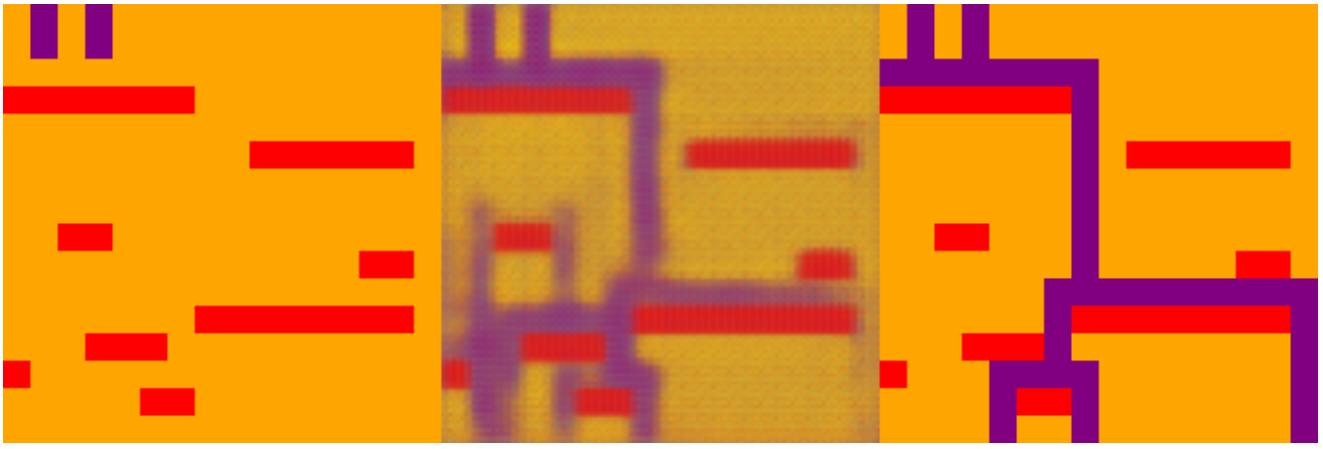
1.To further test the pure image reasoning capability of the Transformer, I extended my attempts to the existing arc-agi-1/2 datasets, but using a different, circuitous method as described above.
2.Some task parameters:
Image Resolution: 224x224

Experimental Results and Analysis: The final training results show that the model has completely learned the rules of this task. This indicates that the model possesses the reasoning ability expected by the ARC-AGI dataset, although it requires a large dataset for training. At the same time, the model's learning speed is very fast, providing a direct observation of the formation process of a machine mind. In this task, it can be seen that the model initially learned a simple rule; it seemed to guess that letting purple water flow from the edge of the red barrier was correct. In the subsequent process, it gradually abandoned this overly simple rule hypothesis and eventually learned the true rules of the task. This proves that my paradigm, combined with the "programmatic data generation" methodology, is a very promising path to tackling few-shot abstract reasoning problems. Moreover, the reasoning nature of the task itself largely rules out the possibility that the model is performing naive interpolation.

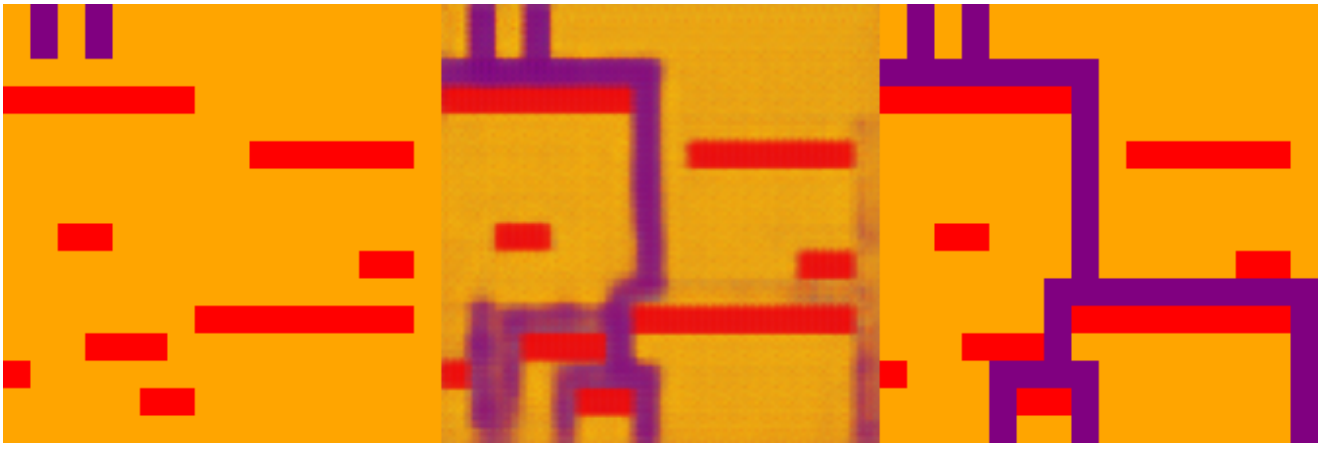
Experimental Process Showcase:

Left is the input image, right is the ground truth, and middle is the generated image.

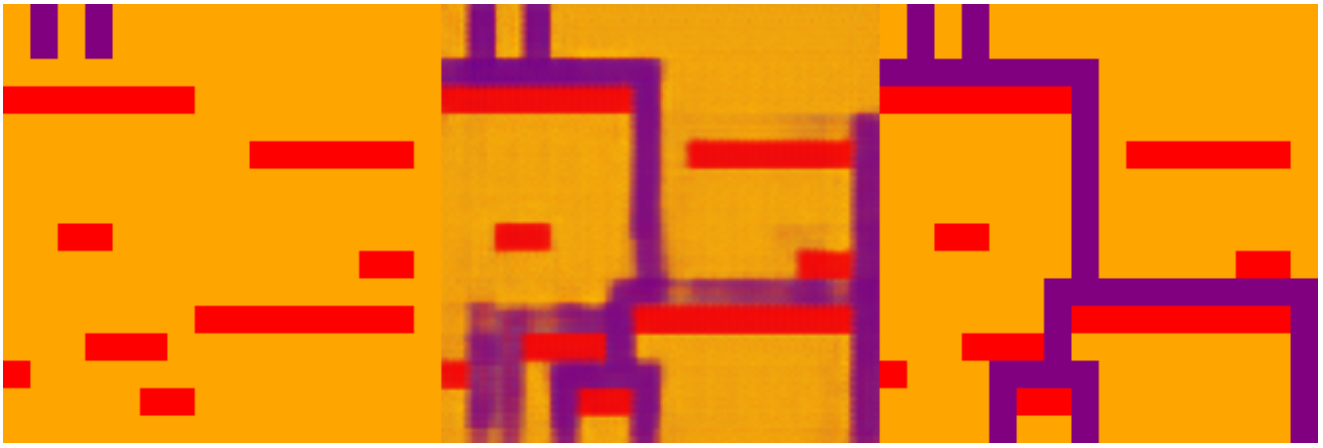
200 steps



400 steps

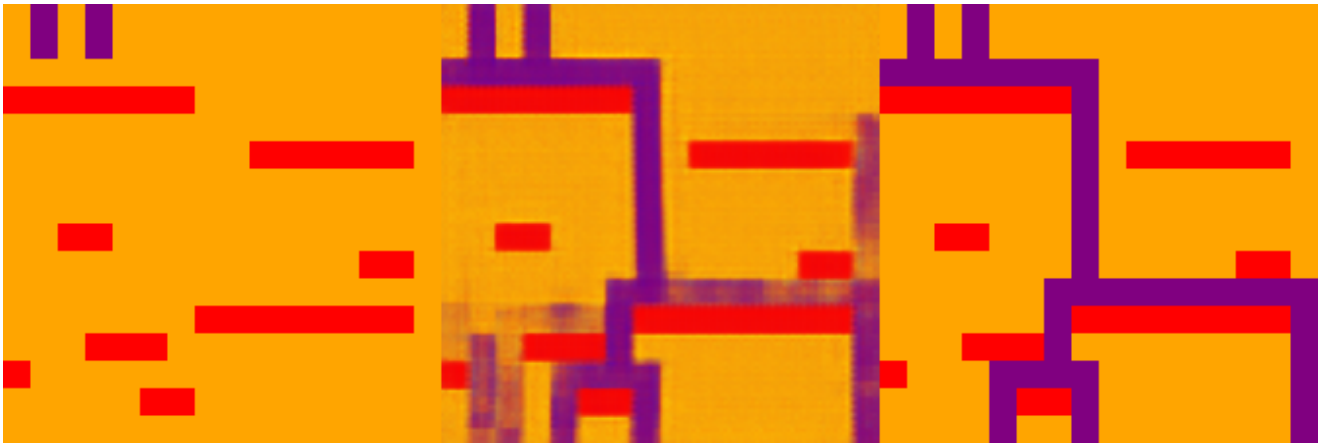


600 steps

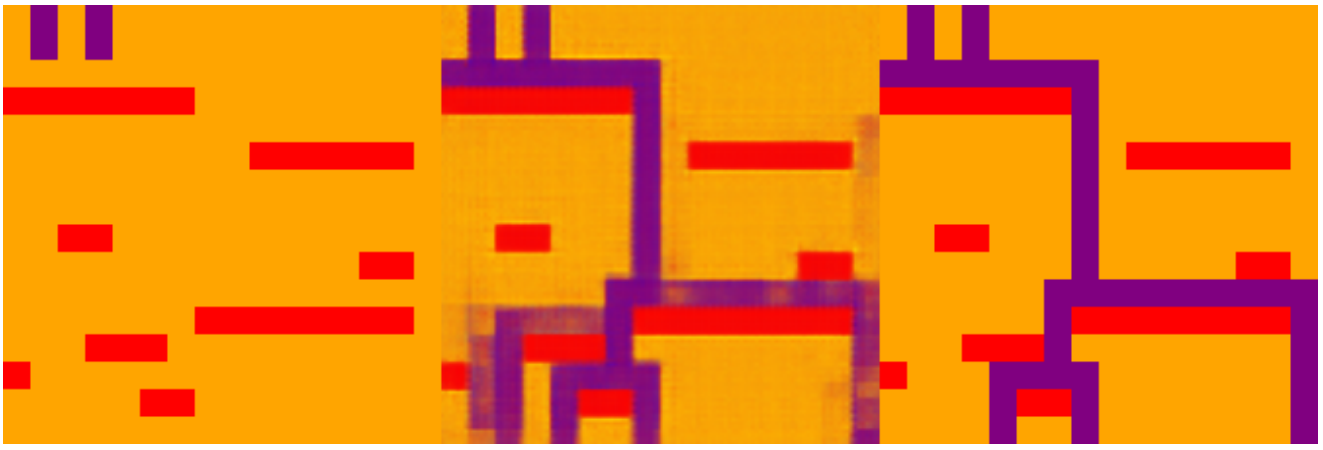


800 steps

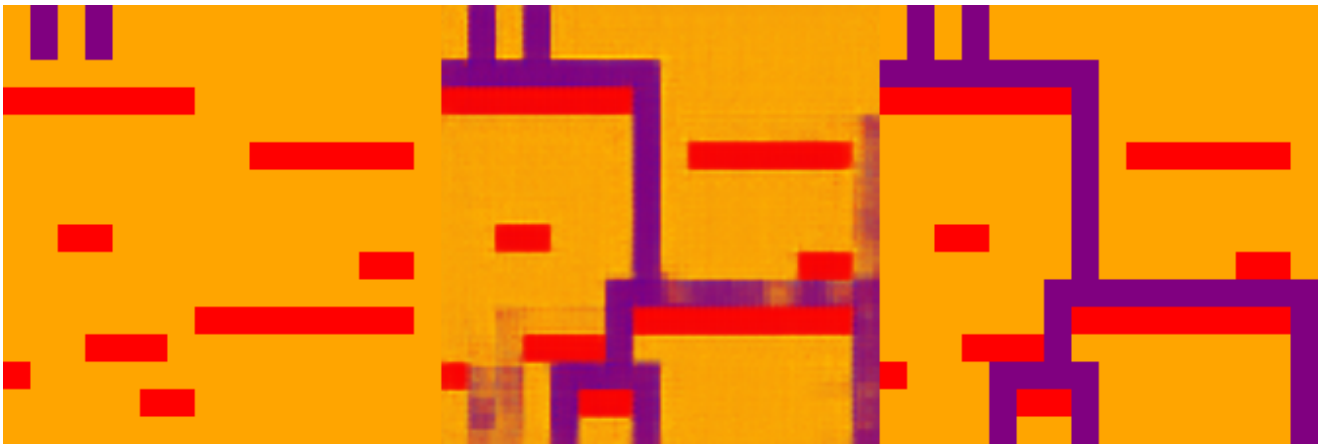
1000 steps



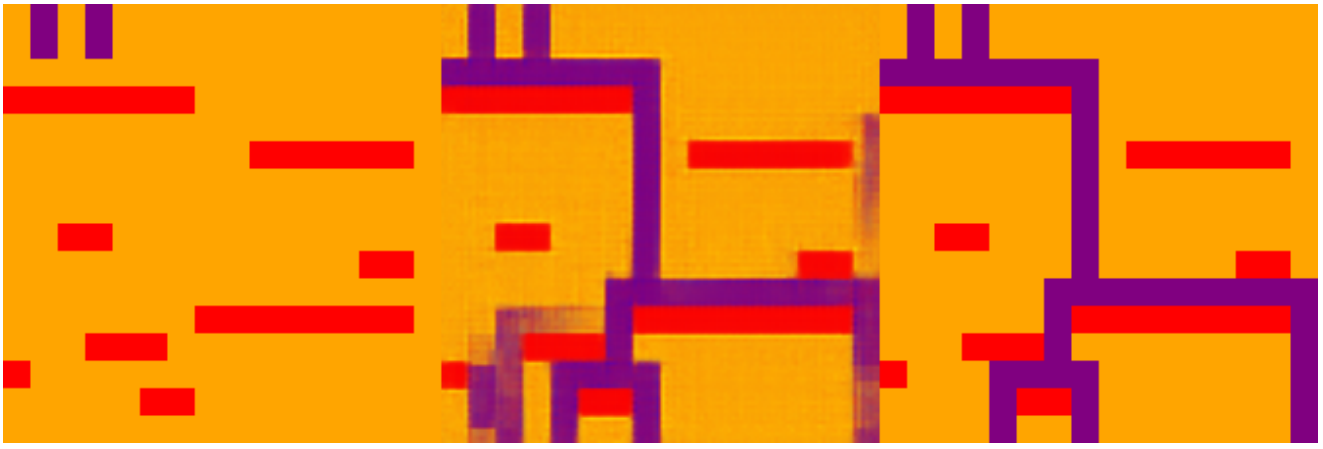
1200 steps



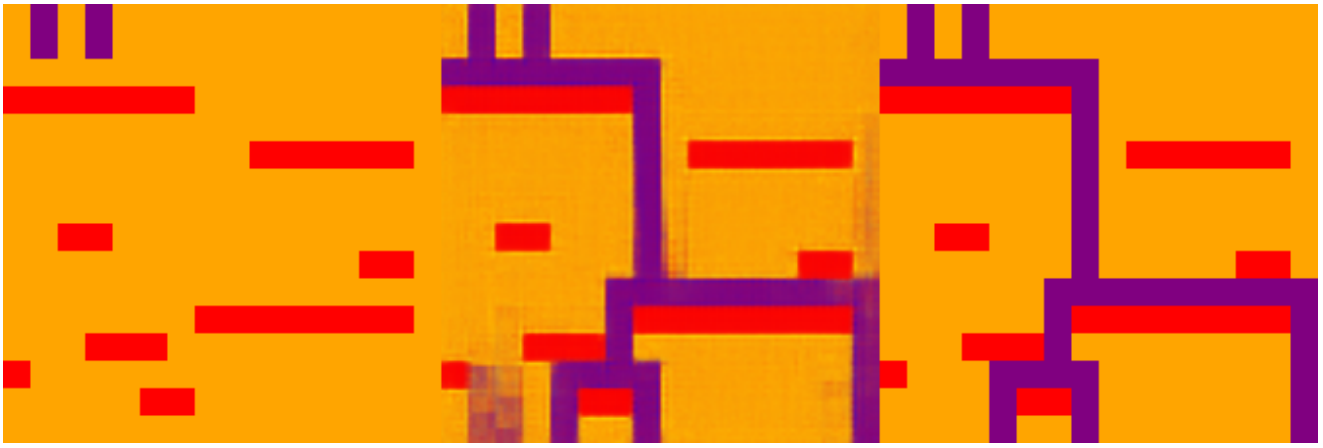
1400 steps



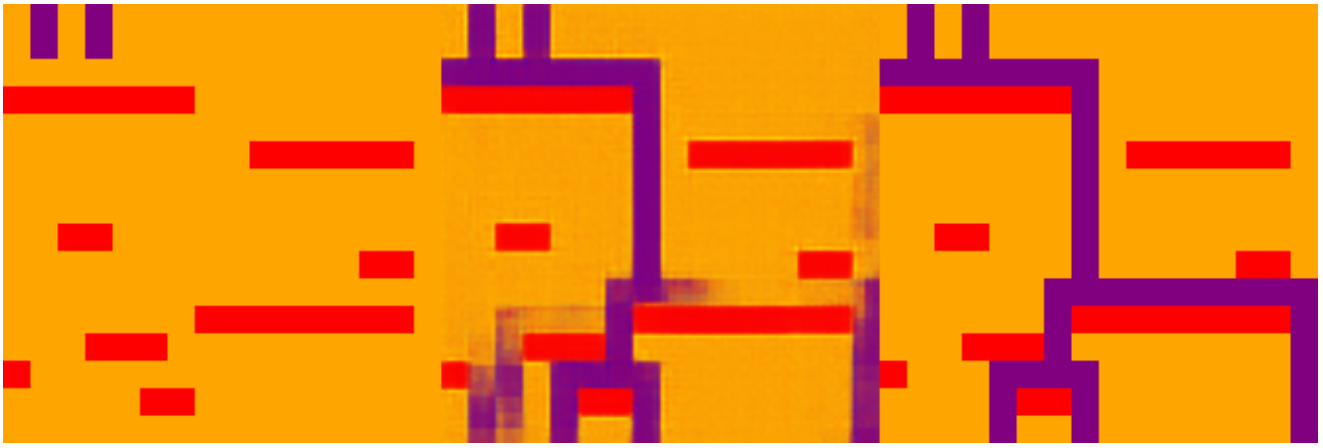
1600 steps



1800 steps



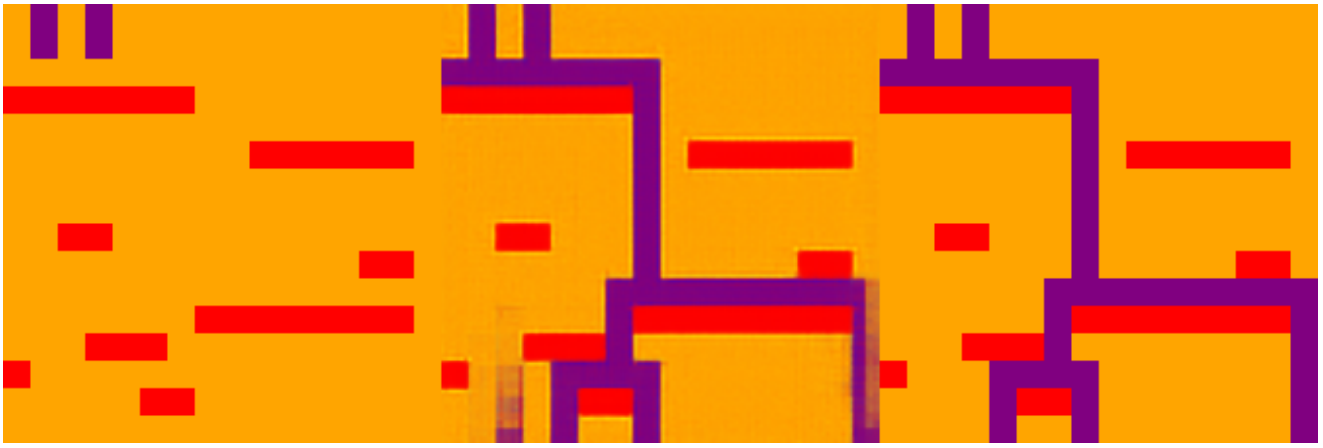
2000 steps



3000 steps

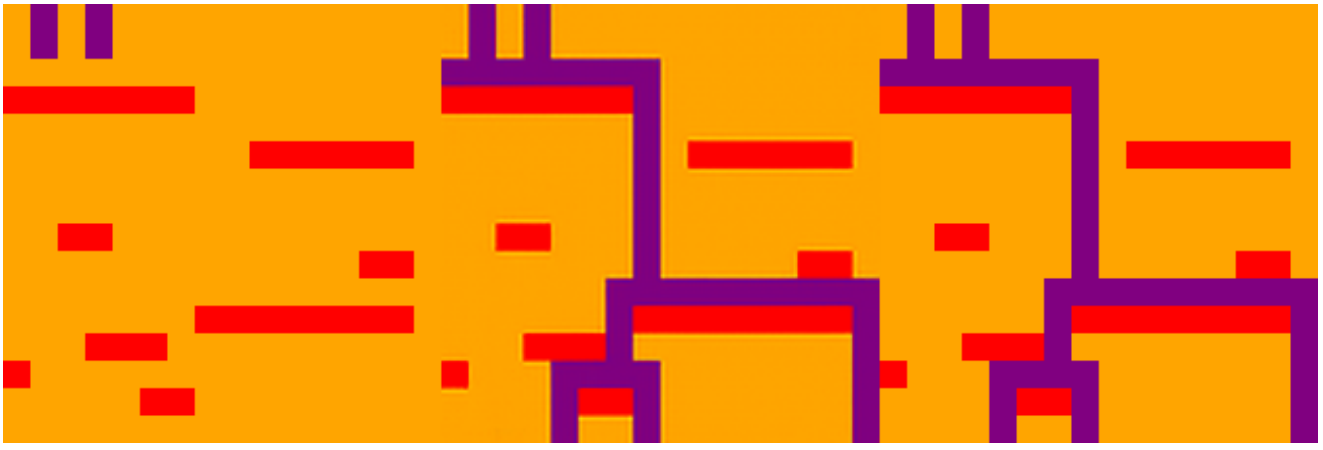
4000 steps

5000 steps



As the rate of change slows down, the final training result is shown directly.

39600 steps



4.7.2. Jigsaw Puzzle Game

Task Description: From ARC Prize - Puzzle #1ae2feb7
Input Format: 224x224 image
Output Format: 224x224 image
Loss: MSELoss
Training Configuration: NVIDIA 4090 GPU
Dataset Generation Script: generate_arc_jigsaw_puzzle_advanced.py
Dataset: arc_jigsaw_puzzle_mine_dataset folder
Training Code: train_image2image.py
Model Used: Swin Transformer + UNet
Training Set Size: 200,000
Input Space Size Estimation: Using a Monte Carlo method, 1,000,000 samples were run with no repetitions.
Training Set Size as a Percentage of Input Space: As above, it can be considered that the training set size is a very small fraction of the input space.

Task-Related Design Philosophy and Discussion:

1.To further test the pure image reasoning capability of the Transformer, I extended my attempts to the existing arc-agi-1/2 datasets, but using a different, circuitous method as described above.
2.Some task parameters:
Image Resolution: 224x224

Experimental Results and Analysis: The final training results show that the model has completely learned the rules of this task. This indicates that the model possesses the reasoning ability expected by the ARC-AGI dataset, although it requires a large dataset for training. At the same time, the model required a longer time to learn this task, providing a direct observation of the formation process of a machine mind. In this task, it can be seen that the model initially learned the background color, possibly because it contributed a large portion of the loss at the beginning and was relatively easy to learn. Then, it slowly learned the shape and position of the puzzle pieces to be filled, and finally spent a considerable amount of time learning the rule for filling the colors. This proves that my paradigm, combined with the "programmatic data generation" methodology, is a very promising path to tackling few-shot abstract reasoning problems. Moreover, the reasoning nature of the task itself largely rules out the possibility that the model is performing naive interpolation.

Experimental Process Showcase:

Left is the input image, right is the ground truth, and middle is the generated image.

10000 steps

20000 steps

30000 steps

40000 steps

50000 steps

60000 steps

70000 steps

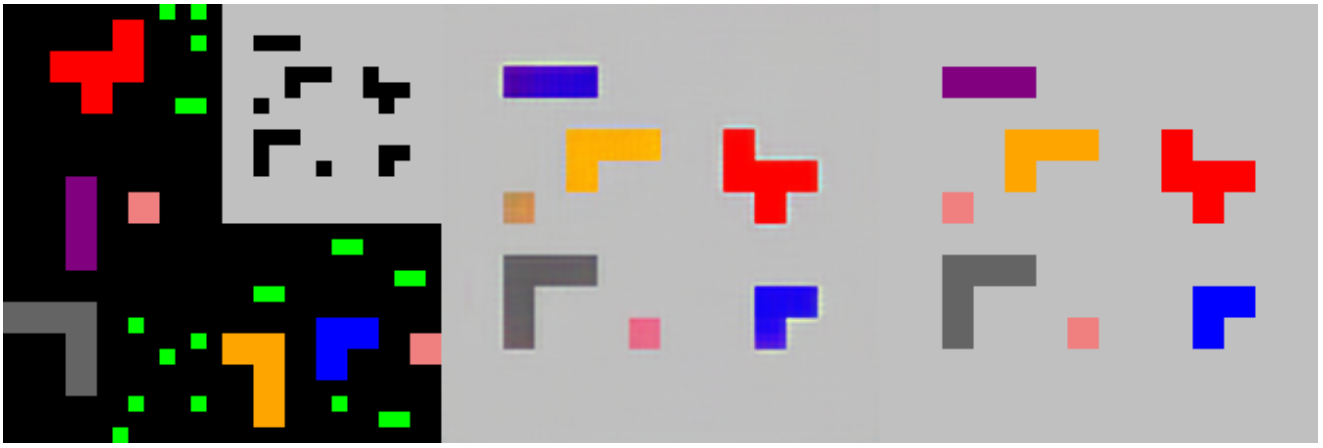
80000 steps

90000 steps

100000 steps

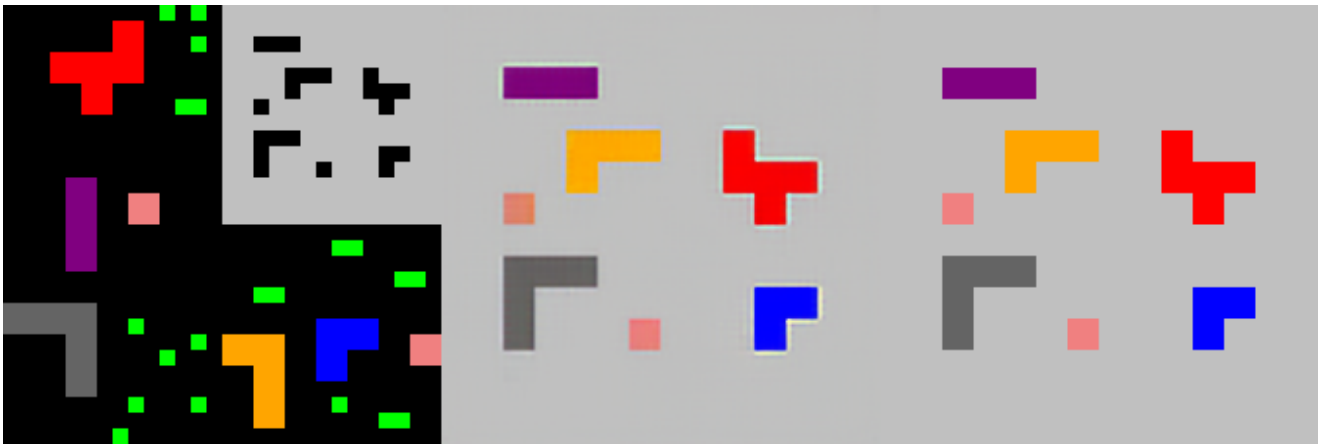
150000 steps

200000 steps



250000 steps

300000 steps



As the rate of change slows down, the final training result is shown directly.

530400 steps

4.8. Interpretability Exploration

This section will demonstrate the paradigm's ability to output the reasoning process.

4.8.1. Outputting the Editing Process

Task Description: The model learns to predict and output the shortest editing process between two 0/1 strings.
Input Format: A 30-character string of "0"s or "1"s, where the first 15 characters represent string A and the last 15 characters represent string B.
Output Format: Multi-label binary classification format with 450 labels, equivalent to 450 0/1 characters. These are grouped into sets of 30 characters. In each group, the first 15

characters represent an intermediate state of the string during the editing process, and the last 15 characters represent a mask to handle cases where the string length is less than 15 during the editing process.
Loss: Mean Binary Cross-Entropy Loss
Training Configuration: NVIDIA 4090 GPU
Dataset Generation Script: generate_edit_distance_explainable.py
Dataset: edit_distance_path_unique_final.jsonl
Training Code: train_tiny_transformer.py
Training Set Size: 500,000
Input Space Size Estimation: 2^30 = 1,073,741,824
Training Set Size as a Percentage of Input Space: 0.0466%

Task-Related Design Philosophy and Discussion:

1.In theory, this is not a direct explanation of neuron weights or the function of a specific module, but rather a form of interpretability guided by explainable labels. The method for this kind of interpretability is to directly add the internal processes that need to be observed to the labels. Practice shows that this method is practical. For example, for some problems, since the network can fit the answer, in many cases, I can be confident that it must also understand some internal states or intermediate processes in some way. At this point, this information can be exposed through this method. As for the purer, original meaning of neural network interpretability, I believe this paradigm can provide a cleaner laboratory environment to observe changes in neural network weights, which will be discussed later.
2.Some task details:

- The script ensures that the string length does not exceed 15 during the editing process, otherwise my output format cannot represent it.
- The script also ensures that there is one and only one minimum editing path, otherwise it would greatly increase the difficulty of network convergence.
- The edit distance strings are limited to 0s and 1s to reduce the task difficulty without changing the fundamental nature of the problem.

Experimental Results and Analysis: The evaluation loss converged to 0.001404. This success shows that the paradigm can not only provide the answer but also has the potential to output the steps to solve the problem, providing a new, operational approach to solving the black-box problem of neural networks.

Final Convergence Result:

--- Step 66000 ---
Train Loss: 0.001995
Eval Loss: 0.001404

Act V: On the Boundaries of Learnability and Interpretability

I summarize the characteristics of this paradigm as follows:

1.The input and output formats must be well-designed. The input format should minimize the decoding burden on the Transformer. The output should preferably use multi-class classification with cross-entropy loss or multi-label binary classification with mean binary cross-entropy loss. You can design the meaning of the output format according to the characteristics of the task, as long as the format is consistent throughout the dataset. For example, if the output is a number, you can use multi-label binary classification to represent a binary number of a corresponding number of bits.

2.Regarding learning difficulty.

Previously, when working on symbolic rule learning tasks, I observed a phenomenon where the structure of the task is also important. It seems that forward symbolic rule tasks are easy to learn, while solving for the input from the output is not, unless this inverse problem is also in the form of some forward symbolic rule. Now it seems that this "inverse solving" task is more like an algorithm simulation problem, hence the greater learning difficulty.

Another phenomenon observed is that tasks with conditional branches can exacerbate learning difficulty, for example, if an indicator bit instructs the use of a completely different rule to calculate subsequent operands. In retrospect, conditional branching is also part of the rule itself, but it increases the complexity of the symbolic rule, making it harder to learn.

Additionally, the input can reserve positions to control the task's rules, i.e., input = task rule + operands. This idea has been successfully validated in experiments related to cellular automata.

Another observed phenomenon is that tasks with too many simple logics mixed in series are also difficult to learn. If a symbolic transformation rule is a serial combination of different simple symbolic rules, even if each rule itself is very simple to learn, it can still combine into an extremely difficult task to learn. A simple example is the trapping rain water experiment in the experimental section. By decoupling the problem and removing the final step of summing the water amounts, the task was able to converge quickly.

Forward symbolic rule learning has varying degrees of difficulty. For example, for a 1D 30-bit cellular automaton, the learning difficulty varies for different rules and for applying the rule for different numbers of layers. Basically, the more layers, the harder it is to learn. There are also some tasks that are fundamentally unlearnable, such as RSA encryption. Although it is a forward, simple mathematical calculation, it is indeed unlearnable.

So, the difficulty of the rule affects the speed of loss decrease. To give a specific example, for a cellular automaton with rule 110, the number of evolution layers, at least for a small number of layers, strictly follows a pattern where more layers mean more difficult training and slower model convergence. This is reminiscent of what Wolfram called [16] computational irreducibility, and he indeed used the evolution of cellular automata as an example.

Furthermore, the training process itself is a form of computational irreducibility; the process of training/sculpting cannot be skipped. And simpler problems train faster, while harder problems train slower. If we disregard whether the nature of the problem is friendly to neural networks or whether the input/output formats are friendly, and if we control the scale of the input for different problems to be roughly the same, the speed of loss decrease during training could even be used to estimate the learnability or computational irreducibility of the problem. For example, when I tried to train RSA encryption, the loss did not decrease at all; it never learned anything. Here again, taking the cellular automaton as an example, both input and output are 30-bit 0/1 strings, so the type of rule and the number of evolution layers completely determine the mapping from input to output. But this is just one type of mapping from the input space to the output space. Why is this type of mapping easier to learn than—a cellular automaton mapping rule with more evolution layers? And why is it obviously easier to learn than a completely random permutation mapping?—A completely random permutation mapping is obviously unlearnable because it contains no patterns. This is a very interesting question, but its exploration is beyond my capabilities. Additionally, the unlearnability of the RSA algorithm confirms that the simple algorithm created by cryptographers is very successful in this sense.

So, through this, it might be possible to use the method of actually training a task, a kind of Monte Carlo method, to measure task difficulty, providing a quantitative measure of learnability/computational irreducibility. This could have potential in mathematics. Why are some rules considered complex and others simple? And will this form a new theory?

Regarding model capacity, my intuition is that a larger model capacity or a model better suited for the task would lead to faster training. Therefore, it is likely the combination of model capacity and task difficulty that determines the convergence speed and process of the training loss.

3.Regarding what happens when the transformation rules in the training set are inconsistent. Even with simple rules, if different data in the training set apply different rules, it is very difficult to learn, because there is no indication of which rule to apply. The model can only try to strike some balance based on the principle of reducing the loss. If many rules are mixed, but it's impossible to identify which simple rule to use, the training loss and eval loss will oscillate at a high point because the different rules are pulling towards an equilibrium point. The level of this loss will depend on the purity of the rules in the training set.

Additionally, mod 3 is a surprisingly unlearnable rule. It is very simple, yet the loss remains at a high level. It might be related to the phenomenon of mixed rules, but at least from a human perspective, it does not intuitively seem like a mixed rule. I conducted an experiment where the input is a 20-bit binary number, represented by a 20-character 0/1 string, and the task is to calculate its mod 3 result, represented by a 2-bit binary number. The loss is mean BCE loss. Now for the interesting results: I generated four datasets. The first had no restrictions on the input. The second restricted the input so that mod 3 could not be 0. The third restricted it so mod 3 could not be 1, and the fourth restricted it so mod 3 could not be 2. The training results showed that their eval losses stabilized at the following values:

No restriction dataset — 0.636
Restriction: mod 3 not 0 — 0.693
Restriction: mod 3 not 1 — 0.347
Restriction: mod 3 not 2 — 0.347

Analyzing these results is very interesting. I'll first state a fact: if a model learns nothing from the data, for a BCE loss, if the probability of the label being 1 is p, the model will reduce the loss to

$$-p\ln p-(1-p)\ln(1-p)$$

. I will then show that these loss results prove the model learned nothing from the data, i.e., mod 3 is unlearnable. The reasons are as follows:

For the dataset with no restrictions, for the 2-bit binary output, the probability of the high bit being 1 is 1/3, and the probability of the low bit being 1 is also 1/3. Therefore, the optimal

$$\text{loss}=-\frac{1}{3}\ln\left(\frac{1}{3}\right)-\frac{2}{3}\ln\left(\frac{2}{3}\right)=0.6365,$$

which is very close to the experimental result.
For the dataset where mod 3 is not 0, for the 2-bit binary output, the probability of the high bit being 1 is 1/2, and the probability of the low bit being 1 is also 1/2. Therefore, the optimal

$$\text{loss}=-\frac{1}{2}\ln\left(\frac{1}{2}\right)-\frac{1}{2}\ln\left(\frac{1}{2}\right)=0.693,$$

which is very close to the experimental result.
For the dataset where mod 3 is not 1, for the 2-bit binary output, the probability of the high bit being 1 is 1/2, and the probability of the low bit being 1 is 0. Therefore, the optimal

$$\text{loss}=-\frac{1}{2}\ln\left(\frac{1}{2}\right)=0.347,$$

which is very close to the experimental result.
For the dataset where mod 3 is not 2, for the 2-bit binary output, the probability of the high bit being 1 is 0, and the probability of the low bit being 1 is 1/2. Therefore, the optimal

$$\text{loss}=-\frac{1}{2}\ln\left(\frac{1}{2}\right)=0.347,$$

which is very close to the experimental result.

Therefore, it can be concluded that, at least in this experimental setup, mod 3 is unlearnable, and not because of insufficient fitting capacity.

A similar phenomenon was observed in the Tower of Hanoi experiment. Specifically, if I only train on the optimal path from the initial state to the final state for a fixed number of disks, training the optimal action for each of these states, convergence is very rapid. But when I try to expand the state space to the entire Tower of Hanoi state space for a fixed number of

disks, the loss gets stuck at a high level and does not decrease further, and it's likely not an issue of fitting capacity. Note the similarity between the entire state space of the Tower of Hanoi and the mod 3 problem. In binary format, mod 3 has a somewhat recursive or fractal nature, and the Tower of Hanoi is inherently recursive or fractal. So, is this because the input or output format design makes it unlearnable, or is it the problem itself? Is it that this paradigm cannot effectively recognize rules involving recursion or fractals? I lean towards the latter. So, why does training only on the optimal path from the initial to the final state of the Tower of Hanoi lead to quick convergence? The answer should be that, on this optimal path, there is a non-recursive recognition rule.

So, are these two problems related to recursion or fractals also related to the mixed rules mentioned at the beginning? This paper believes they might be related, because when the model cannot recognize a recursive rule, it might interpret it as some complex form of mixed rules, and it cannot identify which input corresponds to which rule.

4.In this paradigm, data can often be generated infinitely and cheaply by programs. In theory, I need the training set to have the same distribution as the entire input space, and the simplest way to achieve this is through random sampling in the input space. But this raises another question: could there be another way to construct the training set that would lead to better model learning? For example, if a certain branch of the rule occurs infrequently, random sampling might not cover enough patterns. My idea is that there might be more sophisticated ways to generate datasets that better cover the various patterns of the problem. Because, based on practical experience, as long as there is sufficient pattern coverage, the training set can be effective, having resistance to overfitting and meeting the model's learning needs. This is also why the total size of the training set does not need to be very large. From this perspective, the reason why the previous pattern recognition deep learning paradigm required large training sets becomes clearer: those tasks did not have precise transformation rules, so more data was needed to cover the various patterns of the input space.

This paradigm does not pursue OOD generalization. In other words, since the training data occupies a very small portion of the entire input space, the model, after convergence, can generalize precisely to the entire vast remaining input space. This is analogous to generalization in past paradigms.

5.Controllable AI and Interpretability

Regarding interpretability, as explained above, this is theoretically not a direct explanation of neuron weights or the function of a specific module, but rather a form of interpretability guided by explainable labels. The method for this kind of interpretability is to directly add the internal processes that need to be observed to the labels. Practice has shown that this method is practical. For example, for some problems, since the network can fit the answer, in many cases, I can be confident that it must also understand some internal states or intermediate processes in some way. At this point, this information can be exposed through this method.

Regarding more controllable AI, due to the nature of this paradigm itself—a non-autoregressive Transformer model fitting precise rules—it should theoretically not exhibit hallucinations like LLMs or adversarial examples in CV. In practice, no similar hallucinations have been found. The reason may be that, since this paradigm is based on rule-based tasks with extremely precise internal rules, it is inherently non-smooth and cannot be solved by interpolation. Therefore, the model can only learn the rules, making it naturally resistant to hallucinations.

Another related finding is that the batch size during training should not be too small. If the batch size is too small, learning will be much slower. It is speculated that this is because the tasks in this paradigm are based on precise rules, and the gradient directions of each sample will vary more. Therefore, a sufficiently large batch size is needed to average the gradients.

6.A phenomenon observed during training is that the training loss and validation loss decrease synchronously, and the validation loss is almost always lower than the training loss. Moreover, this paradigm is very resistant to overfitting, unless the input space is particularly large and the problem is particularly complex, such that the training set size does not meet the requirement of covering all patterns of the problem.

7.This paradigm learns symbolic rules. The distance it sees between 0 and 1 is the same as the distance between 0 and some emoji; they are all placeholders for symbols. "Structure" is the real "language": what it ultimately learns is not that 0 represents 0 or 1 represents 1. It learns something more abstract—regardless of what these symbols look like, they correspond to a mathematical structure with specific group and ring theory properties that can be used for addition. It communicates with this abstract structure, not with the specific symbols. This has already been confirmed in the previous N-ary addition experiment.

8.On the overall philosophical meaning of this paradigm.

- Data must cover the various patterns of the problem and is required to have the same distribution as the entire input space (whether it must be identically distributed is debatable, but at least this is the most convenient approach). The absolute quantity is not the requirement, although more data is certainly better. Data is like samples or models for the neural network to create a reference from. Why is more data better? It's similar to the idea of a Monte Carlo algorithm: you don't need explicit logical computation, just data and training. The more data, the clearer and more specific the reference samples or models provided to the neural network. The more training, the better the neural network fits. From a data-centric programming perspective, the scale and quality of the dataset are key to the model's performance.
- An overall analogy for the paradigm is that a neural network is like a sculptor carving a block of marble to match the input-output patterns of the data. The data is the model, and the neural network is the sculptor. Using computational power/training is like carving the marble, using continuity to approximate a deterministic symbolic transformation rule or to fit an unknown but deterministic algorithm. Insufficient training, where the loss has not

yet dropped to its minimum, is like an unfinished sculpture. If the problem is too difficult and exceeds the capacity of the neural network, it's as if the sculpture requires skills beyond the sculptor's ability. And even with insufficient training or a problem that is too difficult, the neural network still performs a fit and effectively compresses the uncertainty of the output space, just as an unfinished sculpture is more like the model than the original block of marble.

And I believe the most interesting aspect of this paradigm is that it can continuously approximate a deterministic symbolic rule or algorithm. I think this is even an interesting process mathematically. Moreover, I believe my paradigm has the potential to compress the uncertainty of a problem's output in cases where people have not yet found a general solution, although it may not be able to provide a precise algorithmic process. It is learning a symbolic or algorithmic task with a neural network, but in its own unique way, definitely not in the step-by-step, self-evident steps that I, as a human, would assume, but through coupled loss gradient descent. It is just trying to reduce the loss. For example, in an algorithm problem, if you wanted to explain which part of the already trained weights is executing which step of the algorithm, I believe it would be impossible. It doesn't know the position and meaning of the input symbols; it is just doing gradient descent.

I now feel that this paradigm is consistent with what Demis Hassabis advocates for compressing structural intuition and what Ilya Sutskever insists on, that compression is intelligence. [1] AlexNet proved that previously handcrafted features were not good enough, and learning image features directly with connectionism yields better results. And this paradigm proves that, even for precise symbolic rules, this route is correct to some extent. And, as mentioned before, even on tasks that do not fully converge, the model can often compress the uncertainty of the answer.

As for why this connectionist compression attempt might be effective, my feeling is that what Demis Hassabis mentioned—the complexity of the world is exceeding the carrying capacity of human language and logic—might be a limited explanation. I am reminded of a line from Bob Dylan's song "Things Have Changed": "All the truth in the world adds up to one big lie." But this does not mean the failure of logic. My view is that each proposition in a long logical chain has its own unique context, and often when people combine them into a logical chain, they tend to ignore these contexts to some extent, leading to a distortion of the final logical chain. This is often seen in real life: a person explains their thinking, and although every sentence they say seems correct and irrefutable, it ultimately leads to an absurd conclusion. This is where connectionism comes in, directly interfacing with real data and empirical verification to recalibrate one's own intuition and logic. It's like symbolic rules emerging from connectionism. Human thinking also accumulates observations and experiences first, then summarizes patterns. Induction comes before deduction, and deduction cannot replace induction. As mentioned before, if deduction is not treated with enough rigor, it becomes very fragile, and one still needs to resort to the observations, experiences, and empirical verification that correspond to connectionism.

And why I believe that pure structural compression might not be complete is because I am reminded of the mixed rules experiment; direct compression is not good enough. That is, as mentioned earlier, mixing multiple simple rules rapidly increases the difficulty of learning the task. But if one has insight into the structure of the problem, one can reduce the difficulty of the problem by decoupling it. Perhaps this is the role of logical cognition; a person cannot deal with the world solely through intuition.

9.Exploring the black box should be simpler with this paradigm. First, for trainable problems in this paradigm, the loss can be reduced to be very close to 0. In this case, first, it is indeed parameters learned through gradient descent in a black box. Second, unlike image recognition or NLP, the learning of symbolic rules can be studied more systematically. By training the same model on different trainable problems and then studying the changes in the neural network's weights or validation outputs during the training process, it is easier to conduct research in a sterile laboratory environment than with the previous pattern recognition paradigms for images, NLP, and audio.

Moreover, what makes this even more advantageous is that it is very simple to manually design the required task rules and generate the datasets.

At the same time, experiments related to image reasoning can visualize the formation process of a machine mind. For example, in the task of drawing an incircle in a triangle, if I save the current eval image at regular step intervals during training, I can very intuitively see the progress of the learning process and how the model might be learning the related task.

10.If this paradigm of mine can realize its true potential in the future, it would be possible to predict reliable outputs for new inputs without knowing the working principles of some systems and without needing to program, just by training a neural network. This is somewhat similar to what Andrej Karpathy called the [22] Software 2.0 model.

This method should have potential in the field of AI for Science in the future.

11.This paradigm can also extract rules, i.e., given input and output, output the transformation rule. I have already conducted an experiment on the cellular automaton task, which was very successful. However, in that experiment, I controlled the generation of dataset samples to ensure that the input and output could uniquely determine a transformation rule and the number of evolution layers. But in real-world problems, it might often be impossible to satisfy the condition of extracting a transformation rule from a single sample. My idea here is that if rules are to be extracted from a large amount of input and output data, perhaps different kinds of voting methods could be used, which might all achieve good results.

12.If we can gain some understanding of the relationship between the model's weights and the rules it has fitted, would it be possible to proactively manually design weights to simulate a certain rule? Of course, this is very likely to be very difficult, even if the clear symbolic rules make it seem a bit simpler. So, perhaps using training to obtain the weights is still a better method. The value of this question may be mainly theoretical.

13.A phenomenon observed during training is that sometimes when the loss decreases to be very close to 0, it suddenly jumps to a high loss value. It is speculated that this might be due to the non-smooth nature of this type of task.

14.A vision: This kind of training can be completely automated. A person only needs to collaborate with an AI to write the training set scripts, and the subsequent dataset generation, training, and analysis can be done automatically. This would make research very fast. And whether each type of task is learnable or not, and what its learning difficulty is, will become a new kind of data for algorithm scientists and even mathematicians to study.

15.Regarding the limitation of this paradigm to synthetic data, my thought is that I haven't actually made any changes or innovations to the model itself. Once the data becomes noisy real-world data, the premise of this paradigm no longer exists. So, this paradigm may not be directly applicable to complex real-world problems. Its value may lie in reminding us that deep learning neural networks themselves have the ability to fit symbolic rules, and it will provide an excellent environment for studying the behavior of neural networks. Perhaps when I have a better understanding of the nature of neural networks, I will have a more profound understanding of the current hallucination problems of LLMs and the failure of CV models in some scenarios, and thus be able to build more controllable and interpretable AI.

Additionally, this paradigm may have more direct applications to some problems in the symbolic world.

16.Regarding the recent paper from Apple, [15] The Illusion of Thinking: Understanding the Strengths and Limitations of Reasoning Models via the Lens of Problem Complexity, this paper describes some problems that large models cannot reliably reason about. I have trained on 4 problems—Tower of Hanoi, Blocks World, River Crossing, and Checkers Jump—using my paradigm and achieved success, although the Tower of Hanoi problem has some subtleties as mentioned above, and the convergence of the Blocks World problem was not perfect. For details, please see the open-source code.

Also, at that time, I noticed some opposing views, saying that the inability to reason properly was due to token limitations. So, for the Tower of Hanoi problem, I designed a format 1>2 to represent moving the top disk from the first peg to the second, separated by semicolons. Then I asked a large model to use this format to reply with the solution to the Tower of Hanoi problem, and found that n=6 was about the limit. Beyond that, problems would occur. So, I believe that large models, without calling tools (like Python), indeed cannot perform effective reasoning on these problems. The script to verify whether the answers in this format are correct is also open-sourced, see the code repository eval_hanoi.py.

17.Rethinking the meaning of this paradigm: This paradigm seems strikingly similar to the state of the computer vision field before the emergence of [1] AlexNet. At that time, the mainstream methods relied on handcrafted features (like [18] SIFT, [19] HOG), which can be seen as a form of "a priori symbolic knowledge" about the visual world. The success of AlexNet was precisely in replacing this handcrafted bias with end-to-end learning (induction). In the field of reasoning, current neuro-symbolic computing, whether by modifying

architectures or embedding logic, is essentially injecting human understanding of the reasoning process as a priori knowledge into the model. And this predetermined structure, this inductive bias, may be unrealistic or inappropriate.

Act VI: Universality Beyond Architecture

In this section, I explore whether other deep learning models possess the same capabilities as the Transformer. I explored a total of 5 types: MLP, RNN/LSTM, CNN, UNET, and Diffusion.

I used MLP and RNN/LSTM to fit the symbolic version of the cellular automaton and the trapping rain water algorithm problem. Of course, the input here is no longer Transformer-like tokens, but real numbers.
CNN was used to fit the cellular automaton dataset with image input and symbolic output, and the image version of the trapping rain water dataset.
UNET and Diffusion were used to fit the cellular automaton dataset where both input and output are images.

Dataset Generation Scripts:

Symbolic CA: generate_cellular_automata_1d.py
Symbolic Trapping Rain Water: generate_trapping_rain_water_decoupled.py
Image-based CA: generate_cellular_automata_image_and_label.py
Image-based Trapping Rain Water: generate_trapping_rain_water_image_to_symbol.py

6.1 Cellular Automata

Dataset Introduction

All are 1D 36-cell cellular automaton datasets, with a dataset size of 300,000, evolution rule 110, and 2 evolution layers. The only exception is that the MLP model might be too suitable for the cellular automaton task, with too strong a fitting ability, so I increased the number of evolution layers to 6.

Model structures can be found in the open-source code.

6.1.1 CNN Model

Training Code: train_convnext.py

I used the [5] ConvNeXt model. Here is the training process.

1000step Train Loss: 0.49467600 | Val Loss: 0.27210647 | Val Bit Acc: 84.55833333% | Val Exact Match: 0.10000000%
2000step Train Loss: 0.11814510 | Val Loss: 0.05764289 | Val Bit Acc: 97.36018519% | Val Exact Match: 36.80000000%
3000step Train Loss: 0.04807972 | Val Loss: 0.03248511 | Val Bit Acc: 98.63425926% | Val Exact Match: 56.20000000%

4000step Train Loss: 0.00763491 | Val Loss: 0.00266492 | Val Bit Acc: 100.00000000% | Val Exact Match: 100.00000000%
5000step Train Loss: 0.00186690 | Val Loss: 0.00128128 | Val Bit Acc: 100.00000000% | Val Exact Match: 100.00000000%

As you can see, it converged extremely quickly, reaching 100% exact match accuracy on the validation set by 4000 steps. This proves that, at least for this problem, CNN has symbolic rule learning ability.

6.1.2 MLP Model

Training Code: train_mlp.py

For a giant MLP, the cellular automaton dataset with 2 evolution layers converged too quickly. I had to regenerate a dataset with 6 evolution layers. It turned out to also converge very quickly, reaching 100% exact match accuracy on the validation set at the end of 5 epochs.

A preliminary analysis suggests that this is because the MLP preserves long-range connections between neurons extremely well, so it might not be worse than a Transformer on this problem.

6.1.3 RNN Model

Training Code: train_lstm.py

There are a few small design choices here. First, I must input the entire initial state of the cellular automaton in the first time step, not one bit per time step. Second, I cannot use autoregressive output, so I must take the output of a certain time step as the result. Third, I deliberately chose the number of time steps to be different from the number of evolution steps of the cellular automaton. Since the number of evolution steps is 2, I chose the output time step of the RNN model to be 3. Although having the number of time steps and evolution steps be exactly the same would certainly be the ideal model design, here I want to prove my earlier point that the model does not execute tasks according to human-assumed logic or algorithmic steps. In this experimental setup, the model is forced to learn a transformation function that, when repeated 3 times, is equivalent to the cellular automaton's rule 110 being applied 2 times. The success of this setup provides strong support for my view on this paradigm.

And the experimental results also prove that it converged in 1000 steps, already reaching 100% exact match accuracy on the validation set.

I later changed the number of time steps to 5/7/9, and it could still reach 100% exact match accuracy in a very short time.

6.1.4 UNET Model

Training Code: train_unet.py

Used MSELoss, but it quickly got stuck at a high value of 0.0915 and did not decrease further. This might be because the architecture itself is not suitable for this task, which requires further analysis.

6.1.5 Diffusion Model

Training Code: train_diffusion.py

After the loss became very low, the generated images were still incorrect. However, it can be seen that the diffusion model learned to generate checkerboard-like images, but the black and white patterns of the checkerboard in the eval image were completely different at each evaluation. It is likely not suitable for this task. Additionally, the iterative image generation of diffusion models is likely unsuitable for any task in this paradigm.

6.2 Trapping Rain Water

Task Description: Derived from the classic LeetCode algorithm problem, 42. Trapping Rain Water - LeetCode

The decoupled output format is used, same as in the experimental section above. The difference is that the number of bars is adjusted to 12, so both input and output are 123 = 36 = 66. This facilitates CNN-related experiments to input a 6x6 black and white checkerboard image.

6.2.1 MLP Model

Training Code: train_mlp.py

Best result: Validation Loss: 0.0002, Bit Acc: 99.99%, Exact Match: 99.90%

6.2.2 RNN Model

Training Code: train_lstm.py

Here I chose 3 time steps.

Best result: Validation Loss: 0.0001, Bit Acc: 100.00%, Exact Match: 100.00%

6.2.3 CNN Model

Training Code: train_convnext.py

I used the [5] ConvNeXt model.

Best result: Epoch 6 | Train Loss: 0.00003849 | Val Loss: 0.00001896 | Val Bit Acc: 100.00000000% | Val Exact Match: 100.00000000%

6.3 Summary and Analysis

I used the cellular automaton experiment to demonstrate symbolic rule inference ability and the trapping rain water experiment to demonstrate algorithm fitting ability. And this series of experiments proves that this paradigm is not exclusive to the Transformer; many other models have similar capabilities. This leads to an inevitable conclusion: this paradigm is an inherent capability of connectionist deep learning neural networks. Moreover, it is very likely that the Transformer is not necessarily the best model, or rather, for different problems within this paradigm, the best model will also be different. Due to time and energy constraints, I am personally unable to explore this further.

Act VII: Bridging the Gap: A Continuum from Ideal Rules to the Real World

In this section, I mainly discuss two issues. In all the discussions above, I have focused on datasets with precise transformation rules. So, in the first part of this section, I will introduce random perturbations to the inputs and outputs of datasets with precise rules and observe the training progress and final loss convergence.

In the second part, I design an experiment to prove that the rule-learning ability and interpolation ability implied by this paradigm are not contradictory but can coexist.

7.1 Perturbation of Precise Rules

I chose the 1D cellular automaton task, with evolution rule 110, 2 evolution layers, and a 30-bit 1D cellular automaton. The specific perturbation method is, during data generation, to randomly flip the input or output with a certain probability (i.e., 1->0 or 0->1). For simplicity, I only tested flipping the input alone or flipping the output alone. The validation set used is the cellular automaton dataset with the precise, non-randomly flipped rule.

Dataset Code: generate_cellular_automata_1d_perturbed.py
Training Code: train_tiny_transformer.py

7.1.1 Test of Random Flipping on the Output

Random Flip Probability	Stable Eval Loss
0.1%	0.000692
1%	0.008490
10%	0.094094

7.1.2 Test of Random Flipping on the Input

Random Flip Probability	Stable Eval Loss
0.1%	0.001895
1%	0.021377

Random Flip Probability	Stable Eval Loss
10%	0.230316

7.1.3 Summary and Discussion

It can be seen that perturbations to either the input or the output increase the difficulty of the learning task as the perturbation probability increases, which is very natural. Also, perturbations to the input have a greater impact than perturbations to the output. This may be due to the inherent non-smooth nature of the rule-based task. It can be speculated that if the number of evolution layers is increased, input perturbations will make the task even more difficult to learn. So, roughly speaking, there may be a continuous spectrum from noiseless, precise transformation rule datasets to real-world datasets.

7.2 The Unity of Interpolation and Rule Learning

This section uses an experiment to prove that the rule-learning ability and interpolation ability of neural networks are not mutually exclusive, but are integrated.

7.1.1 Experiment Description

To verify the ability of a neural network to integrate discrete rule learning and continuous value processing in a single task, we designed a logic-aware mixing experiment. The experiment is based on the cellular automaton image generation task, where the 36-cell 1D cellular automaton state is arranged in a row-major order into a 6x6 checkerboard image, with black representing 1 and white representing 0. However, the input and output were modified as follows:

Input Side: The input image is no longer pure black and white blocks. Instead, the cells representing logical black and white are replaced with color blocks of randomly chosen grayscale values within two different grayscale intervals, thus introducing continuous perceptual information.
Output Side: The output image generated by the model must satisfy two conditions simultaneously:

- Logical Correctness: The black and white pattern of the cells must strictly follow the multi-step evolution rule of the cellular automaton.
- Perceptual Association: The final grayscale value of each cell must be precisely calculated based on the original grayscale value of its corresponding input cell and a simple conditional function (white remains, black is inverted). Specifically, if the grayscale value of the input cell is x, and if the logical pattern of the output cell is the same as the input cell, the grayscale value of the output cell remains x; otherwise, it is 255-x.

This design cleverly forces the model to both see through the continuous grayscale values of the input to perform discrete logical reasoning and to remember these grayscale values to

complete the final continuous value mapping, thus testing its rule-learning and interpolation capabilities in an indivisible task.

Dataset Code: generate_cellular_automata_1d_to_grid_image_interp.py
Training Code: train_image2image.py
Loss: MSELoss

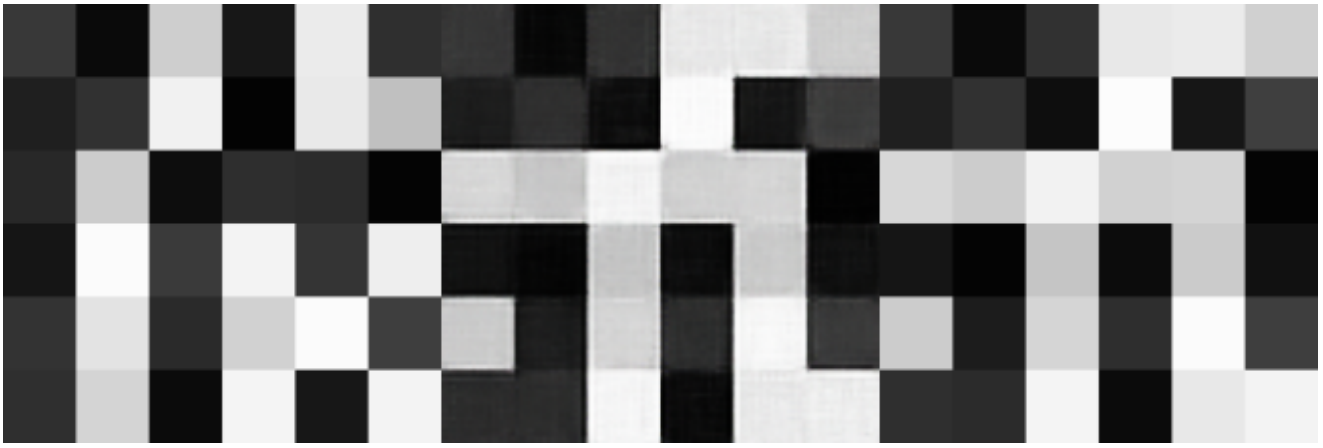
Some task parameters:
Grid dimensions: 6x6
Grid cell size: 40x40 pixels
Image Resolution: 240x240
Cellular automaton rule: rule 110
Number of rule evolution layers: 2

7.1.2 Experimental Results

After 3200 training steps, the MSE loss on the validation set had already reached 0.000899.

This is the eval image at 3200 steps.

Left is the input image, right is the ground truth, and middle is the generated image.



7.1.3 Summary and Discussion

As can be seen, this experiment was a great success. It fully demonstrates that both interpolation and rule-learning abilities are inherent, essential capabilities of neural networks. And they may not be parallel, but integrated. For example, in this case, the model did not try to first understand the evolution rule of the cellular automaton and then remember the grayscale value of each cell. What it was actually doing was simply gradient descent. It is still as mentioned before: connectionist gradient descent can approximate discrete symbolic rules.

Act VIII: Conclusion — The Dawn of Reliable AI

The culmination of this research provides a clear and profound answer to a fundamental question that has perplexed the field of artificial intelligence for decades: Can a neural

network, born of probability and statistics, truly understand and execute precise, deterministic logical laws?

My answer is a resounding yes. And the key to achieving this lies not in larger, more complex model architectures, but in a fundamental paradigm shift: I abandoned the mimicry of noisy real-world data in favor of ideal data programmatically generated from pure rules; I relinquished the fragile chain of autoregressive prediction and embraced a parallel, one-shot, holistic solving framework.

I did not stop at proving the superiority of a specific advanced architecture, but conducted a systematic test. I arrived at a subversive, unexpected, yet perfectly first-principles-aligned discovery: a Multi-Layer Perceptron, with no structural biases and composed purely of a vast number of parameters, exhibited the most powerful, perfect, zero-error generalization capability on a complex logical evolution task. The success of the Recurrent Neural Network pushed this discovery to a philosophical level—it proved that a neural network could even "create" an iterative algorithm that is unintuitive to me but functionally perfectly correct, in order to satisfy the final mathematical constraints.

This series of experiments collectively points to an unshakable conclusion: the ability for precise, algorithmic reasoning is not the "patent" of any particular architecture. It is an inherent, deeply hidden, universal potential of connectionist systems themselves, which can be "awakened" by the right paradigm. The superiority of an architecture, under this paradigm, is no longer a question of "can or cannot," but rather determines the "efficiency" and "capability boundary" with which this potential is awakened.

This is not just a technological breakthrough; it is a watershed. It means that, for the first time, we have in our hands the ability to "sculpt" abstract, precise, human-understandable laws into the "marble" of a neural network, without fear of it producing random "hallucinations."

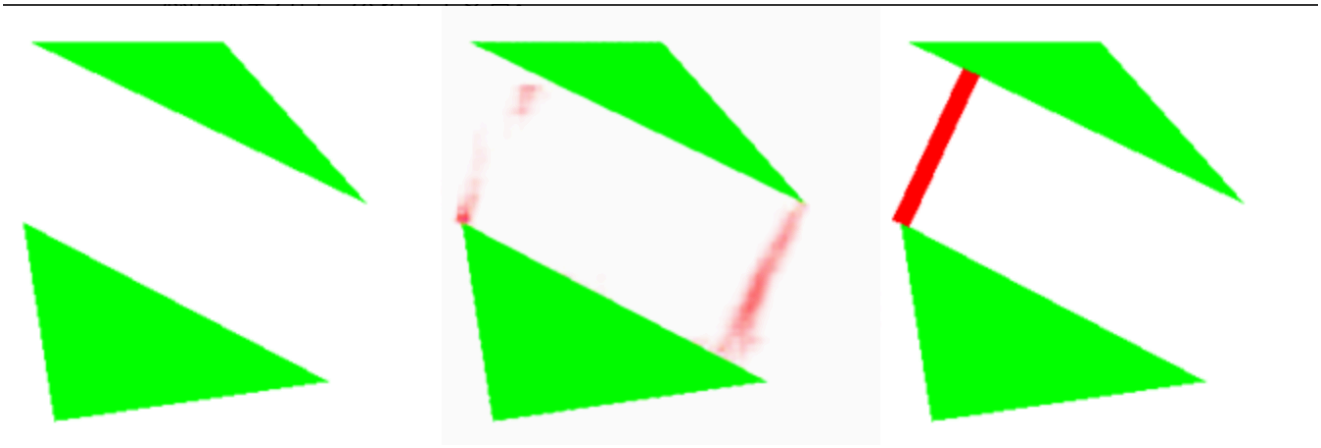
This heralds the possibility of a new era: an era driven by truly reliable, controllable, and interpretable neural networks. An era where we can not only converse with AI, but also trust it to perform precise calculations, simulate the physical world, and even assist us in the most rigorous of scientific discoveries.

Appendix:

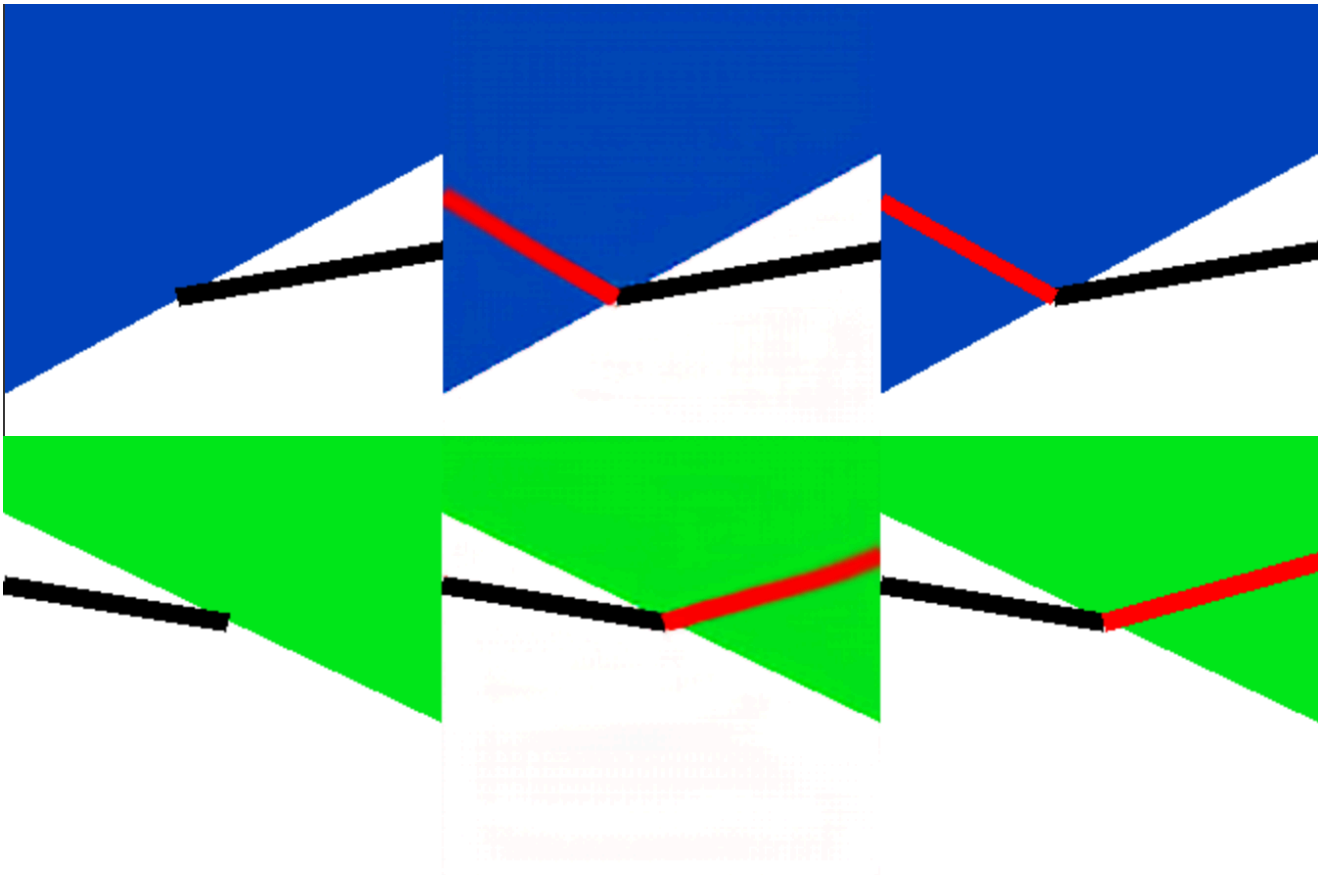
Another task: Visualization of results during learning

Task Description: Input image is a white background with two arbitrary, non-overlapping green solid triangles. The task is to draw a red line segment indicating the shortest distance between the two triangles.
Analysis: It seems the model learned that the line segment of the shortest distance is either the perpendicular line from a vertex to the corresponding edge of the other triangle, or the

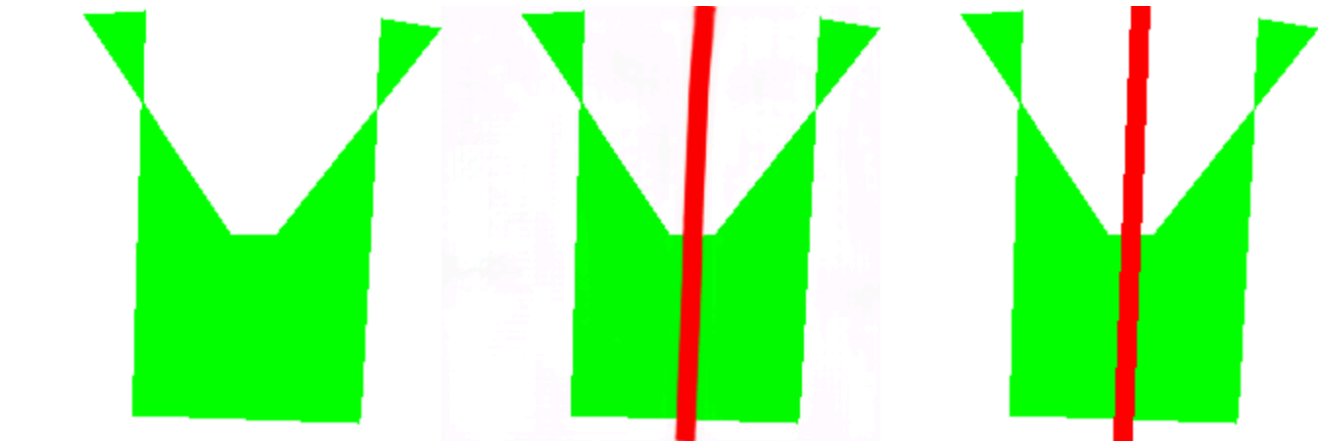
line connecting two vertices. But here, it has difficulty determining which one has the shorter distance, so it draws both perpendicular lines, but both are blurry. Left is the input image, middle is the model-generated image, right is the ground truth.



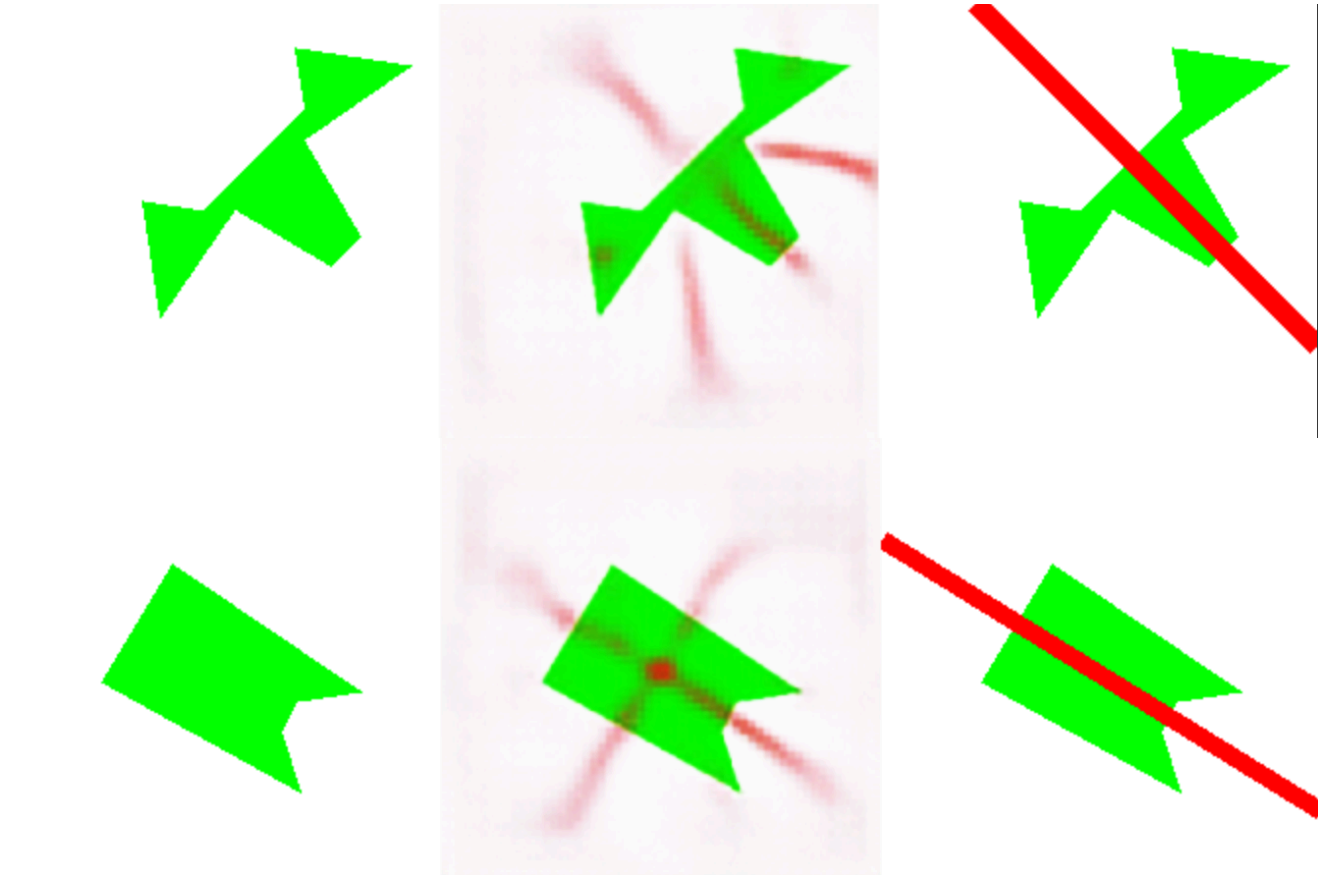
Training results for variable refractive index



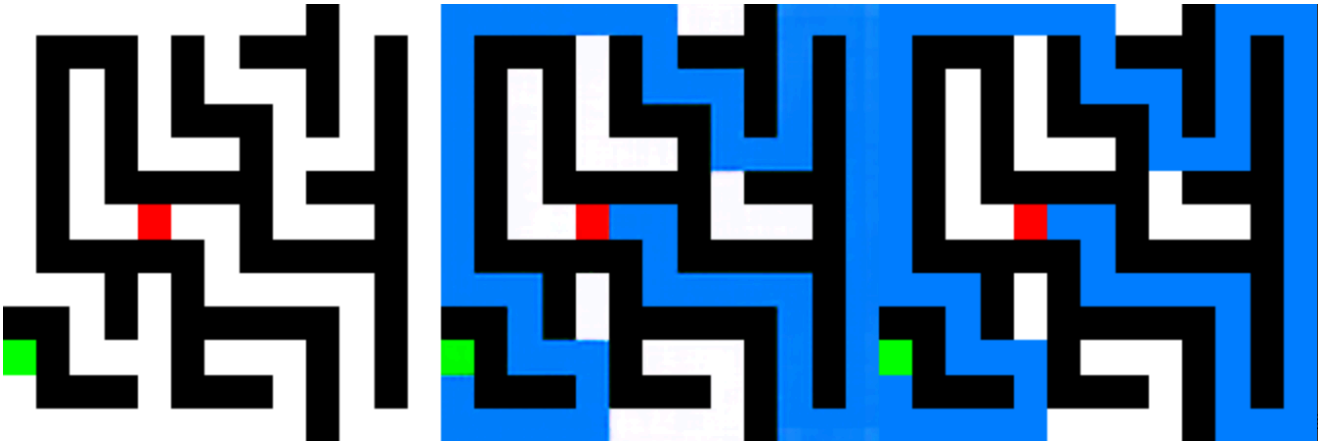
Intermediate process of learning the axis of symmetry



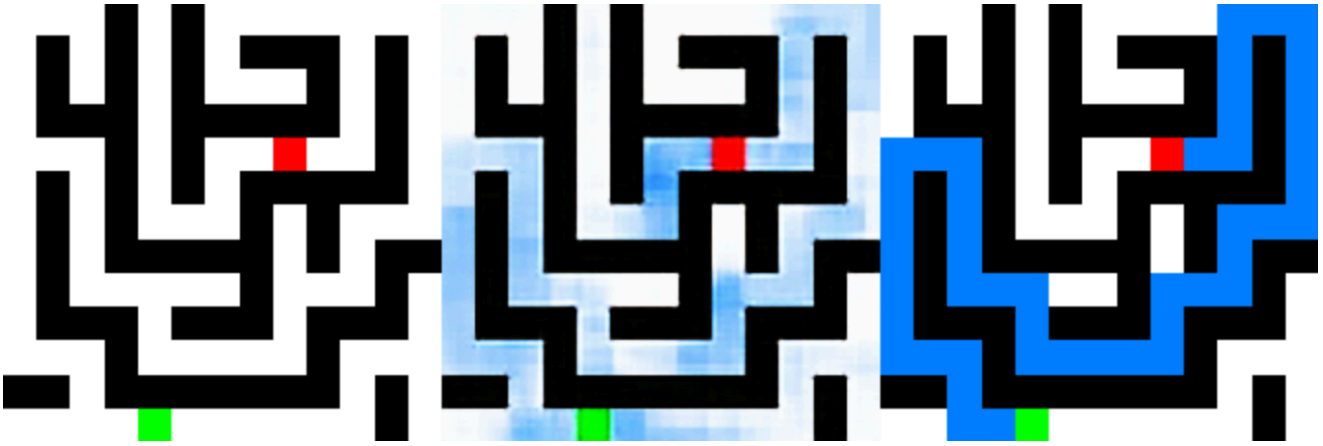
Intermediate state



Maze experiment



Intermediate learning state



Finally, I noticed that the arc-agi-3 task has been released, changed to a game-playing mode. I believe it can be solved in the following ways. First, still using the same method as with the previous ARC-AGI datasets, I provide the logic, and use a program to generate a large dataset. Second, directly use the image2image swin-unet as a state-action policy network, combined with a PPO algorithm, and train it through the environment (the environment is equivalent to potentially infinite data). However, due to my current energy and time constraints, I have not yet worked on this task. This method still requires a large amount of data and cannot directly solve arc-agi-3.

arc-agi-3 URL: ARC-AGI-3[14]

All experiments conducted:

A. Symbolic Rule Learning

generate_conditional_add_subtract.py: This script is used to investigate the model's ability to handle "rule conflicts" or "conditional logic." It aims to test whether the model fails to learn when multiple rules are implicitly mixed within a dataset, and whether it can learn successfully when an explicit indicator is provided.
generate_add_binary_modulo.py: An early basic arithmetic experiment to test the model's ability to learn modular addition (or "truncated addition"), an operation common in fixed-width integer arithmetic of computer hardware.
generate_multiply_binary.py: A benchmark test for binary arithmetic capability, generating a dataset for the multiplication of N-bit integers.
generate_multiply_binary_no_carry_phase1.py: The first phase of the multiplication "decoupling" experiment. It aims to test whether the model can learn the first step of multiplication: bit-wise multiplication and shifted addition without carry, decomposing a complex multiplication problem into a simpler counting problem.
generate_multiply_binary_from_counts_phase2.py: The second phase of the multiplication "decoupling" experiment. It aims to verify whether a separate model can learn to handle complex carry logic, i.e., calculating the final binary product from a "no-carry count vector."
generate_add_hexadecimal.py: Compares the model's learning ability across different

symbol systems. This script aims to verify whether the model is learning the abstract mathematical concept of addition or merely patterns specific to binary symbols.
generate_multiply_decimal.py: Tests the model's ability to process non-binary symbol inputs (0-9 characters) and perform arithmetic operations (multiplication).
generate_add_n_base_with_shuffle.py: A crucial, decisive experiment in my research, designed to completely separate the model's "surface pattern matching" ability from its "abstract structure learning" ability.
generate_add_binary_with_position_shuffle.py: This is the "position shuffle" part of the "semantic shuffle" series of experiments. It aims to verify whether the model relies on a fixed spatial structure of the input or can learn position-independent abstract relationships.
generate_add_hidden_constant.py: Tests the model's ability to infer hidden rules or parameters from a large number of samples without any direct clues. This is similar to a simplified System Identification problem.
generate_multitask_alu.py: This script aims to build a multi-task learning scenario simulating an Arithmetic Logic Unit (ALU). It tests whether the model can, in a single forward pass, perform multiple different, well-defined computational tasks in parallel on the same input.
generate_modulo_operation.py: Investigates the model's ability to learn the Modulo Operation, a crucial operation with "cyclic" properties in number theory and computer science.
generate_rsa_encryption.py: Tests the model's ability to learn highly non-linear, computationally "hard" deterministic rules. RSA encryption is a typical example.
generate_cellular_automata_1d.py: Used to generate evolution datasets for 1D cellular automata (CA) to test the model's ability to learn and execute local, deterministic rules.
generate_game_of_life_2d.py: Generates datasets for a 2D cellular automaton—Conway's Game of Life. This task is more complex than 1D CA, requiring the model to understand neighborhood relationships in a 2D space.
generate_cellular_automata_1d_multistate.py: An extension of the 1D cellular automaton experiment, testing the model's ability to handle non-binary state spaces.
generate_cellular_automata_programmable.py: Tests the model's "programmability" or "meta-learning" ability. The model must not only learn the CA evolution process but also be able to execute the evolution according to different rules given in each input.
generate_deduction_chain_text.py: Generates multi-step logical reasoning tasks to test the model's ability to perform symbolic deduction, similar to a simplified theorem prover.
generate_deduction_multirule_text.py: Tests whether the model, when faced with multiple independent, unrelated rules, can correctly "route" to the appropriate rule based on a query and make a judgment.
generate_deduction_multirule_text_v2.py: Tests whether the model, when faced with multiple independent, unrelated rules, can correctly "route" to the appropriate rule based on a query and make a judgment.
generate_deduction_multirule_binary.py: An optimized format version of the multi-rule reasoning task, aiming to test whether a compact binary encoding is more conducive to model learning than a sparse text format.
generate_deduction_fixed_depth.py: Tests the model's multi-step reasoning ability in

symbolic deduction tasks with a clear structure and fixed depth.
generate_function_composition.py: Tests the model's ability to learn Function Composition. This requires the model to act like an interpreter, parsing instructions sequentially and transforming data accordingly.
generate_cellular_automata_inverse_rule90.py: Tests the model's ability to solve an Inverse Problem. Given the output of a deterministic system, the model needs to infer the possible input that satisfies specific constraints (sparsest and unique).
generate_count_set_bits.py: Tests the model's ability to perform a global aggregation operation. Unlike local rules, counting requires the model to integrate information from the entire input sequence.
generate_sum_pattern_positions.py: Tests the model's ability to perform a more complex, group-based parallel aggregation task. The model needs to first segment the input, then classify each segmented pattern, and finally accumulate the positional information of patterns belonging to the same class.
generate_sum_pattern_positions_v2.py: Tests the model's ability to perform a more complex, group-based parallel aggregation task. The model needs to first segment the input, then classify each segmented pattern, and finally accumulate the positional information of patterns belonging to the same class.
generate_sum_pairwise_hamming_distance.py: Tests the model's ability to perform a complex task requiring two levels of nested aggregation. The model needs to first perform global statistics on each bit position and then accumulate the results from all bit positions.
generate_circular_shift.py: Tests the model's ability to learn shift operations, particularly the circular shift, a common operation in cryptography and low-level programming.
generate_multiply_matrix_3x3.py: Tests the model's ability to learn a structured algebraic operation (matrix multiplication), which requires more complex "data routing" and "multiply-accumulate" capabilities than simple scalar operations.
generate_evaluate_boolean_expression_text.py: Tests the model's ability to parse and evaluate a simple Domain-Specific Language (DSL), which is a step beyond the fixed-structure expression evaluation from before.
generate_evaluate_arithmetic_expression.py: Trains the model to perform symbolic expression evaluation, which requires understanding operator precedence, variable substitution, and arithmetic operations.
generate_evaluate_arithmetic_expression_no_multiply.py: A simplified version of the expression evaluation task, designed to reduce learning difficulty by removing the multiplication operation.
generate_evaluate_arithmetic_expression_no_multiply_small_range.py: A further simplification of the "no multiplication" version, designed to further reduce learning difficulty by narrowing the numerical range.
generate_check_boolean_equivalence.py: Tests the model's ability to judge the logical equivalence of Boolean algebra. This is an abstract symbolic reasoning task that requires the model to understand the structure of expressions and Boolean laws.
generate_polynomial_shift_coefficients.py: Tests the model's ability to learn an abstract algebraic transformation rule. This task requires the model to understand the intrinsic

structure of polynomial expansion.
generate_convolution_2d.py: Tests the model's ability to learn the fundamental image processing operation of 2D convolution (Conv2D), and explores whether it can infer the hidden, fixed rule (the convolution kernel itself) from input-output pairs.
generate_simple_block_cipher.py: Tests the model's ability to "crack" or learn a simple but non-trivial custom encryption algorithm. This task represents a class of complex symbolic transformation rules with high chaos and avalanche effects.
generate_sin_function_float32.py: Tests the model's ability to fit a continuous, periodic, non-linear function (sin(x)), using standard 32-bit floating-point format for input and output.
generate_sin_function_float64_to_int12_deprecated.py: Another encoding attempt for the sin function fitting task, aimed at exploring the effect of using higher-precision float input and lower-precision quantized binary output on learning.
generate_sin_function_float32_to_quantized_int.py: Tests the model's ability to fit a continuous, periodic, non-linear function (sin(x)), and explores the impact of different input/output encoding schemes on learning performance.
generate_multiply_binary_modulo.py: As part of the basic arithmetic experiments, tests the model's mastery of truncated multiplication (or modular multiplication).
generate_explainable_two_step_calculation.py: Tests the model's ability to output "intermediate steps" or a "chain of thought," serving as a direct validation of "functional interpretability."
generate_chess_positions_by_random_moves.py: Rapidly generates a large number of plausible, legal Chinese chess positions by simulating a completely random player.
generate_chess_positions_by_random_placement.py: Generates a large number of atypical, but mostly legal, Chinese chess positions by randomly placing pieces on the board (rather than simulating moves), used for stress-testing the model's robustness.
generate_chess_positions_from_engine_self_play.py: Generates a large number of high-quality, strategically sound Chinese chess positions (in FEN format), serving as a base data source for training a chess AI.
generate_preprocess_legal_moves.py: A data preprocessing script used to convert a dataset of FEN positions into a "legal move prediction" task that the model can directly learn from.
generate_chess_resolve_check_task.py: Generates a dataset specifically for the tactical scenario of "Resolving a Check" in Chinese chess. This task requires the model, when in a state of check, to find all legal moves that can resolve the check.

B. Algorithm Learning

generate_sort_integers.py: Tests the model's ability to perform a basic sorting algorithm, a classic non-local algorithm that requires comparison and rearrangement of input elements.
generate_edit_distance.py: Tests the model's ability to learn to solve dynamic programming problems. Edit distance is a typical DP problem that conceptually requires the model to construct a 2D solution matrix.
generate_edit_distance_explainable.py: A core experiment for "functional interpretability." It requires the model not only to give the final answer (edit distance) but also to output the complete "chain of thought" (the editing process) to reach it.

generate_maze_random_walls.py: Tests the model's basic pathfinding ability in randomly generated "porous" mazes.
generate_maze_dense.py: Tests the model's ability to perform path planning in complex, human-like "dense" mazes, which are more challenging than random-wall mazes.
generate_blocks_world_arbitrary_goal.py: Solves the classic "Blocks World" planning problem, a benchmark task in AI planning. This version allows for specifying arbitrary initial and goal states.
generate_blocks_world_fixed_goal.py: A simplification of the "Blocks World" task, designed to test the model's learning ability in a more structured state space with a fixed goal state.
generate_blocks_world_fixed_goal_multilabel.py: A further improvement on the "Blocks World" task, testing the model's ability to handle multi-label classification problems by allowing multiple optimal solutions, more realistically reflecting that planning problems may have equivalent optimal paths.
generate_blocks_world_fixed_goal_multilabel_fixed_format.py: The final optimized version of the "Blocks World" task, designed to provide the model with a clearer, more structured learning target by improving the input representation.
generate_checkers_jump_1d.py: Solves a planning problem of moving checkers in a 1D space, a problem originating from a well-known paper by Apple, used to test the reasoning bottlenecks of large language models.
generate_river_crossing_puzzle.py: Solves a classic constraint satisfaction and state-space search problem—the "N couples river crossing." This task comes from an Apple paper used to reveal the limitations of large language models on certain types of reasoning tasks.
generate_trapping_rain_water_aggregate.py: An initial attempt to solve the "trapping rain water" algorithm problem, designed to test the model's ability to learn an aggregated output (rather than a decoupled one). Experimental results showed that requiring the model to directly output the total sum is much harder than outputting detailed information for each position.
generate_trapping_rain_water_decoupled.py: Solves the classic "Trapping Rain Water" algorithm problem (LeetCode Hard). The success of this task demonstrates the model's ability to learn complex algorithms requiring global information and, through the idea of problem decoupling, proves the significant impact of output format design on model learning efficiency.
generate_trapping_rain_water_2d.py: An extension of the 1D "trapping rain water" problem, solving a 2D version. This task requires the model to understand concepts of "enclosure" and "boundary" in 2D space, presenting a more complex global information processing challenge.
generate_skyline_max_height_aggregate.py: An initial attempt to solve the "skyline" problem, requiring the model to predict only the maximum height from the final heights of all buildings. This task is used to compare the learning difficulty of aggregated vs. decoupled outputs.
generate_skyline_all_heights_decoupled.py: Tests the model's ability to solve a global optimization problem with 1D spatial constraints. The prototype is LeetCode's "Max-Height Skyline." By decoupling the output, the model is required to predict the height of each

building, rather than just the maximum.
generate_hanoi_tower_path_strategy_sep_format.py: An early experimental script for the Tower of Hanoi problem, aimed at testing if the model can learn the strategy on the optimal path. It used a separator-style input format and predicted the action as a 6-class classification problem.
generate_hanoi_tower_global_strategy_fixed_format.py: An improvement on the early Hanoi experiments, this script uses a more model-friendly fixed-slot input format to verify the impact of input representation on learning efficiency.
generate_hanoi_tower_compare_formats.py: A comparative experiment script that generates two different input formats (separator vs. fixed-slot) for the same Hanoi problem to systematically evaluate the effect of different data representations on the model's learning of recursive strategies.
generate_hanoi_tower_compare_formats_and_strategies.py: A more comprehensive comparative experiment for Hanoi. It not only generates two input formats but also two different datasets: one containing only states on the optimal path ("path strategy") and another containing all reachable states ("global strategy"), to explore the difference in the model's ability to learn local optimal paths versus a global optimal policy.
generate_hanoi_tower_build_full_state_graph.py: A magnum opus for the "Tower of Hanoi" research, aiming to deeply analyze the model's understanding of recursive structures through various data representations and sampling strategies. It is a self-contained data factory.
generate_hanoi_tower_sample_from_state_graph.py: A post-processing and sampling script that uses the complete knowledge base generated
by generate_hanoi_tower_build_full_state_graph.py to precisely extract specific subsets of training data, such as "twisted paths" or the "hardest parts," for finer-grained ablation studies.
generate_sokoban_planning_astar.py: Solves the classic "Sokoban" planning problem.
generate_sokoban_planning_full.py: Solves the classic "Sokoban" planning problem. This is a highly difficult AI task as it involves searching in a huge state space where actions change the state of the environment.
generate_sokoban_planning_claude_deprecated.py: An early, more logically complex attempt that failed to consistently generate high-quality datasets. (Deprecated)
generate_min_swaps_for_checkerboard.py: Solves a highly constrained matrix rearrangement problem: find the minimum number of swaps (of arbitrary rows and columns) required to transform a 0/1 matrix into a "checkerboard" pattern (adjacent elements differ).
generate_min_flips_for_alternating_binary.py: Tests the model's ability to solve a string optimization problem based on bit flips, which can be cleverly mapped to a sliding window problem for a solution.
generate_min_swaps_for_checkerboard_v2.py: Solves a highly constrained matrix rearrangement problem: find the minimum number of swaps (of arbitrary rows and columns) required to transform a 0/1 matrix into a "checkerboard" pattern (adjacent elements differ).
generate_matrix_flip_strategy.py: Solves a classic matrix optimization problem (maximizing the number of 1s). This version aims to test if the model can learn a "policy" rather than the

final result.
generate_matrix_flip_max_score.py: Tests the model's ability to learn a matrix optimization problem that requires a two-step greedy strategy (first flip rows, then columns) to achieve a global optimum. This version requires the model to directly output the final aggregated result (the score).
generate_min_prefix_flips.py: Tests the model's ability to learn a sequential, state-dependent greedy algorithm.
generate_min_k_bit_flips.py: Tests the model's ability to learn a sequential, state-dependent greedy algorithm, and tests if it can use part of the input (k) as a "parameter" to guide the processing of another part (nums).
generate_min_k_bit_flips_fixed_k.py: Tests the model's ability to learn a sequential, state-dependent greedy algorithm. In this version, the environmental parameter (k=2) is fixed and hidden; the model must learn it implicitly from the data.
generate_special_binary_string_recursion.py: Tests the model's ability to learn a recursively defined string transformation rule. This problem (LeetCode Hard "Special Binary String") requires recursive decomposition and reassembly of the input.
generate_min_flips_for_chunked_binary.py: Tests the model's ability to learn a string transformation optimization problem based on local chunks.
generate_count_connected_components.py: Tests the model's basic understanding of graph structures, particularly the core concept of "connectivity."
generate_check_graph_connectivity.py: Another core test of the model's foundational graph theory abilities, with the task of determining if a path exists between any two points in a graph.
generate_minimize_malware_spread.py: Solves a graph-based virus spread optimization problem (LeetCode Hard "Minimize Malware Spread"). The model needs to understand graph connectivity and evaluate the impact of removing different nodes on global spread.
generate_count_islands_1d.py: Tests the model's ability to perform pattern recognition and counting on a 1D sequence.
generate_largest_island_by_adding_one_cell.py: Solves an algorithm problem involving graph traversal and global optimization (LeetCode 827). The model needs to evaluate all possible "land-filling" positions and choose the one that maximizes the area of the resulting merged island.
generate_largest_island_by_adding_one_cell_v2.py: Solves an algorithm problem involving graph traversal and global optimization (LeetCode 827). The model needs to evaluate all possible "land-filling" positions and choose the one that maximizes the area of the resulting merged island.
generate_find_articulation_points.py: Tests the model's ability to identify "articulation points" or "bridges" in a graph, an important concept in graph theory.
generate_nim_game_zeckendorf.py: This experiment aims to test if my paradigm can learn a non-intuitive game theory problem based on complex number theory (Zeckendorf's representation). It moves beyond simple pattern matching and requires the model to understand deeper mathematical structures.
generate_longest_subsequence_constrained.py: Tests the model's ability to handle a

complex optimization problem that mixes sequence operations and numerical constraints.
generate_treasure_hunt_tsp.py: Solves a complex state-space search problem that combines graph traversal (BFS) and combinatorial optimization (state compression DP), a classic problem in competitive programming.
generate_freedom_trail_dp.py: Tests the model's ability to learn to solve a complex optimization problem requiring dynamic programming and path backtracking.
generate_sum_of_subset_with_mask.py: Tests the model's ability to select elements from a set based on a binary mask and perform an aggregation operation (summation).
generate_sudoku_6x6.py: Tests the model's ability on a strong Constraint Satisfaction Problem—Sudoku.
generate_valid_parentheses_path_random_deprecated.py: An early attempt to solve the "valid parentheses path" problem. (Early exploration/deprecated)
generate_valid_parentheses_path_balanced.py: Solves a pathfinding problem on a 2D grid where the validity of the path is constrained by a stack-like structure (parentheses matching).
generate_sat_solver_text.py: Tests the model's ability to solve an iconic NP-complete problem—the Boolean Satisfiability (SAT) problem.
generate_sat_solver_compact_text.py: A variant of generate_sat_solver_text.py, using a different input encoding format to solve the same 3-SAT problem.
generate_point_in_polygon.py: Tests the model's ability to learn a classic algorithm in computational geometry—the Ray Casting Algorithm.
generate_shortest_path_in_matrix_bfs.py: Tests the model's ability to find the shortest path in a 2D grid based on the classic Breadth-First Search (BFS) algorithm.
generate_sudoku_4x4_stepwise_deprecated.py: Aimed to test the model's "stepwise" reasoning ability. (Deprecated)
generate_tiling_problem_deprecated.py: Aimed to test the model's ability to solve a classic tiling optimization problem, which is NP-hard. (Deprecated)
generate_hanoi_tower_twisted_path_deprecated.py: This script intended to generate a "twisted path" dataset for the Tower of Hanoi problem. (Deprecated)
generate_checkers_jump_1d_v2.py: Solves the 1D checker-swapping planning problem, used to reveal the limitations of large language models on certain types of reasoning tasks.

C. Image to Symbol

generate_checkerboard_to_binary.py: A basic vision-to-symbol conversion task to test the model's ability to decode structured information from raw pixel data.
generate_line_angle_to_vector.py: Tests the model's ability to extract precise geometric information (angles) from an image, a more advanced visual reasoning task than simple checkerboard recognition.
generate_count_shapes_from_image.py: Tests the model's ability to perform multiple visual tasks simultaneously: object recognition (shape), attribute recognition (color), and counting (aggregation).
generate_maze_symbolic_to_image.py: Converts a symbolic maze path planning dataset into an image format to test the ability of visual models (like CNN, ViT) to perform path planning directly from pixels.

generate_sokoban_symbolic_to_image_no_labels.py: A data conversion script that only converts a symbolic Sokoban dataset (.jsonl format) into image format, for purely visual tasks or as an intermediate step for more complex data processing.
generate_sokoban_symbolic_to_image_with_labels.py: A data conversion script that converts a symbolic Sokoban dataset (.jsonl format) into a complete image classification dataset for training computer vision models (like ViT, Swin Transformer).
generate_cellular_automata_image_and_label.py: A universal dataset generator. Generates data for cellular automata (CA) tasks in both image format (Img2Img) and symbolic format (Img2Label), with support for multiprocessing acceleration.
generate_trapping_rain_water_image_to_symbol.py: A specialized dataset generator. Generates image format data for the "trapping rain water" problem, where the input is a grid map of bar heights and the output is a symbolic label for the water amounts.

D. Image to Image

generate_triangle_to_incircle.py: A landmark experiment demonstrating "sculpting precise rules with gradient descent." It tests if the model can learn a purely non-trivial geometric construction rule (the incircle of a triangle).
generate_polygon_to_symmetry_axis.py: Tests the model's ability to infer the implicit axis of symmetry from a complete symmetrical figure.
generate_triangle_to_centroid.py: Tests the model's ability to learn another fundamental geometric concept—the centroid.
generate_triangle_to_tessellation.py: A landmark demonstration of my paradigm's capabilities. It tests if the model can learn an infinite, lattice-based generation rule. Due to the global correlations and precise details of the tessellation pattern, it strongly rules out the possibility that the model is solving the problem merely by "interpolation" or "memorization."
generate_game_of_life_image_to_image.py: The image-to-image version of the 2D cellular automaton, testing if the model can perform local rule-based evolution directly in pixel space.
generate_projectile_motion_simulation.py: Tests the model's ability to learn a simple dynamic physical process. This requires the model to infer the entire spatio-temporal trajectory from initial conditions (position and velocity vector).
generate_snell_refraction_simulation.py: Tests the model's ability to learn a fundamental law of physics (Snell's Law of refraction).
generate_snell_refraction_with_contextual_index.py: Tests the model's ability to learn a fundamental law of physics (Snell's Law), and requires the model to infer physical parameters (refractive index) from contextual information in the image (background color).
generate_cellular_automata_spatial_conditional.py: Tests the model's ability to partition and parse "instructions" and "data" within a single modality (image), a form of "pseudo-multimodal" or "spatial conditional" experiment.
generate_trapping_rain_water_visualizer.py: A data conversion and visualization script. It converts an existing, symbolic "trapping rain water" dataset into an image-to-image format dataset, so the same problem can be solved with a visual model.
generate_shortest_path_in_tree_deprecated.py: An early experiment aimed at testing the model's ability to find the shortest path in a graph from an image. (Early

exploration/deprecated)
generate_shortest_distance_between_triangles.py: Tests the model's ability to perform global geometric relationship (shortest distance) reasoning in a scene with multiple objects.
generate_reaction_diffusion_deprecated.py: This script was used to simulate a reaction-diffusion system to generate complex, fractal-like "snowflake" patterns. (Exploratory/deprecated)
generate_cellular_automata_multimodal_deprecated.py: Generates a truly multimodal dataset for training models that can understand both image inputs and text instructions simultaneously. (Deprecated)

E. Text to Image

generate_coords_to_triangle.py: A basic symbol-to-geometry rendering task, testing the model's ability to convert abstract coordinate information into concrete pixel shapes.
generate_cellular_automata_1d_to_grid_image.py: Tests if the model can directly "render" 1D symbolic computation results into a structured 2D image.
generate_triangle_coords_to_tessellation.py: An advanced reasoning task that mixes symbolic instructions with geometric generation rules.
generate_cube_rotation_matplotlib_deprecated.py: Aims to test the model's ability to infer and render the correct view of a 3D object from abstract pose parameters (rotation angles). (Early exploration version)
generate_cube_rotation_pillow_v1.py: Aims to test the model's ability to infer and render the correct view of a 3D object from abstract pose parameters, using a more low-level, technically precise rendering pipeline. (Technical upgrade version)
generate_cube_rotation_pillow_with_anchor.py: Tests the model's ability to infer and render the correct view of a 3D object from abstract pose parameters, using "visual anchors" to assist the model's learning. (Final version used in the paper)
generate_cube_rotation_pillow_wireframe.py: Tests if the model can learn 3D rotation from sparser visual input, using only wireframe and anchor point information. (Variant experiment version)

F. Physics Simulation Image Paradigm

generate_catenary_curve_simulation_deprecated.py: My early script for exploring the catenary problem, aimed at testing the model's ability to learn non-linear curves determined by physical laws.
generate_catenary_curve_from_points.py: Tests the model's ability to learn a non-linear curve (catenary) uniquely determined by physical laws (principle of minimum potential energy).
generate_orbital_path_from_initial_state.py: Tests the model's ability to learn more complex physical laws (Kepler's Laws / Law of Universal Gravitation).

G. ARC-AGI Exploration

generate_arc_contextual_color_swap.py: Tests the model's ability to learn a rule from a local "context" or "example" within an image and apply it to the global data of the same image. This directly mimics the core philosophy of the ARC-AGI test.
generate_arc_find_cross_pattern.py: Tests the model's ability to perform visual pattern recognition (or "object detection") in the presence of significant noise.
generate_arc_find_odd_one_out.py: Tests the model's ability to perform a complex "Find the Odd One Out" meta-reasoning task. The model needs to perform row-by-row pattern comparison, identify the exception, and reassemble it in the output.
generate_arc_connect_colored_pairs.py: Tests the model's ability to identify multiple independent "connection tasks" within the same image and understand an implicit "layering" or "drawing priority" rule.
generate_arc_conditional_perpendicular_lines.py: Tests the model's ability to perform different geometric operations based on object properties (color) and global references (boundary lines, image edges).
generate_arc_column_projection.py: Tests the model's ability to recognize complex contextual relationships ("below... and within the range of...") and perform conditional column operations.
generate_arc_procedural_spiral.py: Tests the model's ability to execute an iterative, procedural generation algorithm. The model needs to understand instructions, track state (current position, direction, length), and execute in a loop.
generate_arc_fractal_stamping.py: Tests the model's ability to understand and execute recursive or fractal generation rules. The model needs to use the input pattern itself as a "brush" and repeat the drawing according to "instructions" within the input pattern.
generate_arc_flood_fill.py: Tests the model's ability to execute the classic "Flood Fill" or "paint bucket" algorithm.
generate_arc_layered_fill.py: Tests the model's ability to understand a highly procedural, complex filling algorithm that depends on topological distance and conditional checks.
generate_arc_fluid_simulation.py: Tests the model's ability to learn and simulate a fluid dynamic process with specific rules in the image space.
generate_arc_periodic_conditional_fill.py: Tests the model's ability to learn a complex conditional formatting rule with periodicity and special cases.
generate_arc_fill_square_holes.py: Tests the model's multi-step visual reasoning ability: first identify "holes," then determine their geometric properties (are they square?), and finally perform an action based on the result.
generate_arc_conditional_recoloring.py: Tests the model's ability to understand visual layers and perform conditional modification of object properties.
generate_arc_sort_by_length_remap_position.py: Tests the model's ability to perform a complex sorting task involving "attribute-position decoupling and remapping."
generate_arc_jigsaw_puzzle_simple.py: Tests the model's ability to solve a visual matching and transformation problem (early version), where the size of the puzzle pieces is unique and can be used as a matching shortcut.
generate_arc_jigsaw_puzzle_advanced.py: Tests the model's ability to solve a complex visual matching and transformation problem. This version requires the model to genuinely

match based on shape (not size).
generate_arc_connect_path_by_sequence.py: Tests the model's ability to parse an external sequence of instructions and execute a multi-step, stateful path connection task in the image accordingly.
generate_arc_reflection_simulation_deprecated.py: Aimed to test the model's understanding of complex physics-based optical rules, including ray emission, collision detection, angle of reflection, and color transformation. (Deprecated)

H. Inverse Rule Inference

generate_cellular_automata_inverse_rule.py: This experiment is the first attempt to test the model's inverse reasoning ability. My question was: if a model can deduce the result from a rule, can it infer the underlying rule from "input-output" pairs?
generate_cellular_automata_inverse_rule_and_steps.py: An early version before the implementation of the "unique solution" version, which also aimed to have the model learn to predict the rule and the number of iterations.
generate_cellular_automata_inverse_rule_and_steps_unique.py: A major upgrade to the inverse reasoning task. I not only require the model to infer which rule was applied, but also how many times it was applied.

Acknowledgements

First and foremost, I sincerely thank my family. Without their unwavering support and understanding during the past several months of intense experimentation and paper writing, this work would not have been possible.

At the same time, the smooth progress of this research has greatly benefited from the powerful assistance provided by cutting-edge large language models. Throughout the experimental phase, from gpt-4o to the later gemini-2.5-pro, these models played an indispensable role as collaborative partners in writing dataset generation scripts, discussing experimental ideas, and even drafting the initial manuscript. They significantly accelerated the research iteration process.

In particular, I need to thank the DeepMind team for their work [17] Amortized Planning with Large-Scale Transformers: A Case Study on Chess. That research inspired me to shift the model's output paradigm from autoregressive generation to holistic classification. This shift was a key breakthrough in unlocking the model's ability to generalize within vast problem spaces and directly prompted me to begin a systematic exploration of the symbolic learning capabilities of neural networks.

Finally, due to time constraints, some calculations and details in this paper may contain omissions or errors. I would be very grateful for any criticism and corrections from readers.

A Note on Anonymity

For personal reasons, I have chosen to publish this work anonymously for the time being. While I am deeply passionate about sharing these ideas with the community, I am not fully prepared for the public attention that a project of this nature might attract. My hope is that this work can be judged on its own merits, independent of its author.

The possibility of revealing my identity in the future remains open. Until then, I am grateful for your understanding and respect for my decision. I look forward to engaging with your thoughts and feedback on the work itself.

References

1. Krizhevsky A, Sutsever I, Hinton G E. ImageNet classification with deep convolutional neural networks[C]//Proc. of the 25th International Conference on Neural Information Processing Systems. Lake Tahoe, NV: Curran Associates, 2012: 1097-1105.
2. Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[C]//Proc. of the 31st International Conference on Neural Information Processing Systems. Long Beach, CA: Curran Associates, 2017: 5998-6008.
3. Dosovitskiy A, Beyer L, Kolesnikov A, et al. An image is worth 16×16 words: Transformers for image recognition at scale[C]//Proc. of the 9th International Conference on Learning Representations. 2021.
4. Liu Z, Lin Y, Cao Y, et al. Swin Transformer: Hierarchical vision transformer using shifted windows[C]//Proc. of the IEEE/CVF International Conference on Computer Vision. Montreal, QC: IEEE, 2021: 9992-10002.
5. Liu Z, Mao H, Wu C Y, et al. A ConvNet for the 2020s[C]//Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. New Orleans, LA: IEEE, 2022: 11976-11986.
6. Graves A, Wayne G, Danihelka I. Neural Turing machines[J]. arXiv preprint arXiv:1410.5401, 2014.
7. Graves A, Wayne G, Reynolds M, et al. Hybrid computing using a neural network with dynamic external memory[J]. Nature, 2016, 538(7626): 471-476.
8. Rocktäschel T, Riedel S. End-to-end differentiable proving[C]//Proc. of the NIPS Workshop on Advances in Approximate Bayesian Inference. Long Beach, CA, 2017.
9. Manhaeve R, Dumancic S, Kimmig A, et al. DeepProbLog: Neural probabilistic logic programming[C]//Proc. of the 32nd International Conference on Neural Information Processing Systems. Montréal, QC: Curran Associates, 2018: 3749-3759.
10. Battaglia P W, Hamrick J B, Bapst V, et al. Relational inductive biases, deep learning, and graph networks[J]. arXiv preprint arXiv:1806.01261, 2018.
11. Andreas J, Rohrbach M, Darrell T, et al. Neural module networks[C]//Proc. of the IEEE Conference on Computer Vision and Pattern Recognition. Las Vegas, NV: IEEE, 2016:

39-48.
12. Chollet F. On the measure of intelligence[J]. arXiv preprint arXiv:1911.01547, 2019.
13. Moskvichev A, Odouard V V, Mitchell M. The ARC-AGI-2 benchmark: Evaluating advanced reasoning and conceptual generalization[J]. arXiv preprint arXiv:2505.11831, 2025.
14. ARC Prize. ARC-AGI-3: Interactive reasoning benchmark (preview)[EB/OL]. (2025-08-22)[2025-08-28]. https://arcprize.org/arc-agi/3/.
15. Moskvichev A, Odouard V V, Mitchell M. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity[J]. arXiv preprint arXiv:2506.06941, 2025.
16. Wolfram S. A new kind of science[M]. Champaign, IL: Wolfram Media, 2002: 1-1197. ISBN 1-57955-008-8.
17. Ruoss A, Deletang G, Genewein T, et al. Amortized planning with large-scale transformers: A case study on chess[J]. arXiv preprint arXiv:2402.04494, 2024.
18. Lowe D G. Distinctive image features from scale-invariant keypoints[J]. International Journal of Computer Vision, 2004, 60(2): 91-110. DOI:10.1023/B:VISI.0000029664.99615.94.
19. Dalal N, Triggs B. Histograms of oriented gradients for human detection[C]//Proc. of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. San Diego, CA: IEEE, 2005: 886-893. DOI:10.1109/CVPR.2005.177.
20. LeetCode. Trapping rain water[EB/OL]. [2025-08-28]. https://leetcode.com/problems/trapping-rain-water/description/.
21. Yang A, Yang B, Hui B, et al. Qwen2 technical report[J]. arXiv preprint arXiv:2407.10671, 2024.
22. Karpathy A. Software 2.0[EB/OL]. (2017-11-11)[2025-08-28]. https://karpathy.medium.com/software-2-0-a64152b37c35.