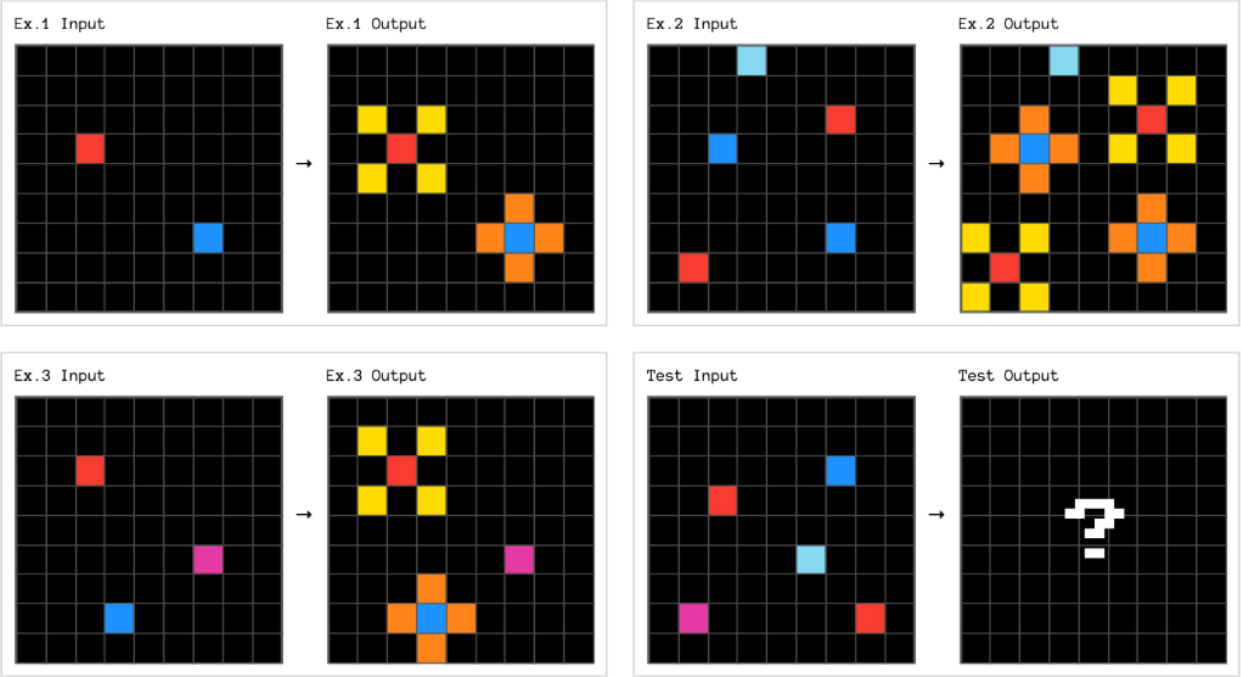
My ARC Challenge Africa Solution: A Personal Walkthrough



When I first joined the ARC Challenge, I knew this wasn't just any machine learning competition. Each task was more like a puzzle—like the kind that test not just intelligence but flexibility. The goal was to design a system that can observe examples, identify hidden patterns, and apply the same reasoning to new unseen inputs. No training. No large datasets. Just pure reasoning.

My final solution is based on this very idea: if humans can learn from a few examples, so should a good enough program. I didn't train a model. Instead, I built a hybrid logic-based pipeline that tries to decode what's happening in each task. Here's how it works.

Step 1: Getting Familiar with the Task Format

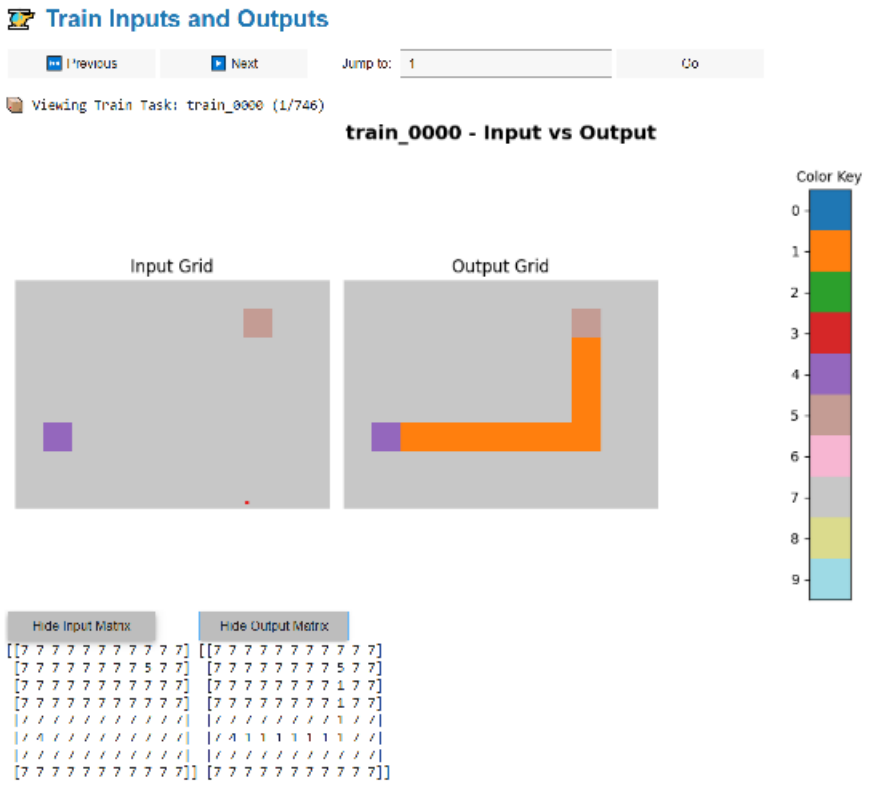
Every ARC task gives one or more input/output examples for training. Each is a grid of numbers (0–9), where each number stands for a color. The test input is also a grid, and the job is to produce the correct output grid using the same logic the training pairs followed.

So my first step was to read the task files from JSON and visualize them. But I didn't just display a single image — I created an interactive viewer. This tool allowed me to move forward or backward through the training tasks, and for each one, I could see:

- The input grid

- The output grid
- How the transformation looked visually

I also made it possible to jump to a specific task by ID, which made it easier to analyze one problem at a time. This helped a lot in understanding what kind of logic each task followed. Sometimes it was just color changes, sometimes it involved flipping, and other times it was about size or shape.



Step 2: Inferring the Output Grid Size

One tricky part is that test inputs don't tell you what size the output should be. But the sample submission file gives us a hint — the number of rows × columns for each predicted output.

So I wrote a small function to extract the shape of the output grid from the submission file. That way, my final predictions would always be the right shape.

Step 3: Detecting Simple Color Mappings

The first real logic check my code performs is this:

> Does the output just change colors in a consistent way?

Let's say in the training input, every 2 becomes a 3, and every 5 becomes a 0. That's a one-to-one color mapping. So I scan all training pairs and check if a consistent color map exists.

If I find such a mapping, I apply the same rule to the test input. This handles a surprising number of tasks.

Step 4: Trying Geometric Transformations

Next, if color remapping didn't solve the task, I move to geometry. I try out:

- Flipping (horizontal and vertical)
- Rotating (90°, 180°, etc.)
- Transposing
- Cropping or padding

Each transformation is applied to the test input, and then I compare the result to the training output. If it matches well (measured by how many cell values match), I take it.

I built a set of named transformations so I could try them in sequence, like:

```
TRANSFORMATIONS = {
    'flip_horizontal': lambda g: np.flip(g, axis=1),
    'flip_vertical': lambda g: np.flip(g, axis=0),
    'rotate_90': lambda g: np.rot90(g),
    'rotate_180': lambda g: np.rot90(g, 2),
    'rotate_270': lambda g: np.rot90(g, 3),
    'invert_colors': lambda g: 10 - g,  # Assuming 10 colors max
    'shift_right': lambda g: np.roll(g, 1, axis=1),
    'shift_down': lambda g: np.roll(g, 1, axis=0),
    'dilate': lambda g: np.kron(g, np.ones((2,2), dtype=int)),
    'erode': lambda g: g[::2, ::2],
    'border_extract': lambda g: g - np.pad(g[1:-1,1:-1], ((1,1),(1,1)), mode='constant', constant_values=0),
    'color_swap': lambda g, mapping: np.vectorize(lambda x: mapping.get(x, x))(g)
}
```

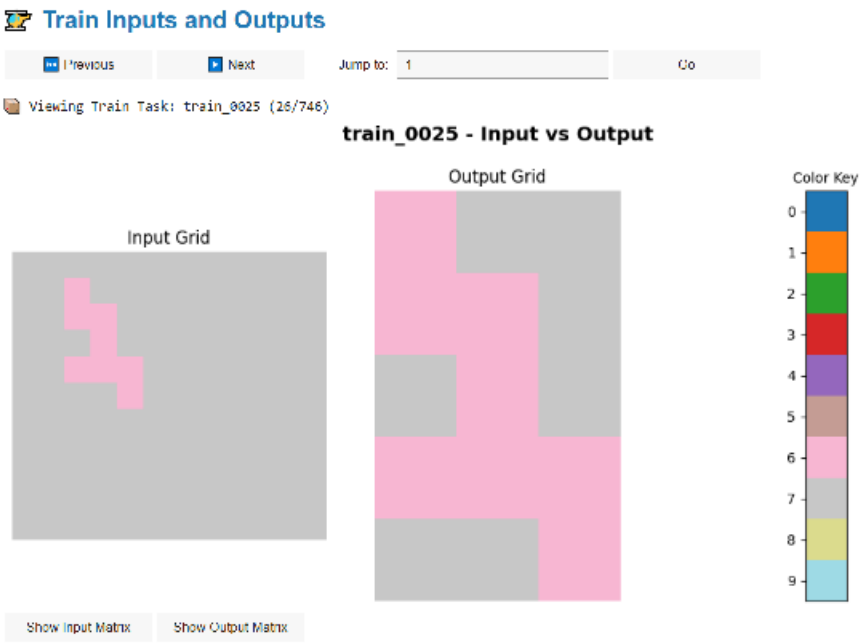
This made testing different strategies much easier.

Step 5: Looking at Shape-Based Changes

Sometimes the logic isn't about flipping or color — it's about how big things are. For example:

- A shape might double in size.
- The object might get thicker.
- The background might be removed.

To handle that, I added support for dilation (growing shapes), erosion (shrinking), and even simple outline detection.



These kinds of transformations helped cover shape-centric tasks that couldn't be solved with just rotation or flipping.

Step 6: Fallback Strategies

If all logic-based strategies fail (and that does happen), I have a few last-resort tricks:

1. Tiling: Repeat the input pattern until it fills the target output grid.
2. Color shift: Just shift all colors by a constant (e.g., add 5 modulo 10).
3. Border padding: Surround the object with a background color.

These aren't guaranteed to be correct, but they're better than empty guesses.

Step 7: Resizing and Saving Predictions

Regardless of how the output is produced, I always resize it to the required shape using nearest-neighbor interpolation. This makes sure the submission file matches Zindi's expected format.

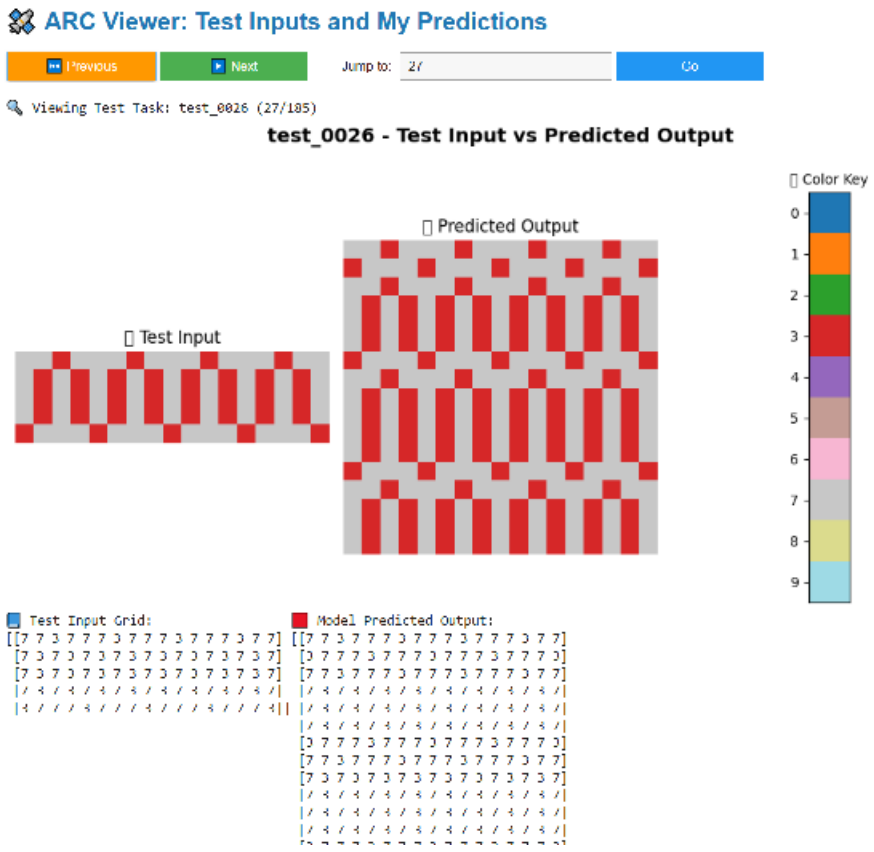
Finally, I flatten the grid row-by-row and save it to CSV. This output is named `arc_agi_final_submission.csv`.

Visualizing My Predictions

Once the system was working, I used a second interactive viewer to explore my predictions. This one displayed:

- The test input
- The output generated by my code

Like before, I could move through tasks or jump to any task ID. This helped me quickly spot mistakes or successes, and compare them visually. It gave me more confidence that my logic was actually doing what I expected.



How to Use My Code

To run the full pipeline:

1. Install the libraries in `requirements.txt`
2. Place `train.json`, `test.json`, and `SampleSubmission.csv` in the same folder
3. Open the notebook (or script)
4. Run all cells
5. The final CSV will be saved as `arc_agi_final_submission.csv`

I also added progress bars and optional visualization tools for inspection.

Real-World Applications

The same pattern-finding logic used here can apply to:

- Robotics: Where a robot learns how to transform environments based on prior interactions
- Design automation: Automatically resizing or recoloring templates
- Rule inference in low-data situations, where you can't train a model but still need generalization

---

Challenges and Weaknesses

I'll be honest: my solution isn't perfect. Some puzzles require combining multiple rules, and mine can usually only apply one dominant pattern at a time. It also struggles when a task requires counting objects, or when output logic depends on subtle pixel relationships.

I tried to counter this with a validation function that scores how well each candidate output matches the training pattern — but it's still heuristic. In a future version, I'd consider adding symbolic reasoning or more robust object detection.

Conclusion

This project taught me a lot — not just about solving ARC tasks, but about building systems that think step by step. While my model isn't the most complex, it's explainable, fast, and reproducible. I can trace every prediction back to a specific transformation or rule.

And that's what makes me proud of this solution.

Works Cited

Chollet, François, et al. *The ARC-AGI Benchmark: Towards General Intelligence through Abstraction and Reasoning*. arXiv preprint arXiv:2412.04604v1, 2024. https://arxiv.org/abs/2412.04604.

Zindi. "How to Ensure Success When Submitting Your Code." *Zindi Learn*, https://zindi.africa/learn/how-to-ensure-success-when-submitting-your-code-for-review.

Zindi. "Documentation Guidelines." *Zindi Learn*, https://zindi.africa/learn/documentation-guideline.