Introduction

Note: This report assumes some familiarity with ARC-AGI (their website is the best place for a quick intro, this technical report is the best place for a deep dive).

I followed ARC-AGI 2024 closely and was impressed by the incredible results that teams achieved, but also surprised that there were not many approaches that had a firm anchor to ground truth. The top approaches were largely very clever ways to get the most out of transformer based models, but ultimately accepted the output of the LLM at its word. This made me feel uneasy. While the empirical results were undeniable, not knowing quite how the model got to its predictions (as a search tree would show) or that it was anchored in reality in some way (as a well-designed reinforcement learning environment would establish) was deeply unsettling.

I attempted to address this by building an approach to solving ARC-AGI that was more grounded in reality: a custom Group Relative Policy Optimization (GRPO) implementation inspired by Deepseek R1 with a few modifications suited to ARC. This approach achieves 51% on a random subset of 100 evaluation tasks from the ARC-AGI public evaluation dataset. I'd like to talk about this approach and work through how I got here, focusing on the mistakes I made along the way. Finally, I'd like to explore what's next, specifically, why I'd like to move farther away from raw LLM-based reasoning, despite it working well for me, and closer toward searching the solution space using an RL-trained policy.

Approach

The motivation for my approach came from my experience as an intermediate chess player, and framing my play in terms of the System 1/System 2 analogy defined in Kahneman's book. I first subconsciously intuit a move (most times the move seems to appear out of thin air), and then perform conscious calculations on lines from that move. Similarly, I wanted my approach to begin with a guess from a finetuned model with good intuition, and then perform concrete modifications grounded in reality to steer that guess to a final prediction.

Method

1. Initial finetuning on Mistral-NeMo-Minitron-8B-Base using the ARChitects' script, making a few changes to exclude the ARC-AGI evaluation set from the training data (of course, it is still possible that the eval set solutions leaked to the base model even though it wasn't finetuned on it, so it's important to look at the improvement delivered by RL over the baseline finetuned model). This step uses the Re-ARC dataset (excluding the public eval set), the ARC prize training set, and the ConceptARC dataset. I've shared the initial finetuned model to Huggingface here.
2. Second round of finetuning on augmented versions of the demonstration pairs in the eval set, again using the test-time finetuning code shared by the ARChitects (I've shared the LoRA model for this step here). The motivation for using a finetuned model instead of applying RL to the base model directly was to spend more time actually refining the prediction rather than figuring out the correct format.

3. Then, for each puzzle, create 128 augmented leave-one-out tasks from the demonstration pairs, i.e., a task where given an augmented input and output for x, and the augmented input for y, predict the augmented output for y. The augmentations used were reflections, transpositions, rotations and color permutations, pretty standard in several ARC approaches.
4. Run the main training pipeline for each puzzle:
    a. Generate 32 completions for each prompt using the initial policy.
    b. Assign an advantage to each completion based on how good that completion was relative to the others in its batch (reward described in Reward function), how likely the current policy was to come up with this completion relative to the initial policy, and a KL divergence penalty for straying too far from the original distribution.
    c. Track an average reward for training batches, so the model can choose to move on to the next problem early if it is confident in its current prediction (if the average reward crosses a threshold).

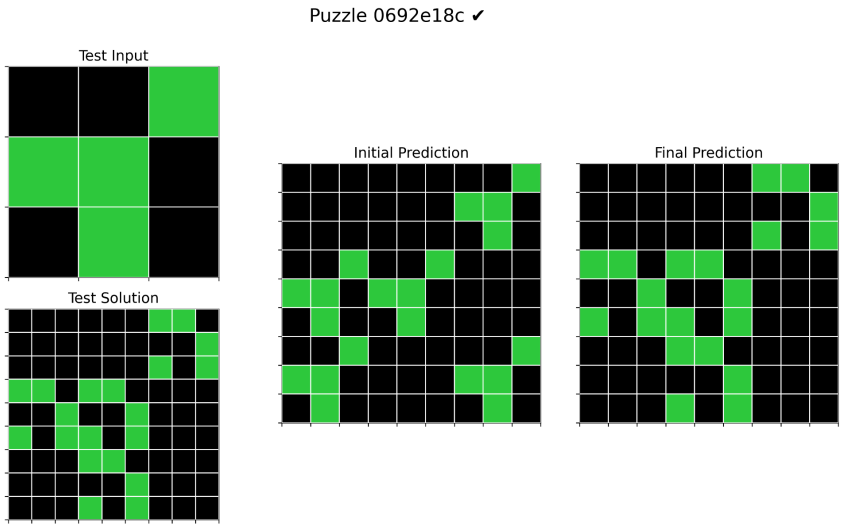
Reward function

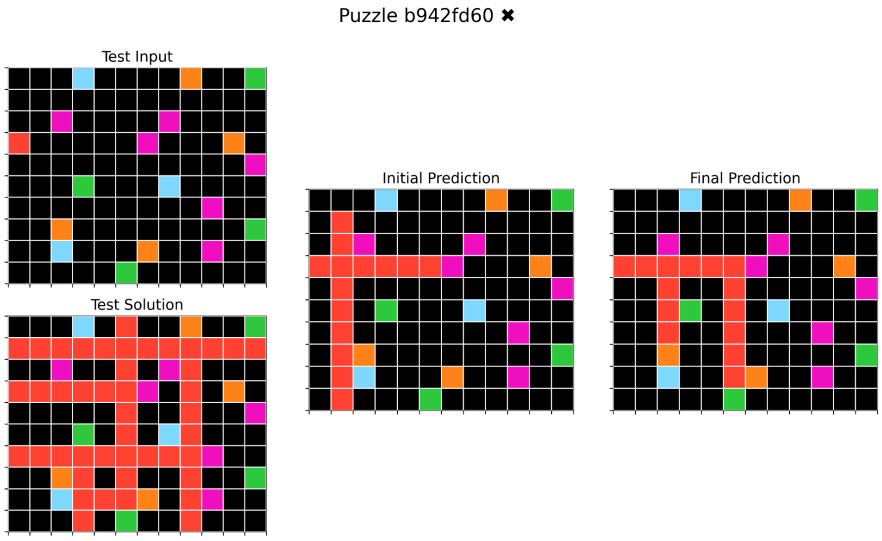
The reward function is a simple Hamming distance, with a bonus of +5 for a perfect answer, and penalties of -2 and -3 for incorrect shapes and invalid outputs respectively. For puzzles with several test inputs, the scores are averaged among them.

Results

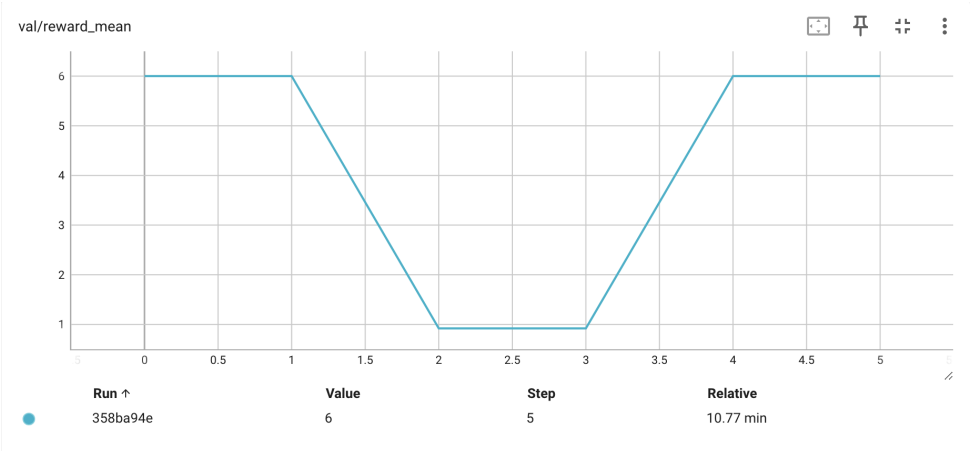
The model ultimately scored 51% on a single attempt for a random sample of 100 puzzles from the public evaluation set (I wanted to test on all 400, but was constrained by compute), and applying the GRPO step specifically solved 8 problems above the 43 problems that finetuning alone was able to solve. There is a lot of variance from puzzle to puzzle, but on average applying GRPO improves the puzzle score by 0.45 (scoring described in Reward function). A complete table of results is in this Google sheet. In 2 cases it figured out a correct solution but went back to its initial incorrect guess, but in 0 cases did applying RL take the model from the correct initial solution to an incorrect final prediction.

Below are problems the model attempted, with the initial predictions from the finetuned models. RL solves puzzle 0692e18c where the finetuned model alone fails, but is unable to solve puzzle b942fd60.





The most interesting ones to me are cases like [#358ba94e] below, where it actually starts off at the correct solution (receiving a perfect score of 6.0), makes an incorrect modification, but changes its mind quickly and goes right back to the correct solution. Confident at this point, the model even returns early.



I'd love to perform a thorough set of experiments changing some of the generation parameters, and study the impact of train times on results to test this concretely, but in the few parameters I tried, I was pleased to see predictable and stable loss curves, flat or improving rewards, consistent gradient norms and found that the results were robust to small changes in hyperparameters and training time.

Conclusion

So is this a win? It depends. On the one hand, although this approach isn't exactly comparable to most others since it does not train on any solutions of the public eval set, and does not test on the private eval set, the model does achieve solid results, scoring 51%, with RL granting it an improvement of 18.6% over a finetuned model that is arguably already good.

On the other hand, on a spectrum from pure supervised learning to pure RL, this approach to me sits closer to the supervised learning end. It incorporates real, ground truth feedback, but only in the final step to grade guesses that the LLM generates, not through a systematic search of the solution space guided by rewards from its environment. This means that, first, although it has a degree of grounding in

reality, it feels very marginal, and second, it is still largely not explainable. These characteristics were present in my previous approaches, but I could not get them to perform well due to some flaws that I'd like to discuss next.

Attempts along the way

Monte Carlo Tree Search

My first attempt was a fundamental 'pure RL' approach, and I quickly learned why winning submissions to the 2024 ARC-AGI challenge did not use pure RL. This approach recreated an AlphaGo style MCTS with an exhaustive moveset (color a cell, color a rectangle, rotation, color permutation, reshape and reflection), a partial reward for each move based on the Hamming distance between the current state and the solution, and a big bonus for reaching the correct solution, or the terminal state.

Each simulation would start from an input grid, explore the search space and receive rewards for how well each 'program' did (thinking of each node in our search tree as the state and each edge in our search tree as the next operation in our program), which would then bubble up the tree to incentivize successful operations, tempered by an exploration bonus so it wouldn't tunnel vision into a single decent node without sufficiently exploring the rest.

The result was pure noise. The search space was just too large, and a few hours on a single H100 was woefully insufficient to get it to converge. I wasn't entirely surprised by this: after all, the number of legal moves from any position in Go is far smaller than the number of legal moves I had defined in this ARC-solving game, and even then solving Go wasn't possible without brilliant engineering and several orders of magnitude more compute.

My next idea was to shrink the action space, and build in a synergy between my action space and my reward function. Right now, the two were not in sync: Since it just looked at the number of matching cells, it was possible for my algorithm to hand out no reward in a state only one rotation away from a correct solution. I decided to simplify my action space drastically to just one action: coloring a single cell. This would then perfectly complement my reward function which graded on cell-wise similarity. I backed my RL policy with a ~40M param network and ran hundreds of simulations per training example, expecting it to perform at least a bit better than earlier, but to my disappointment, it did not. The search space was just far too deep.

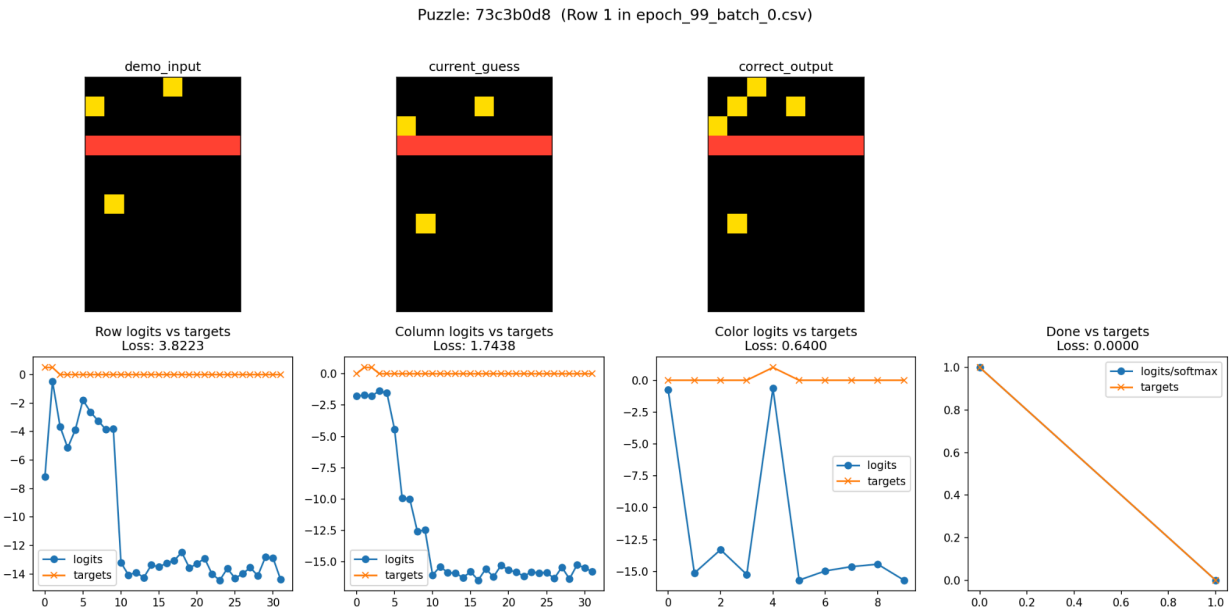
Error correction

This led me to my next idea: if the solution was too many moves away, what if I just started from closer? I finetuned a new model to generate a 'first pass' prediction, and a second model would learn to make moves (again, through a policy learned from MCTS explorations) that guided it from the initial prediction to the solution. Now that the solution was only a handful of moves away, it should work, right?

Here's where I learned my next lesson: the model got pretty good at learning the error distribution of a given demonstration pair it was trained on, but that did not, by any means, generalize to the test pair.

This approach implicitly assumed that all errors from the finetuned model followed a fixed, learnable distribution for a given puzzle across all examples, and that was simply not true.

The model reliably learned how to pick colors that appeared in other demo outputs and not draw out of bounds of the grid, which was sometimes enough to make meaningful moves, as shown below, but this was far from a general reasoning system capable of solving ARC-AGI.



Here you can see that the model correctly learned that to get puzzle 73c3b0d8 from the current prediction (top row, middle) to the solution (top row, right), reading along the peaks of the logit plots, it would need to color a cell somewhere in the first few rows and columns yellow (4 represents yellow).

But there is also a deeper reason this approach failed. Although my intuition that solving this would require applying both System 1 and System 2 reasoning was not incorrect, I was applying System 2, the error correction mechanism, in isolation from the first. Tunneling on just the errors between the initial prediction and the solution completely ignored the wealth of meaning present in the relationship between the demonstration input and its solution. In chess, the System 2-like calculation of concrete lines happens with awareness of all the data available to System 1, and my approach failed to account for that. It was obvious in hindsight, but System 2 reasoning does not happen in a separate context from System 1 reasoning. The learnings from this led me to the final approach covered above.

Future work

There are two avenues I could take next: invest more into the LLM based approach, or revisit approaches more similar in philosophy to the ones in Attempts along the way.

The ideas to improve my LLM and GRPO model seem limitless—including using a granular advantage function that rewards tokens based on their contribution to the final reward, using a single adapter for all

problems to learn patterns helpful across puzzles, improving the early stopping function to address its high false positive rate while still keeping it efficient—but these feel like incremental improvements to an approach that is still fundamentally not explainable and not deeply rooted in reality. No amount of scaling or tweaking can fix this.

The second avenue, revisiting approaches using an RL-trained policy to guide a search algorithm, is far more exciting. My previous approaches here were flawed, but I'd like to learn from those mistakes and build new models to achieve these goals. I'm not sure exactly what shape this will take, but my next step will be to explore segmenting the search space, which in the limit means pruning sections that are unlikely to bear fruit, and incentivizing learning in more promising areas of the search space. Although unlikely to have results comparable to the first avenue in the short-run, I believe that the systems of the future will reason in explainable ways anchored to reality, and I'd like to help build those systems.