

Appendix A. Search algorithm pseudo-code

Algorithm 1: TreeSearch

Input: X , input grids
Input: Y , target grids
Input: M , the neural network model
Parameter: τ , the time budget
Parameter: ϵ , the maximum tree depth
Output: The solution, if found.

```

1: // Instantiate the root node
2:  $root = selected = SearchTreeNode(X, None)$ 
3:  $start\_time = time()$ 
4: while  $time() - start\_time < \tau$  do
5:    $output = execute\_program(selected)$ 
6:   if  $output == Y$  then
7:     // Solution found!
8:     return  $selected$ 
9:   end if
10:   $progs = enumerate(M, Y, selected)$ 
11:  for all  $prog \in progs$  do
12:     $log\_prob = calculate\_joint\_prob(prog)$ 
13:    Insert into program queue:  $(prob, prog)$ 
14:  end for
15:   $next\_best = Dequeue\ program\ queue$ 
16:   $leaf = program\ leaf\ node\ for\ next\_best$ 
17:   $child = SearchTreeNode(None, leaf)$ 
18:   $leaf.child = child$ 
19:   $selected = child$ 
20: end while
21: return  $None$ 

```

In the above pseudocode, *SearchTreeNode* is the class representing a node in the tree structure. The first argument is the state variable if we already have it, and the second argument is the parent node. The node’s state variable can be set to *None* and later updated from inside the *execute_program* call. The latter executes the instruction step associated with the node, gets the resulting output, and updates the node’s internal state variable. It also returns the result to verify if the goal was reached.

The *enumerate* function aggregates encoder output from the parent node states and the target grid Y , and then uses the model M to recursively query token probabilities and generate all non-zero probability token sequences for the next instruction step. It returns these new generated programs. Suppose *enumerate* generates 42 different possible instruction steps at this stage in the program, this means that we add 42 new programs to the program queue: each starting with the same instruction steps for the node parents trajectory, and ending with the newly generated instruction steps.

Appendix B. Hyperparameters

Execution-guided NPS The neural network used in the experiments presented in this paper is a standard Encoder-Decoder transformer architecture (using PyTorch), with 4 encoder and 4 decoder layers. The feedforward layer’s dimensionality is 1024, the embedding space (*d_{model}*) has a

dimension of 256, and 16 attention heads were used. The input vocabulary has a size of 55 and the target vocabulary has a size of 45.

TTFT Fine-tuning from scratch on our training data was not done using LoRA. Instead, it was a full, ”standard” training of the weights. While most of the hyperparameters used were the same ones used by Omni-ARC to reach 2nd place in the 2024 ARC-AGI competition, in an attempt to improve the performance of *TTFT-no-augments* we varied some hyperparameters in the test-time fine-tuning phase. In particular:

1. *max_steps*: we tried fine-tuning a different number of steps from 10 steps up to the maximum number of steps that took 2 minutes 30 seconds, to respect the 3-minute time budget (the subsequent LoRA merge step, inference step and voting steps took approximately 30 seconds). This led to trying up to about 60 fine-tuning steps.
2. *learning rate*: varied between $2e-4$ to $1e-5$ in various increments.
3. *batch_size*: we tried the default 16, 8 and 32.
4. *eval_steps*: we tried 10, 20, 50.

Appendix C. OOD Task details

The neural network is trained on generated training data that implements the following 14 simple ARC-AGI-like tasks, on grids of variable dimensions between 3×3 and 30×30 :

1. Horizontally flip the grid around the vertical axis formed by the center column.
2. Vertically flip the grid around the horizontal axis formed by the center row.
3. Set all non-zero, non-black pixels to the color green.
4. Horizontally flip the grid and set all its foreground pixels to green.
5. Vertically flip the grid and set all its foreground pixels to green.
6. Shift pixels to the right by 1 column, with no wrapping; the leftmost column is left completely black.
7. Shift pixels upward by 1 row (no wrapping).
8. Shift pixels downward by 1 row (no wrapping).
9. Shift pixels to the right by 1 column, and then horizontally flip the resulting grid around the central axis.
10. Vertically flip the grid, and then shift the resulting grid to the right by 1 column.
11. Horizontally flip the grid, and then shift the pixels upward by 1 row.
12. Shift the pixels downward by 1 row, and horizontally flip the grid.
13. Shift the pixels upward by 1 row, and then vertically flip the grid.
14. Vertically flip the grid, and then shift the pixels upward by 1 row.

The OOD tasks are:

1. Shift the pixels to the right by 1 column, and set all the non-zero pixels to green.
2. Horizontally flip the grid, and then shift the pixels to the right by 1 column.
3. Shift the pixels diagonally towards the upper-right corner by 1 cell.
4. Rotate 180 degrees
5. Horizontally flip the grid, shift it to the right, then vertically flip it.
6. Set the non-zero pixels to green, shift diagonally the pixels towards the upper-right corner.
7. Shift pixels to the left (no wrapping).

Appendix D. DSL details

Object attributes:

1. `.x`: a list of all x coordinates of each grid cell from left to right, top down
2. `.y`: a list of all y coordinates of the grid from left to right, top down
3. `.c`: a list of all pixel colors of the grid from left to right, top down
4. `.width`: the number of columns in the grid
5. `.height`: the number of rows in the grid
6. `.max_x`: width - 1 (essentially a shortcut to simplify code)
7. `.max_y`: height - 1
8. `.ul_x`: x coordinate of the upper left corner of this grid. Can be non-zero if this is a sub-grid representing an object in the outer grid.
9. `.ul_y`: y coordinate of the upper left corner of this grid. Can be non-zero if this is a sub-grid representing an object in the outer grid.