ARC-AGI WITHOUT PRETRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Conventional wisdom in the age of LLMs dictates that solving IQ-test-like visual puzzles from the ARC-AGI-1 benchmark requires capabilities derived from massive pretraining. To counter this, we introduce *CompressARC*, a 76K parameter model without any pretraining that solves 20% of evaluation puzzles by minimizing the description length (MDL) of the target puzzle purely during inference time. The MDL endows CompressARC with great generalization abilities - to our knowledge, CompressARC is the only deep learning method for ARC-AGI where training happens only on a fraction of one sample: the target inference puzzle itself, with the final solution information removed. Moreover, CompressARC does not train on the pre-provided ARC-AGI "training set". Under these extremely data-limited conditions, we do not ordinarily expect any puzzles to be solvable at all. Yet CompressARC still solves a diverse distribution of creative ARC-AGI puzzles, suggesting inference-time MDL to be a feasible alternative way to produce intelligence, besides conventional massive pretraining.

1 INTRODUCTION

The ARC-AGI-1 benchmark consists of abstract visual reasoning puzzles designed to evaluate a system's ability to rapidly acquire new skills from minimal input data. (Chollet, 2019) Recent progress in LLM-based reasoning has shown impressive skill acquisition capabilities, but these systems still rely on massive amounts of pretraining data. In this paper, we explore how little data is truly required to tackle ARC-AGI by introducing *CompressARC*, a solution method derived from the Minimum Description Length (MDL) principle. (Rissanen, 1978) CompressARC performs all of its learning at inference time and achieves 20% accuracy on ARC-AGI-1 evaluation puzzles—using only the puzzle being solved as inference-time training data.

The key to CompressARC's data efficiency is its formulation as a minimum description length (MDL) problem: predicting the unsolved grids by finding the shortest possible self-contained program that outputs the entire ARC-AGI dataset, including any unsolved grids filled arbitrarily. Historically, multiple theoretical frameworks have proven this program search to yield close-to-optimal predictions in diverse problem settings. (Hutter, 2000; 2005; Solomonoff, 1964; Rissanen, 1978) Intuitively, they show that by Occam's razor, the shortest program which outputs a matching replica of the given problem portion of a puzzle is expected to also default-output solutions that would "make sense" in their context, to save on program length. To implement this, a naïve program search would require exhaustively enumerating and executing many candidate programs, which is computationally infeasible. (Solomonoff, 1964; Hutter, 2005) CompressARC keeps the search feasible by only searching over a parametric subspace of programs involving neural networks (see Figure 1). This converts the combinatorial program-shortening search into a neural network optimization problem, so the search can be performed by gradient descent instead.

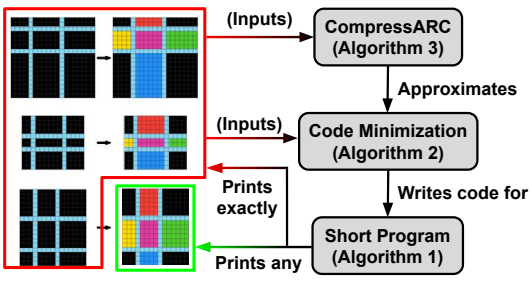


Figure 1: CompressARC approximates a specific description-length-minimizing algorithm, whose goal is to convert the ARC-AGI puzzle dataset into the shortest piece of code that prints it out exactly, with arbitrary puzzle solutions included. These printed solutions are assumed to be good predictors of the actual solutions, according to Occam's razor.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

This framing preserves several attractive properties of the original MDL formulation:

- **No pretraining:** Since we begin with the target puzzle(s) in hand, no training phase is required.
- **Inference-time learning:** Program length is minimized during inference by optimizing network weights with respect to the target puzzle(s).
- **Minimal data requirement:** The only data we need is the puzzle(s) themselves-no additional data is loaded into memory. Despite this, we achieve enough generalization to grok the solutions.

We emphasize that the contribution of CompressARC is not achieving state-of-the-art performance on ARC-AGI, but introducing a novel compression framework with surprising results and implications for generalization in the low-data regime. We give background on ARC-AGI and discuss how CompressARC compares to related ARC-AGI methods in Section 2, with other compression-related work in Appendix A. Our MDL-based framework will be introduced in Section 3, with architectural details in Section 4. We will then discuss results and implications in Sections 5 and 6.

2 Background: The ARC-AGI Benchmark

ARC-AGI-1 is a benchmark designed to test a system's ability to acquire new skills from minimal examples. (Chollet, 2019) Each puzzle in the benchmark consists of a different hidden rule, which the system must apply to an input grid of colors to produce a ground truth target grid. The hidden rules make use of themes like objectness, goal-directedness, numbers & counting, basic geometry, and topology. Several input-output grid pairs are given as examples to allow the system figure out the hidden rule in the puzzle, and no other information is given. Appendix K contains more details about the ARC-AGI-1 benchmark, which we generally refer to as ARC-AGI in this paper. Several classes of solutions to ARC-AGI have been studied, which we list in order of most-to-least data-hungry:

- The most prominent is a group of pretrained-LLM-based solutions which focus on reasoning capabilities (Chollet, 2024), harnesses, and/or model-controlled programmatic manipulation of ARC-AGI puzzles (Li et al., 2024a; Butt et al., 2024). Puzzle-specific fine-tuning at test time and test-time training have been shown to improve results (Akyürek et al., 2024). Meanwhile, closed-source LLMs which step outside of the official scoring environment's compute constraints have saturated the benchmark (Anthropic, 2026).
- Direct supervised learning on puzzles, using transformers usually with 2D positional embeddings and/or iterated neural networks (Li et al., 2024b; Hu et al., 2026; Vakde, 2025; Wang et al., 2025; Jolicoeur-Martineau, 2025), have been able to reach a solve rate of 60.2%. These are powered by large amounts of synthetic data created via hand-crafted per-training-puzzle data augmentations and custom data generators (Hodel, 2024; Moskvichev et al., 2023). (Bonnet & Macfarlane, 2024) introduced a VAE-based method for searching through a latent space of programs.
- An older class of methods consists of hard-coded searches through program spaces in hand-written domain-specific languages designed specifically for ARC (Hodel, 2024; Odouard, 2024; Ferré, 2025). More recent variants use neural networks to accelerate the search (Joffe & Eliasmith, 2025).

All methods we know which use neural networks must train the networks on the ARC-AGI training set beforehand, or rely on pretrained LLMs, unlike CompressARC which uses no data other than the test puzzle. All methods are also trained by direct supervised learning in some form, except for Bonnet & Macfarlane (2024) who optimize a VAE objective similar to CompressARC's objective. Several methods hold similarity to CompressARC by performing puzzle-specific gradient steps at test time, including Akyürek et al. (2024) and Bonnet & Macfarlane (2024).

3 Method

We frame ARC-AGI as a problem of finding the shortest possible program that reproduces the ARC-AGI dataset, entrusting that this program will output the correct answers. (Rissanen, 1978) We try to cast this problem to match the conditions where theoretical frameworks result in strong predictions: so the code must be entirely self-contained, receive no inputs, and must print out the entire ARC-AGI dataset of puzzles with any solutions filled in. Each puzzle takes the form of a tensor of shape $[\texttt{n_exmpl}, \texttt{width}, \texttt{height}, 2]$, containing color designations for every pixel in the

$2 \times$ n_exmpl grids. (The shapes listed in this section are for explanatory purposes and the actual data format is introduced in Section 4.) If a programmer were to attempt this by hand, a first-pass approach would be to hard-code the dataset as a giant string and print it out. This will indeed correctly print the ARC-AGI dataset's questions, but the program will be lengthy and it will print incorrect answers. As the programmer shortens the program by de-duplicating substrings and taking advantage of other structures, the answers produced by the program may gradually become more reasonable.

This problem setup seems amenable to discrete symbolic methods rather than neural networks. Instead, we try to bring the problem into the continuous domain, where gradient descent can be used to solve the problem. The following sections will re-frame the problem in a continuous way so that a small overfitted neural network can be inserted into a short program to act as a decent solution.

3.1 Restricting the Program Space for Search

Brute-force search for the shortest satisfying program is infeasible, because there are exponentially many programs whose outputs do not match our ARC-AGI dataset. Despite this, search can be made more amenable in a suitably well-conditioned subspace of programs. We chose a space consisting of a fixed template program (Algorithm 1) to be completed by substituting various hard-coded values into designated locations (shown in red), with different values producing different programs in the space. The idea of this template is to generate individual ARC-AGI puzzles using small overfitted neural networks, and any errors can be corrected by seed-manipulating to exert control over the input and output of the network. The fixed template program performs the operations for every puzzle:

1. it randomly samples an input tensor z of shape $[\texttt{n_exmpl}, \texttt{n_colors}, \texttt{width}, \texttt{height}, 2]$ from a standard normal distribution, (line 4)
2. processes this z with a puzzle-specific neural network which has been overfitted to output a $[\texttt{n_exmpl}, \texttt{n_colors}, \texttt{width}, \texttt{height}, 2]$-shaped logit tensor, (line 6)
3. and samples colors for a $[\texttt{n_exmpl}, \texttt{width}, \texttt{height}, 2]$-shaped puzzle using the logits. (line 8)

Algorithm 1: Template for a short program that produces completed puzzles P_{filled} with solutions filled in. Red text is to be substituted in with hard-coded values produced via Algorithm 2.

1	Define an equivariant_NN architecture;	
2		
3	**Set** seed_z = <seed_z$_1$>;	Hardcoded seed from Algo 2, puzzle 1
4	$z \leftarrow \text{sample}_{\text{seed_z}}(N(0,1))$;	Generate inputs z
5	**Set** $\theta = <\theta_1>$;	Hardcoded weights from Algo 2, puzzle 1
6	grid_logits $\leftarrow \text{equivariant_NN}_{\theta}(z)$;	Forward pass
7	**Set** seed_error = <seed_error$_1$>;	Hardcoded seed from Algo 2, puzzle 1
8	$P_{\text{filled}} \leftarrow \text{sample}_{\text{seed_error}}(\text{grid_logits})$;	Generate puzzle
9	**Print** P_{filled}	
10		
11	**Set** seed_z = <seed_z$_2$>;	Hardcoded seed from Algo 2, puzzle 2
12	(...code repeats for all puzzles)	...

The template allows for two pseudo-random sampling seeds to be filled in for every puzzle (lines 3 and 7), completing that portion of the program which prints one puzzle. The printed puzzle can even be forced to match the desired ARC-AGI puzzle (only the question, not solution) by algorithmically manipulating the final seed in line 7, given any possible seed chosen on line 3. With this correctness guarantee out of the way, we only need to consider how to algorithmically choose <θ> and <seed_z> to minimize the total length of the template program, which mainly consists of bits/digits spent to write down the two seeds and the weights <θ>. We chose to design Algorithm 1 to focus primarily on minimizing seed bits, since weight bits are not as easy to minimize in a differentiable way.

3.2 Seed Optimization

We present Algorithm 2 for picking the seeds <seed_error>, <seed_z>, and weights <θ> in template Algorithm 1 to reduce it's total length. Algorithm 2's main idea is to control the neural network

Algorithm 2: Find the best <seeds> and <θ>, which when hard-coded into template Algorithm 1, make it's code as short as possible. Here, `w` and `h` stand for `width` and `height`, respectively.

1 **Input:** ARC-AGI dataset;
2 Define an equivariant_NN architecture;
3 **foreach** *puzzle P in ARC-AGI dataset* **do**
4 Measure `n_exmpl`, `n_colors`, `w`, `h` from P to initialize equivariant_NN$_\theta$;
5 Initialize input distribution μ of shape $[\texttt{n_exmpl}, \texttt{n_colors}, \texttt{w}, \texttt{h}, 2]$, and diagonal Σ;
6 **foreach** *step* **do**
7 **Set** seed_z, by manipulating to imitate $z \sim N(\mu, \Sigma)$;
8 $z \leftarrow \text{sample}_{\text{seed_z}}(N(0,1), [\texttt{n_exmpl}, \texttt{n_colors}, \texttt{w}, \texttt{h}, 2])$; so that $z \sim N(\mu, \Sigma)$
9 grid_logits $\leftarrow$ equivariant_NN$_\theta(z)$;
10 **Set** seed_error, by manipulating to match the observed parts of puzzle P in line 11;
11 $P_{\text{filled}} \leftarrow \text{sample}_{\text{seed_error}}(\text{grid_logits})$;
12 $L \leftarrow \text{len}(\text{seed_z}) + \text{len}(\text{seed_error})$; $\approx$len(Algo 1) + const
13 $\mu, \Sigma, \theta \leftarrow \text{Adam}(\nabla_\mu L, \nabla_\Sigma L, \nabla_\theta L)$;
14 **end foreach**
15 Insert values seed_z, θ, and seed_error into the pseudo-code for Algo 1;
16 **end foreach**
17 **Return** code for Algo 1

inputs z and weights <θ> to create an overfit, which in turn allows fewer bits to be spent manipulating <seed_error> (Algorithm 1 line 7). Conversely, tighter control over z requires spending more bits to manipulate <seed_z> (Algorithm 1 line 4), so a balance must be struck. Concretely, Algorithm 2 learns a "desired" distribution $N(z; \mu, \Sigma)$, and picks <seed_z> to imitate a sample from it, before running the template program forwards and then choosing <seed_error> for correct output. Surrounding this procedure, Algorithm 2 gradient-optimizes μ, Σ, and <θ> to minimize the total seed length.

The issue with running Algorithm 2 directly is that the discrete objective is incompatible with gradient descent, and seed manipulation on lines 7 and 10 takes $O(e^{KL(N(\mu,\Sigma)||N(0,1))})$ time (Flamich et al., 2021). To fix both of these issues, we skip the seed manipulation, and approximate the seed length and manipulated sample instead, resulting in CompressARC (Algorithm 3). Namely for lines 7-8 of Algorithm 2, z is now directly sampled from $N(\mu, \Sigma)$, with $\text{len}(\text{seed_z}) \approx \text{KL}(N(\mu, \Sigma)||N(0, 1))$. For line 10, only the known puzzle grids are clamped to the actual given grids; resulting in $\text{len}(\text{seed_error})$ approximated by the negative log likelihood of the known grids, i.e., the crossentropy only on the observed portions of the puzzle.[1] See Appendix B for derivations.

To summarize:

- Template Algorithm 1 outputs the ARC-AGI puzzles together along with any unrestricted solutions,
- Algorithm 2 fills in the template Algorithm 1 with the best values to make its code as short as possible, thereby influencing the template Algorithm 1 to output "reasonable solutions", and
- CompressARC (Algorithm 3) approximates Algorithm 2, and actually runs in reasonable time.

4 Architecture

The model architecture (used in Algorithm 1 line 1, and Algorithms 2 and 3 line 2) is a quite complicated custom design specifically built to have inductive biases for ARC-AGI puzzles.[2] This is because CompressARC's goal is to reduce the length of the seeds, and overfitting the puzzles as easily as possible is instrumental towards minimizing <seed_error>. Moreover, since the architecture definition only appears once in Algorithm 1 while multiple seeds appear for many puzzles, using a

[1] In practice, we over-penalize the crossentropy by $10\times$, to prevent the KL from collapsing during training. This ends up being a good approximation since the learned solution will end up with vanishing cross-entropy anyways. Therefore the program length bound is hardly affected.

[2] The training puzzles play no role in our method other than to inform our efforts to better engineer the inductive biases into our layers, by helping us design any complicated layers, pick magic coefficients, etc.

Algorithm 3: CompressARC. It is the same as Algorithm 2, but with simulated seed manipulation, and truncated to return solved puzzles instead of description.

Input: ARC-AGI dataset;
Define an equivariant_NN architecture;
foreach *puzzle P in ARC-AGI dataset* **do**
  Measure $\texttt{n_exmpl}, \texttt{n_colors}, \texttt{w}, \texttt{h}$ from P to initialize equivariant_NN$_\theta$;
  Initialize input distribution μ of shape $[\texttt{n_exmpl}, \texttt{n_colors}, \texttt{w}, \texttt{h}, 2]$, and diagonal Σ;
  foreach *step* **do**
    $z \leftarrow \text{sample}(N(\mu, \Sigma))$;
    grid_logits $\leftarrow$ equivariant_NN$_\theta(z)$;
    $L \leftarrow \text{KL}(N(\mu, \Sigma)||N(0,1)) + \text{crossentropy}(\text{grid_logits}, P)$; $\approx$len(Algo 1) + const
    $\mu, \Sigma, \theta \leftarrow \text{Adam}(\nabla_\mu L, \nabla_\Sigma L, \nabla_\theta L)$;
  end foreach
  $P_{\text{filled}} \leftarrow \text{sample}(\text{grid_logits})$;
  Add P_{filled} to solved puzzles
end foreach
Return solved puzzles

sophisticated architecture can help us shorten the length of the template Algorithm 1, sacrificing a longer one-time architecture description to allow for hundreds of shorter seeds.

Since ARC-AGI puzzles are just as likely to appear in any combination of input/output example orderings, colors, orientations, etc., we want our network to assign them all equal probability by default. So, we made our architecture equivariant to example permutations, color permutations, rotations (almost, see Appendix E.2 for caveat), and flips; guaranteeing that any computation learned for one example/color/orientation can also equally be applied to any other example/color/orientation (Cohen & Welling, 2016a). We rely on asymmetric input z to encode any asymmetries a puzzle may have, while the architecture itself remains equivariant.

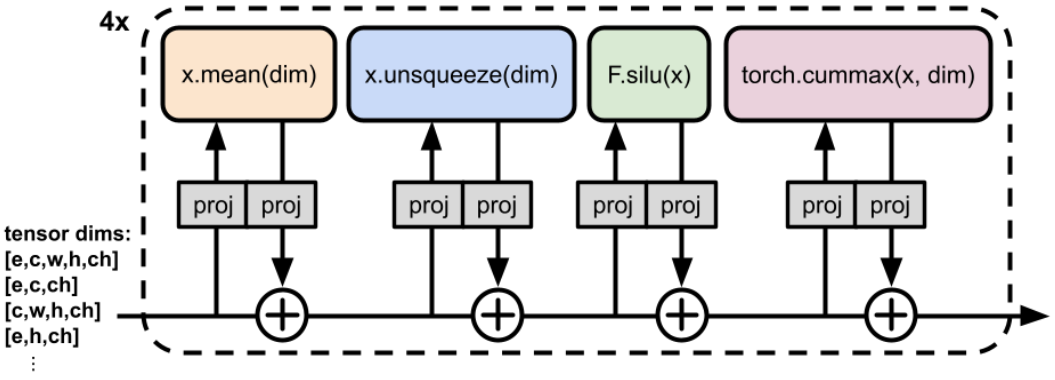


Figure 2: Core structure of CompressARC's neural network, which operates on multitensor data. Individual operations (colored) read and write to a residual backbone through learned projections (grey) in the `channel` dimension. The network is equivariant to permutations of indices along the other, non-`channel` dimensions as a result. Some layers like cummax break certain geometric symmetries, giving the architecture specific geometric abilities listed in Appendix H. Normalization, softmax, shift, and directional layers are not shown.

The architecture functions much like a transformer (Vaswani et al., 2023), with an embedding matrix followed by a sequence of 4 blocks and finally a linear readout (see Figure 2, and Appendices D.1 and D.8). The blocks read and write to the residual stream through linear projections and norms, but the attention and elementwise nonlinear operations are now replaced by a list of other operations better suited to ARC-AGI-shaped data (which can be thought of as $[\texttt{n_exmpl}, \texttt{n_colors}, \texttt{width}, \texttt{height}, \texttt{channel}]$ instead of the classic $[\texttt{batch}, \texttt{seq_len}, \texttt{channel}]$, see Appendix C for details):

- summing one tensor along an axis and/or broadcasting the result back up, (see Appendix D.2)
- taking the softmax along one or multiple axes of a tensor, (see Appendix D.3)
- taking a cumulative max along the `width` or `height` dimensions of a tensor, (see Appendix D.4)
- shifting by one pixel along one of the `width` or `height` dimensions of a tensor, (see Appendix D.4)
- elementwise nonlinearity, as in the standard transformer's FFN layer, (see Appendix D.6)
- normalization, (see Appendix D.7)

along with one more described in Appendices D.5. These operations have no internal parameters internally - they have their behaviors controlled by the surrounding weights in the block. All of these read/write projections operate on the `channel` dimension. We used 16 channels in some parts of the backbone and 8 in others to reduce computational costs. Since these projections take up the majority of the model weights, the entire model only has 76K parameters.

The actual data format that the neural network uses throughout is not a single tensor shaped like $[\texttt{n_exmpl}, \texttt{n_colors}, \texttt{width}, \texttt{height}, \texttt{channel}]$, but instead a bucket of tensors that each have a different subset of dimensions, for example a $[\texttt{n_colors}, \texttt{width}, \texttt{channel}]$ tensor. The network input z, the logit output, and all of the internal activations take this form. This format helps the network to build useful inductive biases by providing natural places for holding information of various shapes. For example, an assignment of grid columns to colors can be stored as a one-hot table in the $[\texttt{color}, \texttt{width}, \texttt{channel}]$-shaped tensor. More details on this format are in Appendix C.

5 Results

We gave CompressARC 2000 steps of inference-time training on every puzzle, taking about 20 minutes per puzzle. With this, CompressARC solves 20% of evaluation set puzzles and 34.75% of training set puzzles. For reference, symbolic solutions and other neural solutions with no prior training solve up to 32% (Joffe & Eliasmith, 2025; Odouard, 2024) of eval puzzles. Figure 3 and Tables 3 and 4 in the Appendix show our performance increase as more inference-time compute is given.

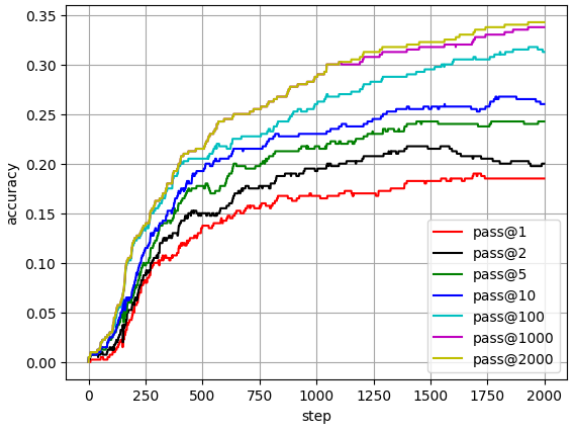


Figure 3: CompressARC's puzzle solve accuracy on the public evaluation set of 400 puzzles rises as inference time learning progresses. Various numbers of allowed solution attempts (pass@n) for accuracy measurement are shown. The official benchmark is reported with 2 allowed attempts, which is why we report 20%.

While ARC-AGI-2 is not the focus of this paper, we were able to submit CompressARC to the more difficult ARC-AGI-2 Kaggle competition, where it solved 4.17% of the puzzles, as a best of six resubmissions. For ARC-AGI-2, we solved multiple puzzles concurrently and saturated the environment's GPU resources to fit the ARC Prize environment's 12 hour budget.

5.1 What Puzzles Can't We Solve?

CompressARC tries to use its abilities to figure out as much as it can, until it gets bottlenecked by one of its inabilities.

Appendix H includes a list of which abilities we have empirically seen CompressARC able to and not able to perform. For example, puzzle 28e73c20 requires extension of a pattern from the edge towards the middle, (see Figure 11a in Appendix). Given the layers in its network, CompressARC is generally able to extend patterns for short ranges but not long ranges. Therefore, it correctly extends the pattern a short distance before guessing what happens near the center (Figure 11b, Appendix).

5.2 Case Study: Color the Boxes

In the puzzle shown (Figure 4), you must color the boxes depending on which side of the grid the box is on. We call this puzzle "Color the Boxes".

Human Solution: We first realize that the input is divided into boxes, and the boxes are still there in the output, but now they're colored. We then try to figure out which colors go in which boxes. First, we notice that the corners are always black. Then, we notice that the middle is always magenta. And after that, we notice that the color of the side boxes depends on which direction they are in: red for up, blue for down, green for right, and yellow for left. At this point, we copy the input over to the answer grid, then we color the middle box magenta, and then color the rest of the boxes according to their direction.

CompressARC Solution: Table 1 shows CompressARC's learning behavior over time. After CompressARC is done learning, we can deconstruct its learned z distribution to find that it codes for a color-direction correspondence table and row/column divider positions (Figure 6).

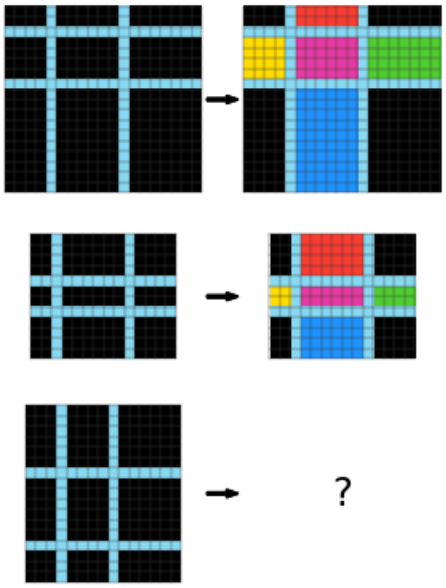


Figure 4: Color the Boxes, puzzle 272f95fa.

During training, the reconstruction error fell far below the KL extremely quickly, indicating an immediate overfit to the given puzzles. This is followed by a long period of representation refinement, during which the KL decreases and the unknown solutions are slowly worked out.

5.2.1 Solution Analysis

We observe the representations stored in z to see how CompressARC learns to solve Color the Boxes.

Since z is a multitensor, each of the tensors it contains produces an additive contribution to the total KL for z. By looking at the per-tensor contributions (see Figure 5), we can determine which tensors in z code for information that is used to represent the puzzle.

All the tensors fall to zero information content during training, except for four tensors. In some replications of this experiment, we saw one of these four necessary tensors fall to zero information content, and CompressARC typically does not recover the correct answer after that. Here we are showing a lucky run where the `[color, direction, channel]` tensor almost falls but gets picked up 200 steps in, which is right around when the samples from the model begin to show the correct colors in the correct boxes.

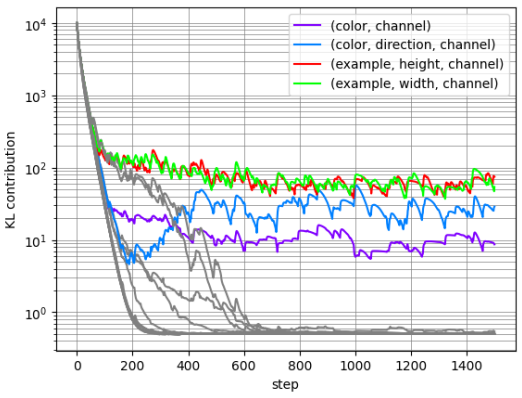
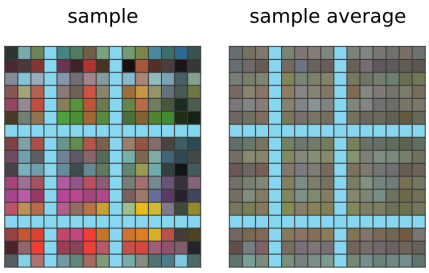
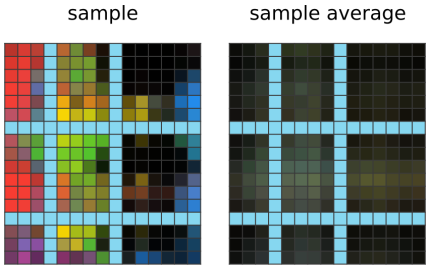
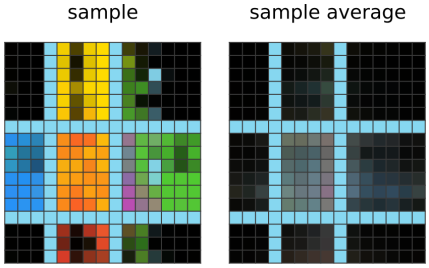
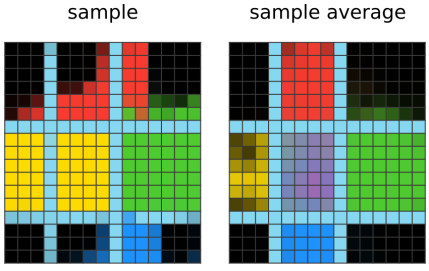
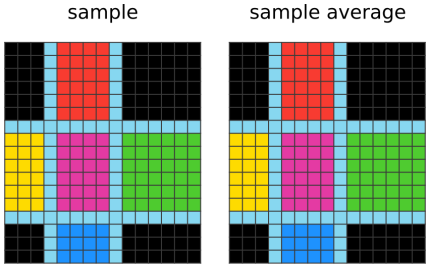
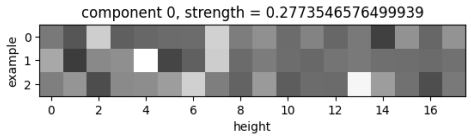


Figure 5: KL loss contribution from each individually shaped tensor in the multitensor z, evolving throughout training. Four latent tensors dominate, indicating they contain information, and the other 14 fall to zero, indicating their lack of information content.

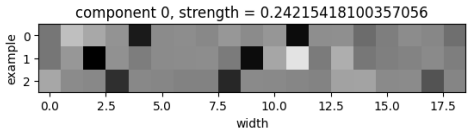
We can look at the average output of the decoding layer (explained in Appendix D.1) corresponding to individual tensors of z, to see what information is stored there (see Figure 6). Each tensor contains a vector of dimension `n_channels` for various indices of the tensor. Taking the PCA of these vectors reveals some number of activated components, telling us how many pieces of information are coded by the tensor.

Table 1: CompressARC learning the solution for Color the Boxes, over time.

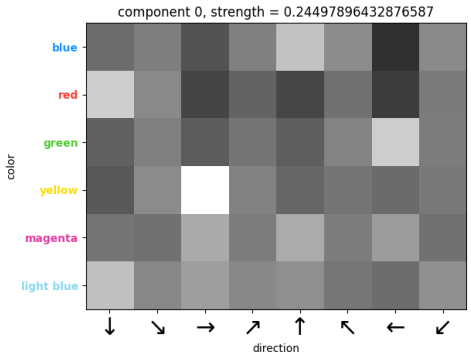
Learning steps	What is CompressARC doing?	Sampled solution guess
50	CompressARC's network outputs an answer grid (sample) with light blue rows/columns wherever the input has the same. It has noticed that all the other input-output pairs in the puzzle exhibit this correspondence. It doesn't know how the other output pixels are assigned colors; an exponential moving average of the network output (sample average) shows the network assigning mostly the same average color to non-light-blue pixels.	
150	The network outputs a grid where nearby pixels have similar colors. It has likely noticed that this is common among all the outputs, and is guessing that it applies to the answer too.	
200	The network output now shows larger blobs of colors that are cut off by the light blue borders. It has noticed the common usage of borders to demarcate blobs of colors in other outputs, and applies the same idea here. It has also noticed black corner blobs in other given outputs, which the network imitates.	
350	The network output now shows the correct colors assigned to boxes of the correct direction from the center. It has realized that a single color-to-direction mapping is used to pick the blob colors in the other given outputs, so it imitates this mapping. It is still not the best at coloring within the lines, and it's also confused about the center blob, probably because the middle does not correspond to a direction. Nevertheless, the average network output does show a tinge of the correct magenta color in the middle, meaning the network is catching on.	
1500	The network is as refined as it will ever be. Sometimes it will still make a mistake in the sample it outputs, but this is uncommon and filtered out.	



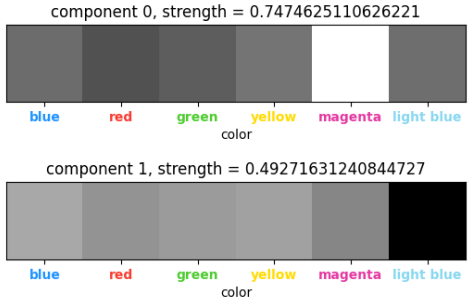
(a) (**`example`**, **`height`**, **`channel`**) **tensor.** For every example and row, there is a vector of dimension `n_channels`. Taking the PCA of this set of vectors, the top principal component (1485 times stronger than the next component) visualized as the (`example`, `height`) matrix shown above shows which examples/row combinations are uniquely identified by the stored information. **For every example, the two brightest pixels give the rows where the light blue rows in the grids are.**



(b) (**`example`**, **`width`**, **`channel`**) **tensor.** A similar story here to 6a: in the top principal component of this tensor, **the two darkest pixels for every example give the columns where the light blue columns in the grids are.** The top principal component is 1253 times stronger than the next.



(c) (**`direction`**, **`color`**, **`channel`**) **tensor.** The four brightest pixels identify blue with up, green with left, red with down, and yellow with right. **This tensor tells each direction which color to use for the opposite edge's box.** The top principal component is 829 times stronger than the next.



(d) (**`color`**, **`channel`**) **tensor.** The first and second principal components are 134 and 87 times stronger than the third component, indicating that they play a role while the third component does not. The **magenta and light blue colors** are uniquely identified, indicating their special usage amongst the rest of the colors as **the center color and the color of the row/column divisions**, respectively.

Figure 6: Mechanistic explanation of what information ends up being stored in the learned latents, and how it can be used to reconstruct the puzzle contents.

6 Discussion

The prevailing reliance of modern deep learning on high-quality data has severely hampered its ability to solve problems that require intelligence but have less data available. This is especially true for the data-limited ARC-AGI benchmark, where LLMs trained on specially augmented, extended, and curated datasets dominate (Knoop, 2024) and inference-time learning strategies are plentiful. CompressARC strikes a different balance by forgoing the usage of any training data as well as the entire process of pretraining altogether, in favor of using inference compute for learning. One should intuitively expect this to fail and solve no puzzles at all, but by applying MDL to the target puzzle during inference time, CompressARC solves a surprisingly large portion of ARC-AGI-1.

CompressARC's theoretical underpinnings come from minimizing the length of a programmatic description of the target puzzle. While other MDL strategies suffer from an intractably large search space of possible discrete programs (Ferré, 2025; Ferré, 2021), CompressARC explores a simplified, neural network-based search space through gradient descent. Though CompressARC's architecture is heavily engineered, its ability to use MDL to generalize from as low as two demonstration input/output pairs demonstrates that neural ARC-AGI solvers are feasible even in this low-data regime.

We challenge the assumption that intelligence must arise from massive pretraining and data, instead showing that combining deep learning frameworks with MDL at inference time can lead to a surprising ability to intelligently generalize from minimal data.

7 Generative AI Usage Disclosure

Large Language Models (LLMs) were used to polish the writing in this paper, specifically to find the most clear and concise way of introducing our work in Section 1. They were also used to help us understand and digest the proof in Li & Gamal (2018) that Relative Entropy Coding is a valid algorithm that works as claimed. We have heavily reviewed all AI-assisted writing. No other part of this work was performed using generative AI, including research ideation, implementation, execution, writing, drawing figures, nor collecting references and prior work.

8 Reproducibility

The full algorithm we used is described in Sections 3 and 4. Detailed description of the architecture and training details are provided in the Appendices C, D, and E when not described in the main text, while pre/post-processing techniques are described in Appendices F and G. Benchmarking details are provided in Appendix K. For any details we may have missed, code used in this project is attached in the supplemental materials.

References

Ekin Akyürek, Mehul Damani, Adam Zweiger, Linlu Qiu, Han Guo, Jyothish Pari, Yoon Kim, and Jacob Andreas. The surprising effectiveness of test-time training for few-shot learning. *arXiv preprint arXiv:2411.07279*, 2024.

Anthropic. Claude fable 5.1: Arc-agi results. ARC Prize, September 2026. URL https://arcprize.org/results/anthropic-claude-fable-5-1.

Guillermo Barbadillo. Solution summary for arc24. https://ironbar.github.io/arc24/05_Solution_Summary/, 2024. Accessed: 2025-05-12.

Clément Bonnet and Matthew V Macfarlane. Searching latent program spaces, 2024. URL https://arxiv.org/abs/2411.08706.

Natasha Butt, Blazej Manczak, Auke Wiggers, Corrado Rainone, David W Zhang, Michaël Defferrard, and Taco Cohen. Codeit: Self-improving language models with prioritized hindsight replay. *arXiv preprint arXiv:2402.04858*, 2024.

Vinod Kumar Chauhan, Jiandong Zhou, Ping Lu, Soheila Molaei, and David A. Clifton. A brief review of hypernetworks in deep learning. *Artificial Intelligence Review*, 57(9), August 2024. ISSN 1573-7462. doi: 10.1007/s10462-024-10862-8. URL http://dx.doi.org/10.1007/s10462-024-10862-8.

François Chollet. On the measure of intelligence, 2019. URL https://arxiv.org/abs/1911.01547.

François Chollet. Openai o3 breakthrough high score on arc-agi-pub. https://arcprize.org/blog/oai-o3-pub-breakthrough, 2024. Accessed: 2025-05-12.

Taco Cohen and Max Welling. Group equivariant convolutional networks. In Maria Florina Balcan and Kilian Q. Weinberger (eds.), *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pp. 2990–2999, New York, New York, USA, 20–22 Jun 2016a. PMLR. URL https://proceedings.mlr.press/v48/cohenc16.html.

Taco S. Cohen and Max Welling. Group equivariant convolutional networks, 2016b. URL https://arxiv.org/abs/1602.07576.

Shiqing Fan, Liu Liying, and Ye Luo. An alternative practice of tropical convolution to traditional convolutional neural networks, 2021. URL https://arxiv.org/abs/2103.02096.

Sébastien Ferré. First steps of an approach to the arc challenge based on descriptive grid models and the minimum description length principle. *arXiv preprint arXiv:2112.00848*, 2021.

Sébastien Ferré. Madil: An mdl-based framework for efficient program synthesis in the arc benchmark, 2025. URL https://arxiv.org/abs/2505.01081.

Gergely Flamich, Marton Havasi, and José Miguel Hernández-Lobato. Compressing images by encoding their latent representations with relative entropy coding, 2021. URL https://arxiv.org/abs/2010.01185.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines, 2014. URL https://arxiv.org/abs/1410.5401.

Ryan Greenblatt. Getting 50% (sota) on arc-agi with gpt-4o. https://redwoodresearch.substack.com/p/getting-50-sota-on-arc-agi-with-gpt, 2024. Accessed: 2025-05-12.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. URL https://arxiv.org/abs/1512.03385.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2023. URL https://arxiv.org/abs/1606.08415.

Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-VAE: Learning basic visual concepts with a constrained variational framework. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=Sy2fzU9gl.

Michael Hodel. Domain specific language for the abstraction and reasoning corpus. https://github.com/michaelhodel/arc-dsl/blob/main/arc_dsl_writeup.pdf, 2024. Accessed: 2025-05-12.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

Keya Hu, Ali Cy, Linlu Qiu, Xiaoman Delores Ding, Runqian Wang, Yeyin Eva Zhu, Jacob Andreas, and Kaiming He. Arc is a vision problem! In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2537–2546, 2026.

Marcus Hutter. A theory of universal artificial intelligence based on algorithmic complexity. *arXiv preprint cs/0004001*, 2000.

Marcus Hutter. *The Universal Algorithmic Agent AIXI*, pp. 141–183. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-26877-2. doi: 10.1007/3-540-26877-4_5. URL https://doi.org/10.1007/3-540-26877-4_5.

Marcus Hutter. Hutter prize for lossless compression of human knowledge. https://prize.hutter1.net/, 2006. Accessed: 2025-05-12.

Isaac Joffe and Chris Eliasmith. Vector symbolic algebras for the abstraction and reasoning corpus. *arXiv preprint arXiv:2511.08747*, 2025.

Alexia Jolicoeur-Martineau. Less is more: Recursive reasoning with tiny networks. *arXiv preprint arXiv:2510.04871*, 2025.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2022. URL https://arxiv.org/abs/1312.6114.

Mike Knoop. ARC Prize 2024 Winners & Technical Report Published — arcprize.org. https://arcprize.org/blog/arc-prize-2024-winners-technical-report, 2024. [Accessed 12-05-2025].

A.N. Kolmogorov. On tables of random numbers. *Theoretical Computer Science*, 207(2):387–395, 1998. ISSN 0304-3975. doi: https://doi.org/10.1016/S0304-3975(98)00075-9. URL https://www.sciencedirect.com/science/article/pii/S0304397598000759.

G. G. Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2):135–149, 1984. doi: 10.1147/rd.282.0135.

Solim LeGris, Wai Keen Vong, Brenden M. Lake, and Todd M. Gureckis. H-arc: A robust estimate of human performance on the abstraction and reasoning corpus benchmark, 2024. URL https://arxiv.org/abs/2409.01374.

Cheuk Ting Li and Abbas El Gamal. Strong functional representation lemma and applications to coding theorems. *IEEE Transactions on Information Theory*, 64(11):6967–6978, 2018. doi: 10.1109/TIT.2018.2865570.

Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer M. Dunn, Hao Tang, Michelangelo Naim, Dat Nguyen, Wei-Long Zheng, Zenna Tavares, Yewen Pu, and Kevin Ellis. Combining induction and transduction for abstract reasoning, 2024a. URL https://arxiv.org/abs/2411.02272.

Wenhao Li, Yudong Xu, Scott Sanner, and Elias Boutros Khalil. Tackling the abstraction and reasoning corpus with vision transformers: the importance of 2d representation, positions, and objects. *arXiv preprint arXiv:2410.06405*, 2024b.

Arseny Moskvichev, Victor Vikram Odouard, and Melanie Mitchell. The conceptarc benchmark: Evaluating understanding and generalization in the arc domain. *arXiv preprint arXiv:2305.07141*, 2023.

Victor Vikram Odouard. Arc-solution_documentation. https://github.com/victorvikram/ARC-icecuber/blob/master/ARC-solution_documentation.pdf, 2024. Accessed: 2025-05-12.

J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978. ISSN 0005-1098. doi: https://doi.org/10.1016/0005-1098(78)90005-5. URL https://www.sciencedirect.com/science/article/pii/0005109878900055.

C. Shannon. The zero error capacity of a noisy channel. *IRE Transactions on Information Theory*, 2 (3):8–19, 1956. doi: 10.1109/TIT.1956.1056798.

C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3): 379–423, 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x.

R.J. Solomonoff. A formal theory of inductive inference. part i. *Information and Control*, 7(1):1–22, 1964. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(64)90223-2. URL https://www.sciencedirect.com/science/article/pii/S0019995864902232.

Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei A. Efros, and Moritz Hardt. Test-time training with self-supervision for generalization under distribution shifts, 2020. URL https://arxiv.org/abs/1909.13231.

Casper Kaae Sønderby, Tapani Raiko, Lars Maaløe, Søren Kaae Sønderby, and Ole Winther. Ladder variational autoencoders, 2016. URL https://arxiv.org/abs/1602.02282.

Arash Vahdat and Jan Kautz. Nvae: A deep hierarchical variational autoencoder, 2021. URL https://arxiv.org/abs/2007.03898.

Mithil Vakde. mdlarc, 2025. URL https://github.com/mvakde/mdlARC.

Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning, 2018. URL https://arxiv.org/abs/1711.00937.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

Guan Wang, Jin Li, Yuhao Sun, Xing Chen, Changling Liu, Yue Wu, Meng Lu, Sen Song, and Yasin Abbasi Yadkori. Hierarchical reasoning model. *arXiv preprint arXiv:2506.21734*, 2025.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture, 2020. URL https://arxiv.org/abs/2002.04745.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. Deep sets, 2018. URL https://arxiv.org/abs/1703.06114.

A Related Work

A.1 Equivalence of Compression and Intelligence

The original inspiration of this work came from the Hutter Prize (Hutter, 2006), which challenges participants to compress a file of Wikipedia text the most, to serve as a motivation for researchers to build intelligent systems. It is premised upon the idea that the ability to compress information is equivalent to intelligence.

This equivalence between intelligence and compression has a long history. As an example, for the most general form of prediction problems, the ideal predictor implements Solomonoff Induction, a theoretically best possible but uncomputable prediction algorithm that works universally for all prediction tasks. (Solomonoff, 1964) This prediction algorithm is then equivalent to a best possible compression algorithm whose compressed code length is the Kolmogorov Complexity of the data. (Kolmogorov, 1998) This prediction algorithm can also be used to decode a description of the data of minimal length, linking these formulations of intelligence to MDL. (Rissanen, 1978) In our work, we try to approximate this best possible compression algorithm with a neural network.

A.2 Information Theory and Coding Theory

Since we build an information compression system, we make use of many results in information theory and coding theory. The main result required to motivate our model architecture is the existence of a Relative Entropy Coding (REC) algorithm (Flamich et al., 2021; Li & Gamal, 2018). REC is an algorithm which allows a decoder which only knows a distribution Q to draw from a distribution P if only an encoder knowing both distribution sends (approximately, and in expectation) $KL(P||Q)$ bits over to the decoder. The fact that REC exists means that as long as a KL divergence can be bounded, the construction of a compression algorithm for data drawn from P is always possible, and the issue of realizing the algorithm can be abstracted away. Thus, details regarding the translation of information between real Gaussian representation and binary can be ignored, as long as we know the relevant KL divergences. While the existence of arithmetic coding (Langdon, 1984) would suffice to abstract the problem away when distributions are discrete, neural networks operate in a continuous space so we require knowledge of the existence of REC algorithms instead.

Our architecture sends z information through an additive white Gaussian noise (AWGN) channel, so the AWGN channel capacity formula (Gaussian input Gaussian noise) plays a heavy role in the design of our decoding layer. (Shannon, 1948)

A.3 Variational Autoencoders

The decoder side of the variational autoencoder (Kingma & Welling, 2022) serves as our decompression algorithm. While we would use something that has more general capabilities like a neural Turing machine (Graves et al., 2014) instead, neural Turing machines are not very amenable to gradient descent-based optimization so we opted for the VAE.

VAEs have a long history of developments that are relevant to our work. At one point, we tried using multiple decoding layers to make a hierarchical VAE decoder (Sønderby et al., 2016) instead. This does not affect the KL calculation, because a channel capacity with feedback is equal to the channel capacity without feedback. (Shannon, 1956) We found empirically that the first decoding layer would absorb all of the KL contribution making the later decoding layers redundant, so our final design used one decoding layer at the beginning.

The beta-VAE (Higgins et al., 2017) introduces a reweighting of the reconstruction loss to be stronger than the KL loss, and we found that to work well in our case. The NVAE applies a non-constant weighting to loss components. (Vahdat & Kautz, 2021) A rudimentary form of scheduled loss recombination is used in CompressARC.

A.4 Deep Learning Architectures

We designed our own neural network architecture from scratch, but not without borrowing crucial design principles from many others.

Our architecture is fundamentally structured like a transformer, consisting of a residual stream where representations are stored and operated upon, followed by a linear head. (Vaswani et al., 2023; He et al., 2015) Pre-and post-norms with linear up- and down-projections allow layers to read and write to the residual stream. (Xiong et al., 2020) The SiLU-based nonlinear layer is especially similar to a transformer's. (Hendrycks & Gimpel, 2023)

Our equivariance structures are inspired by permutation-invariant neural networks, which are a type of equivariant neural network. (Zaheer et al., 2018; Cohen & Welling, 2016b) Equivariance transformations are taken from common augmentations to ARC-AGI puzzles.

B SEED LENGTH ESTIMATION BY KL AND CROSSENTROPY

In Section 3.1, we estimate len(seed_z) in line 12 of template Algorithm 2 as $\mathrm{KL}(N(\mu, \Sigma)||N(0, 1))$, and len(seed_error) as crossentropy(grid_logits, P). In this section, we will argue for the reasonability of this approximation. Readers may also refer to Flamich et al. (2021), which gives an overview of works on algorithms achieving similar approximations - note that their notational conventions swap P and Q compared to ours. Literature on REC generally shows that a shared source of randomness (which in our work we refer to as "seed manipulation") enables a bit-efficient way for an encoder to communicate samples from a distribution P to a decoder if the decoder only knows a sampling distribution Q. REC uses close to $\mathrm{KL}(P||Q)$ expected bits per sample communicated.

As a reminder of the seed manipulation problem we need to solve, the original procedure in Algorithm 2 manipulates the seed for sampling $z \sim Q = N(0, 1)$ to simulate as though $z \sim P = N(\mu, \Sigma)$, and we would like to show how this is done so that the expected number of bits in the seed is approximately $\mathrm{KL}(P||Q) = \mathrm{KL}(N(\mu, \Sigma)||N(0, 1))$. (This is for seed_z. For seed_error, the same argument holds using $P =$ Delta_distribution(puzzle) and $Q =$ Categorical_distribution(logits), for which the KL can be calculated as equal to the cross-entropy.)

In the following, we will go over a solution and proof for this problem which originates from Li & Gamal (2018). All credit for the following goes to them. The algorithm itself is not too complicated, and consists of three steps:

- Draw a sequence of samples $z_1, z_2, \cdots \sim Q$ and $t_1, t_2, \cdots \sim \mathrm{Exp}(1)$. This is done on both the encoder and decoder sides using a shared source of randomness so the values drawn from each side are the same.
- The encoder computes $T_j = \sum_{i=1}^{j} t_j$ and then $N = \arg\min_n T_n \frac{Q(z_n)}{P(z_n)}$, and sends the value of N to the decoder.
- The decoder receives the value of N and selects the value of z_N.

Three properties need to be proven for this algorithm to solve our problem:

1. The decoded value z_N must indeed be distributed like P,
2. The expected number of bits in N must indeed be $\leq KL(P||Q) + O(1)$,
3. The algorithm above must terminate.

We will show the proof for each property in sequence.

For property (1), we make a Poisson point process density transformation argument. Firstly, T_j as generated by the algorithm are distributionally equivalent to sorted samples from a Poisson process with density 1. This is true because in the Poisson process, the spacing between consecutive samples $t_i = T_i - T_{i-1}$ will be distributed i.i.d. as $\mathrm{Exp}(1)$, which is exactly how the algorithm draws t_i in the first place. This makes the pairs (T_i, z_i) equivalent to T-ordered samples from a Poisson process with density $\mathrm{d}\Lambda = \mathrm{d}T\mathrm{d}Q(z)$. Under the mapping $T' = T\frac{dQ(z)}{dP(z)}$, we obtain the density in (T', z) form: $\mathrm{d}\Lambda = \mathrm{d}T'\mathrm{d}P(z)$. Then, T' and z become independent, so picking the sample of z with the smallest T' value (which is what the algorithm does) is sufficient to get a sample from P.

For property (2), we'll make an upper bound on $\mathbb{E}[\log N]$, the number of bits in N. Recalling that $N = \arg\min_n T'_n$ while T' is drawn from Poisson point process $\mathrm{d}\Lambda = \mathrm{d}T'\mathrm{d}P(z)$, we realize that $T'_N \sim \mathrm{Exp}(1)$, since the first sample of a Poisson point process is exponentially distributed.

And since $T' = T\frac{dQ(z)}{dP(z)}$, we must then have $T_N \frac{dQ(z_N)}{dP(z_N)} \sim \text{Exp}(1)$. Then, we take the expected log: $\mathbb{E}[\log T_N] + \mathbb{E}[\log \frac{dQ(z_N)}{dP(z_N)}] = -\mathbb{E}[\text{Gumbel}(-\gamma, 1)] - \gamma$. The second term is just $-KL(P||Q)$ since $z_n \sim P$, and the third expectation gives zero. Substituting these values in gives $\mathbb{E}[\log T_N] = KL(P||Q) - \gamma$. At this point, we need to convert $\mathbb{E}[\log T_N]$ into $\mathbb{E}[\log N]$. To do this, notice that the distribution over $N-1$ conditioned on T_N and z_N is Poisson-distributed, due to $N-1$ being a count of samples drawn from a Poisson process over a region (specifically, the region $\{(T, z) : T < T_N \text{ and } T\frac{Q(z)}{P(z)} > T_N \frac{Q(z_N)}{P(z_N)}\}$). That region has at most a total measure of T_N, which means $\mathbb{E}[N] \leq \mathbb{E}[1 + \text{Poisson}(T_N)] = T_N + 1$ for any given (T_N, z_N). Taking a log and applying Jensen's inequality gives $\mathbb{E}[\log N] \leq \log(T_N + 1) = \log T_N + \log(1 + 1/T_N) \leq \log(T_N) + O(1)$, and finally taking the expectation over (T_N, z_N) gives us $\mathbb{E}[\log N] \leq \mathbb{E}[\log T_N] + O(1)$. Putting it all together gives us the desired inequality, $\mathbb{E}[\log N] \leq KL(P||Q) + O(1)$, completing the proof.

For property (3), we show that after searching for the argmin up to a certain value of n, we can prove that we don't need to search any higher values, so the encoder can always eventually terminate it's search safely and discard any higher n. To see why, consider that in our setup, P and Q are always Gaussian and P is always narrower than Q, according to our parameterization in Appendix D.1. This places a lower bound on $Q(z)/P(z)$. As n counts up, T_n must increase, so the lower bound on possible values of $T_n Q(z_n)/P(z_n)$ will always increase. Eventually, once the lower bound crosses the running argmin, higher values of n will provably leave the argmin unaffected, and the search can terminate. Since T_n becomes arbitrarily high with probability 1, termination happens with probability 1.

In summary, the scheme above originating from Li & Gamal (2018) allows the encoder to send a sample from distribution P to a decoder who knows only distribution Q, for an expected cost of $KL(P||Q) + O(1)$ bits.

C MULTITENSORS

The actual data (z, hidden activations, and puzzles) passing through our layers comes in a format that we call a "**multitensor**", which is just a bucket of tensors of various shapes, as shown in Figure 7. All the equivariances we use can be described in terms of how they change a multitensor.

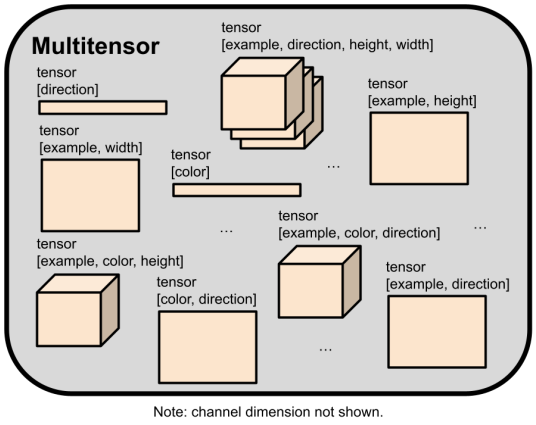


Figure 7: Our neural network's internal representations come in the form of a "multitensor", a bucket of tensors of different shapes. One of the tensors is shaped like $[\texttt{example}, \texttt{color}, \texttt{height}, \texttt{width}, \texttt{channel}]$, an adequate shape for storing a whole ARC-AGI puzzle.

Most common classes of machine learning architectures operate on a single type of tensor with constant rank. LLMs operate on rank-3 tensors of shape $[\text{n_batch}, \text{n_tokens}, \texttt{n_channels}]$, and Convolutional Neural Networks (CNNs) operate on rank-4 tensors of shape $[\text{n_batch}, \texttt{n_channels}, \texttt{height}, \texttt{width}]$. Our multitensors are a set of varying-rank tensors of unique type, whose dimensions are a subset of a rank-6 tensor of shape $[\texttt{n_exmpl}, \texttt{n_colors}, \texttt{n_directions}, \texttt{height}, \texttt{width}, \texttt{n_channels}]$, as illustrated

in Figure 7. We always keep the channel dimension, so there are at most 32 tensors in each multitensor. We also maintain several rules (see Appendix E.1) that determine whether a tensor shape is "legal" or not, which reduces the number of tensors in a multitensor to 18.

Dimension	Description
Example	Number of examples in the ARC-AGI puzzle, including the one with held-out answer
Color	Number of unique colors in the ARC-AGI puzzle, not including black, see Appendix F.2
Direction	8
Height	Determined when preprocessing the puzzle, see Appendix F.1
Width	Determined when preprocessing the puzzle, see Appendix F.1
Channel	In the residual connections, the size is 8 if the `direction` dimension is included, else 16. Within layers it is layer-dependent.

Table 2: Size conventions for multitensor dimensions.

To give an idea of how a multitensor stores data, an ARC-AGI puzzle can be represented by using the $[\texttt{example}, \texttt{color}, \texttt{height}, \texttt{width}, \texttt{channel}]$ tensor, by using the `channel` dimension to select either the input or output grid, and the `height`/`width` dimensions for pixel location, a one hot vector in the `color` dimension, specifying what color that pixel is. The $[\texttt{example}, \texttt{height}, \texttt{channel}]$ and $[\texttt{example}, \texttt{width}, \texttt{channel}]$ tensors can similarly be used to store masks representing grid shapes for every example for every input/output grid. All those tensors are included in a single multitensor that is computed by the network just before the final linear head (described in Appendix D.8).

When we apply an operation on a multitensor, we by default assume that all non-`channel` dimensions are treated identically as batch dimensions by default. The operation is copied across the indices of dimensions unless specified. This ensures that we keep all our symmetries intact until we use a specific layer meant to break a specific symmetry.

A final note on the `channel` dimension: usually when talking about a tensor's shape, we will not even mention the `channel` dimension as it is included by default.

D Layers in the Architecture

D.1 Decoding Layer

This layer's job is to sample a multitensor z and bound its information content, before it is passed to the next layer. This layer and outputs the KL divergence between the learned z distribution and $N(0, I)$. Penalizing the KL prevents CompressARC from learning a distribution for z that memorizes the ARC-AGI puzzle in an uncompressed fashion, and forces CompressARC to represent the puzzle more succinctly. Specifically, it forces the network to spend more bits on the KL whenever it uses z to break a symmetry, and the larger the symmetry group broken, the more bits it spends.

This layer takes as input:

- A learned target multiscalar, called the "target capacity".[3] The decoding layer will output z whose information content per tensor is close to the target capacity,[4]
- learned per-element means for z,[5]
- learned per-element capacity adjustments for z.

[3]Target capacities are exponentially parameterized and rescaled by 10x to increase sensitivity to learning, initialized at a constant 10^4 nats per tensor, and forced to be above a minimum value of half a nat.

[4]The actual information content, which the layer computes later on, will be slightly different because of the per-element capacity adjustments.

[5]Means are initialized using normal distribution of variance 10^{-4}.

We begin by normalizing the learned per-element means for z.[6] Then, we figure out how much Gaussian noise we must add into every tensor to make the AWGN channel capacity (Shannon, 1948) equal to the target capacity for every tensor (including per-element capacity adjustments). We apply the noise to sample z, keeping unit variance of z by rescaling.[7]

We compute the information content of z as the KL divergence between the distribution of this sample and $N(0, 1)$.

Finally, we postprocess the noisy z by scaling it by the sigmoid of the signal-to-noise ratio.[8] This ensures that z is kept as-is when its variance consists mostly of useful information and it is nearly zero when its variance consists mostly of noise. All this is done 4 times to make a channel dimension of 4. Then we apply a projection (with different weights per tensor in the multitensor, i.e., per-tensor projections) mapping the channel dimension up to the dimension of the residual stream.

D.2 Multitensor Communication Layer

This layer allows different tensors in a multitensor to interact with each other.

First, the input from the residual stream passes through per-tensor projections to a fixed size (8 for downwards communication and 16 for upwards communication). Then a message is sent to every other tensor that has at least the same dimensions for upwards communication, or at most the same dimensions for downwards communication. This message is created by either taking means along dimensions to remove them, or unsqueezing+broadcasting dimensions to add them, as in Figure 8. All the messages received by every tensor are summed together and normalization is applied. This result gets up-projected back and then added to the residual stream.

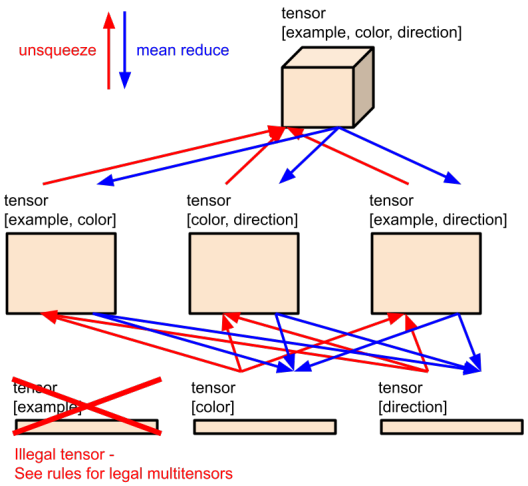


Figure 8: Multitensor communication layer. Higher rank tensors shown at the top, lower rank at the bottom. Tensors transform between ranks by mean reduction and unsqueezing dimensions.

D.3 Softmax Layer

This layer allows the network to work with internal one-hot representations, by giving it the tools to denoise and sharpen noisy one-hot vectors. For every tensor in the input multitensor, this layer lists out all the possible subsets of dimensions of the tensor to take a softmax over,[9] takes the softmax

[6] Means and variances for normalization are computed along all non-channel dimensions.

[7] There are many caveats with the way this is implemented and how it works; please refer to the code (see Appendix N) for more details.

[8] We are careful not to let the postprocessing operation, which contains unbounded amounts of information via the signal-to-noise ratios, to leak lots of information across the layer. We only let a bit of it leak by averaging the signal-to-noise ratios across individual tensors in the multitensor.

[9] One exception: we always exclude the example dimension in the subset of dimensions.

over these subsets of dimensions, and concatenates all the softmaxed results together in the channel dimension. The output dimension varies across different tensors in the multitensor, depending on their tensor rank. A pre-norm is applied, and per-tensor projections map to and from the residual stream. The layer has input channel dimension of 2.

D.4 Directional Cummax/Shift Layer

The directional cummax and shift layers allow the network to perform the non-equivariant cummax and shift operations in an equivariant way, namely by applying the operations once per direction, and only letting the output be influenced by the results once the directions are aggregated back together (by the multitensor communication layer). These layers are the sole reason we included the `direction` dimension when defining a multitensor: to store the results of directional layers and operate on each individually. Of course, this means when we apply a spatial equivariance transformation, we must also permute the indices of the `direction` dimension accordingly, which can get complicated sometimes.

The directional cummax layer takes the eight indices of the direction dimension, treats each slice as corresponding to one direction (4 cardinal, 4 diagonal), performs a cumulative max in the respective direction for each slice, does it in the opposite direction for half the channels, and stacks the slices back together in the direction dimension. An illustration is in Figure 9. The slices are each independently rescaled to have min -1 and max 1 before applying the cumulative max.

The directional shift layer does the same thing, but for shifting the grid by one pixel instead of applying the cumulative max, and without the rescaling.

Some details:

- Per-tensor projections map to and from the residual stream, with pre-norm.
- Input channel dimension is 4.
- These layers are only applied to the $[\texttt{example}, \texttt{color}, \texttt{direction}, \texttt{height}, \texttt{width}, \texttt{channel}]$ and $[\texttt{example}, \texttt{direction}, \texttt{height}, \texttt{width}, \texttt{channel}]$ tensors in the input multitensor.

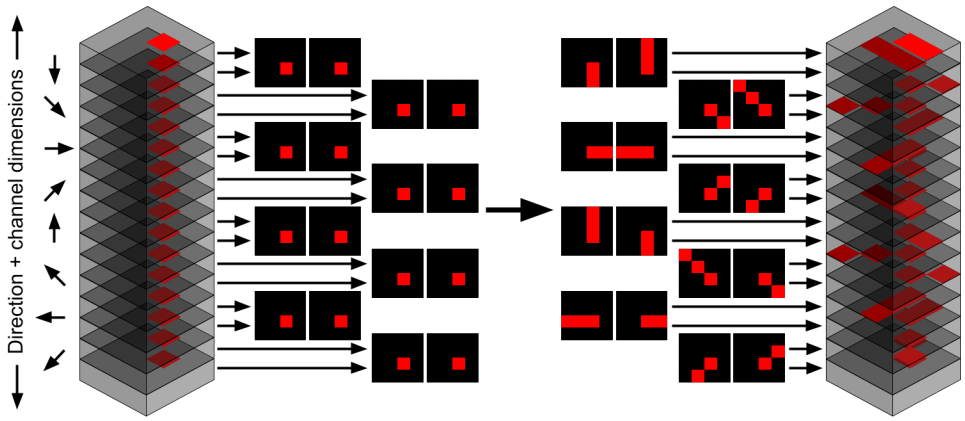


Figure 9: The directional cummax layer takes a directional tensor, splits it along the direction axis, and applies a cumulative max in a different direction for each direction slice. This operation helps CompressARC transport information across long distances in the puzzle grid.

D.5 Directional Communication Layer

By default, the network is equivariant to permutations of the eight directions, but we only want symmetry up to rotations and flips. So, this layer provides a way to send information between two slices in the direction dimension, depending on the angular difference in the two directions. This layer defines a separate linear map to be used for each of the 64 possible combinations of angles,

but the weights of the linear maps are minimally tied such that the directional communication layer is equivariant to reflections and rotations. This gets complicated really fast, since the direction dimension's indices also permute when equivariance transformations are applied. Every direction slice in a tensor accumulates its 8 messages, and adds the results together.[10]

For this layer, there are per-tensor projections to and from the residual stream with pre-norm. The input channel dimension is 2.

D.6 Nonlinear Layer

We use a SiLU nonlinearity with channel dimension 16, surrounded by per-tensor projections with pre-norm.

D.7 Normalization Layer

We normalize all the tensors in the multitensor, using means and variances computed across all dimensions except the channel dimension. Normalization as used within other layers also generally operates this way.

D.8 Linear Heads

We must take the final multitensor, and convert it to the format of an ARC-AGI puzzle. More specifically, we must convert the multitensor into a distribution over ARC-AGI puzzles, so that we can compute the log-likelihood of the observed grids in the puzzle.

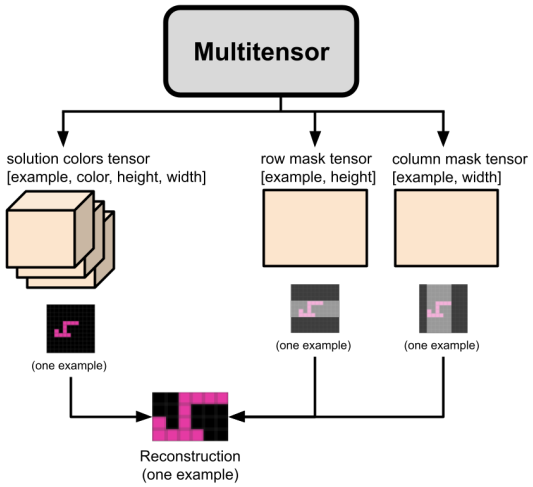


Figure 10: The linear head layer takes the final multitensor of the residual stream and reads a $[\texttt{example}, \texttt{color}, \texttt{height}, \texttt{width}, \texttt{channel}]$ tensor to be interpreted as color logits, and a $[\texttt{example}, \texttt{height}, \texttt{channel}]$ tensor and a $[\texttt{example}, \texttt{width}, \texttt{channel}]$ tensor to serve as shape masks.

The colors of every pixel for every example for both input and output, have logits defined by the $[\texttt{example}, \texttt{color}, \texttt{height}, \texttt{width}, \texttt{channel}]$ tensor, with the channel dimension linearly mapped down to a size of 2, representing the input and output grids.[11] The log-likelihood is given by the crossentropy, with sum reduction across all the grids.

[10] We also multiply the results by coefficients depending on the angle: 1 for 0 degrees and 180 degrees, 0.2 for 45 degrees and 135 degrees, and 0.4 for 90 degrees. This biases the layer to transmit information across more common rotations.

[11] The linear map is initialized to be identical for both the input and output grid, but isn't fixed this way during learning. Sometimes this empirically helps with problems of inconsistent input vs output grid shapes. The bias on this linear map is multiplied by 100 before usage, otherwise it doesn't seem to be learned fast enough empirically. This isn't done for the shape tensors described by the following paragraph though.

For grids of non-constant shape, the [example, height, channel] and [example, width, channel] tensors are used to create distributions over possible contiguous rectangular slices of each grid of colors, as shown in Figure 10. Again, the channel dimension is mapped down to a size of 2 for input and output grids. For every grid, we have a vector of size [width] and a vector of size [height]. The log likelihood of every slice of the vector is taken to be the sum of the values within the slice, minus the values outside the slice. The log likelihoods for all the possible slices are then normalized to have total probability one, and the colors for every slice are given by the color logits defined in the previous paragraph. One major limitation with this technique is that if the true answer grid has a larger shape than any of the grids in the given problem grids, then our model will have zero probability of generating the correct output grid shape, and will get a wrong answer.

With the puzzle distribution now defined, we can now evaluate the log-likelihood of the observed target puzzle, to use as the reconstruction error.[12]

E Other Architectural Details

E.1 Rules for legal multitensors

1. At least one non-example dimension must be included. Examples are not special for any reason not having to do with colors, directions, rows, and columns.
2. If the width or height dimension is included, the example dimension should also be included. Positions are intrinsic to grids, which are indexed by the example dimension. Without a grid it doesn't make as much sense to talk about positions.

E.2 Weight Tying for Reflection/Rotation Symmetry

When applying a different linear layer to every tensor in a multitensor, we have a linear layer for tensors having a width but not height dimension, and another linear layer for tensors having a height but not width dimension. Whenever this is the case, we tie the weights together in order to preserve the whole network's equivariance to diagonal reflections and 90 degree rotations, which swap the width and height dimensions.

The softmax layer is not completely symmetrized because different indices of the output correspond to different combinations of dimension to softmax over. Tying the weights properly would be a bit complicated and time consuming for the performance improvement we expect, so we did not do this.

E.3 Training

We train for 2000 iterations using Adam, with learning rate 0.01, β_1 of 0.5, and β_2 of 0.9.

F Preprocessing

F.1 Output Shape Determination

The raw data consists of grids of various shapes, while the neural network operates on grids of constant shape. Most of the preprocessing that we do is aimed towards this shape inconsistency problem.

Before doing any training, we determine whether the given ARC-AGI puzzle follows three possible shape consistency rules:

[12] There are multiple slices of the same shape that result in the correct puzzle to be decoded. We sum together the probabilities of getting any of the slices by applying a logsumexp to the log probabilities. But, we found empirically that training prematurely collapses onto one particular slice. So, we pre-multiply and post-divide the log probabilities by a coefficient when applying the logsumexp. The coefficient starts at 0.1 and increases exponentially to 1 over the first 100 iterations of training. We also pre-multiply the masks by the square of this coefficient as well, to ensure they are not able to strongly concentrate on one slice too early in training.

1. The outputs in a given ARC-AGI puzzle are always the same shape as corresponding inputs.
2. All the inputs in the given ARC-AGI puzzle are the same shape.
3. All the outputs in the given ARC-AGI puzzle are the same shape.

Based on rules 1 and 3, we try to predict the shape of held-out outputs, prioritizing rule 1 over rule 3. If either rule holds, we force the postprocessing step to only consider the predicted shape by overwriting the masks produced by the linear head layer. If neither rule holds, we make a temporary prediction of the largest width and height out of the grids in the given ARC-AGI puzzle, and we allow the masks to predict shapes that are smaller than that.

The largest width and height that is given or predicted, are used as the size of the multitensor's width and height dimensions.

The predicted shapes are also used as masks when performing the multitensor communication, directional communication and directional cummax/shift layers. We did not apply masks for the other layers because of time constraints and because we do not believe it will provide for much of a performance improvement.[13]

F.2 Number of Colors

We notice that in almost all ARC-AGI puzzles, colors that are not present in the puzzle are not present in the true answers. Hence, any colors that do not appear in the puzzle are not given an index in the color dimension of the multitensor.

In addition, black is treated as a special color that is never included in the multitensor, since it normally represents the background in many puzzles. When performing color classification, a tensor of zeros is appended to the color dimension after applying the linear head, to represent logits for the black color.

G Postprocessing

Since the generated answer grid is stochastic from randomness in z, we save the answer grids throughout training, and roughly speaking, we choose the most frequently occurring one or two (or k, in the case of pass@k) as our denoised final prediction(s). This is complicated by the variable shape grids present in some puzzles.

Generally, when we sample answers from the network by taking the logits of the $[\texttt{example}, \texttt{color}, \texttt{height}, \texttt{width}, \texttt{channel}]$ tensor and argmaxxing over the color dimension, we find that the grids are noisy and will often have the wrong colors for several random pixels. We developed several methods for removing this noise:

1. Find the most commonly sampled answer.
2. Construct an exponential moving average of the output color logits before taking the softmax to produce probabilities. Also construct an exponential moving average of the masks.
3. Construct an exponential moving average of the output color probabilities after taking the softmax. Also construct an exponential moving average of the masks.

When applying these techniques, we always take the slice of highest probability given the mask, and then we take the colors of highest probability afterwards.

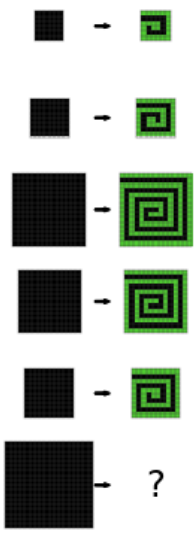
We explored several different rules for when to select which method, and arrived at a combination of 1 and 2 with a few modifications, which we selected based on best performance on exclusively training set puzzles:

- At every iteration, count up the sampled answer, as well as the exponential moving average answer (decay $= 0.97$).

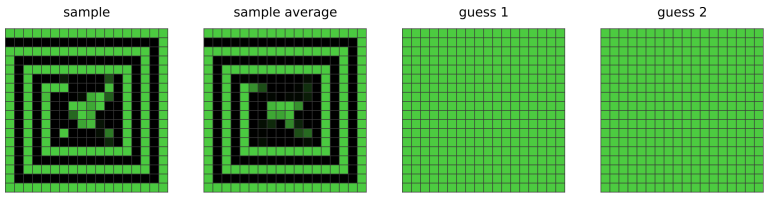
[13] The two masks for the input and output are combined together to make one mask for use in these operations, since the channel dimension in these operations don't necessarily correspond to the input and output grids.

- If before 150 iterations of training, then downweight the answer by a factor of e^{-10}. (Effectively, don't count the answer.)
- If the answer is from the exponential moving average as opposed to the sample, then downweight the answer by a factor of e^{-4}.
- Downweight the answer by a factor of $e^{-10 * \text{uncertainty}}$, where uncertainty is the average (across pixels) negative log probability assigned to the top color of every pixel.

H Empirically Observed Abilities and Disabilities of CompressARC



(a) Puzzle 28e73c20



(b) CompressARC's solution to puzzle 28e73c20

Figure 11: Puzzle 28e73c20, and CompressARC's solution to it.

A short list of abilities that **can** be performed by CompressARC includes:

- Assigning individual colors to individual procedures (see puzzle 0ca9ddb6)
- Infilling (see puzzle 0dfd9992)
- Cropping (see puzzle 1c786137)
- Connecting dots with lines, including 45 degree diagonal lines (see puzzle 1f876c06)
- Same color detection (see puzzle 1f876c06)
- Identifying pixel adjacencies (see puzzle 42a50994)
- Assigning individual colors to individual examples (see puzzle 3bd67248)
- Identifying parts of a shape (see puzzle 025d127b)
- Translation by short distances (see puzzle 025d127b)

We believe these abilities to be individually endowed by select layers in the architecture, which we designed specifically for the purpose of conferring those abilities to CompressARC.

A short list of abilities that **cannot** be performed by CompressARC includes:

- Assigning two colors to each other (see puzzle 0d3d703e)
- Repeating an operation in series many times (see puzzle 0a938d79)
- Counting/numbers (see puzzle ce9e57f2)

- Translation, rotation, reflections, rescaling, image duplication (see puzzles 0e206a2e, 5ad4f10b, and 2bcee788)
- Detecting topological properties such as connectivity (see puzzle 7b6016b9)
- Planning, simulating the behavior of an agent (see puzzle 2dd70a9a)
- Long range extensions of patterns (see puzzle 28e73c20 above)

I PUZZLE SOLVE ACCURACY TABLES

See Tables 3 and 4 for numerically reported puzzle solve accuracies on the whole dataset. Timings are also reported for solving all puzzles serially. In the actual ARC Prize evaluation environment, 12 hours are given to solve all 400 test puzzles, and more GPU resources are provided - parallelization can be applied to fit CompressARC within the time budget. Also see Figure 12 for a graph of training set accuracy as a function of the amount of inference-time learning given.

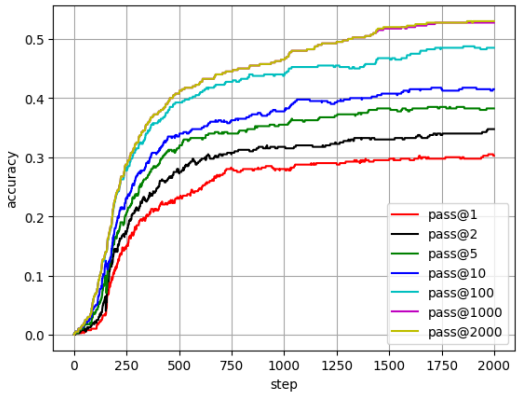


Figure 12: CompressARC's puzzle solve accuracy on the public training set of 400 puzzles rises as inference time learning progresses. Various numbers of allowed solution attempts (pass@n) for accuracy measurement are shown.

Table 3: CompressARC's puzzle solve accuracy on the training set as a function of the number of steps of inference time learning it is given, for various numbers of allowed attempts (pass@n). The official benchmark is reported with 2 allowed attempts, which is why we report 20% on the evaluation set. Total training set solve time is reported for an NVIDIA RTX 4070 GPU by solving one puzzle at a time in a sequence.

Training Iteration	Time	Pass@1	Pass@2	Pass@5	Pass@10	Pass@100	Pass@1000
100	6 h	1.00%	2.25%	3.50%	4.75%	6.75%	6.75%
200	13 h	11.50%	14.25%	16.50%	18.25%	23.25%	23.50%
300	19 h	18.50%	21.25%	23.50%	26.75%	31.50%	32.50%
400	26 h	21.00%	25.00%	28.75%	31.00%	36.00%	37.50%
500	32 h	23.00%	27.50%	31.50%	33.50%	39.25%	40.75%
750	49 h	28.00%	30.50%	34.00%	36.25%	42.75%	44.50%
1000	65 h	28.00%	31.75%	35.50%	37.75%	43.75%	46.50%
1250	81 h	29.00%	32.25%	37.00%	39.25%	45.50%	49.25%
1500	97 h	29.50%	33.00%	38.25%	40.75%	46.75%	51.75%
2000	130 h	30.25%	34.75%	38.25%	41.50%	48.50%	52.75%

Table 4: CompressARC's puzzle solve accuracy on the evaluation set, reported the same way as in Table 3.

Training Iteration	Time	Pass@1	Pass@2	Pass@5	Pass@10	Pass@100	Pass@1000
100	7 h	0.75%	1.25%	2.25%	2.50%	3.00%	3.00%
200	14 h	5.00%	6.00%	7.00%	7.75%	12.00%	12.25%
300	21 h	10.00%	10.75%	12.25%	13.25%	15.50%	16.25%
400	28 h	11.75%	13.75%	16.00%	17.00%	19.75%	20.00%
500	34 h	13.50%	15.00%	17.75%	19.25%	20.50%	21.50%
750	52 h	15.50%	17.75%	19.75%	21.50%	22.75%	25.50%
1000	69 h	16.75%	19.25%	21.75%	23.00%	26.00%	28.75%
1250	86 h	17.00%	20.75%	23.00%	24.50%	28.25%	30.75%
1500	103 h	18.25%	21.50%	24.25%	25.50%	29.50%	31.75%
2000	138 h	18.50%	20.00%	24.25%	26.00%	31.25%	33.75%

J How to Improve Our Work

At the time of release of CompressARC, there were several ideas which we thought of trying or attempted at some point, but didn't manage to get working for one reason or another. Some ideas we still believe in, but didn't use, are listed below.

J.1 Joint Compression via Weight Sharing Between Puzzles

Template Algorithm 1 includes a hard-coded value of θ for every single puzzle. We might be able to further shorten the template program length by sharing a single θ between all the puzzles, knowing that Occam's razor says a shorter program corresponds to more correct puzzle solutions. Algorithm 2 would have to be changed accordingly.

To implement this, we would most likely explore strategies like:

- Using the same network weights for all puzzles, and training for puzzles in parallel. Each puzzle gets assigned some perturbation to the weights, that is constrained in some way, e.g., LORA. (Hu et al., 2021)
- Learning a "puzzle embedding" for every puzzle that is a high dimensional vector (more than 16 dim, less than 256 dim), and learning a linear mapping from puzzle embeddings to weights for our network. This mapping serves as a basic hypernetwork, i.e., a neural network that outputs weights for another neural network. (Chauhan et al., 2024)

Unfortunately, testing this would require changing CompressARC (Algorithm 3) to run all puzzles in parallel rather than one at a time in series. This would slow down the research iteration process, which is why we did not explore this option.

J.2 Convolution-like Layers for Shape Copying Tasks

This improvement is more ARC-AGI-specific and may have less to do with AGI in our view. Many ARC-AGI-1 puzzles can be seen to involve copying shapes from one place to another, and our network has no inductive biases for such an operation. An operation which is capable of copying shapes onto multiple locations is the convolution. With one grid storing the shape and another with pixels activated at locations to copy to, convolving the two grids will produce another grid with the shape copied to the designated locations.

There are several issues with introducing a convolutional operation for the network to use. Ideally, we would read two grids via projection from the residual stream, convolve them, and write it back in via another projection, with norms in the right places and such. Ignoring the fact that the grid size changes during convolution (can be solved with two parallel networks using different grid sizes), the bigger problem is that convolutions tend to amplify noise in the grids much more than the sparse signals, so their inductive bias is not good for shape copying. We can try to apply a softmax to one

or both of the grids to reduce the noise (and to draw an interesting connection to attention), but we didn't find any success.

The last idea that we tried before discarding the idea was to modify the functional form of the convolution:

$$(f * g)(x) = \sum_y f(x - y)g(y)$$

to a tropical convolution (Fan et al., 2021), which we found to work well on toy puzzles, but not well enough for ARC-AGI-1 training puzzles (which is why we discarded this idea):

$$(f * g)(x) = \max_y f(x - y) + g(y)$$

Convolutions, when repeated with some grids flipped by 180 degrees, tend to create high activations at the center pixel, so sometimes it is important to zero out the center pixel to preserve the signal.

J.3 KL Floor for Posterior Collapse

We noticed during testing that crucial posterior tensors whose KL fell to zero during learning would never make a recovery and play their role in the encoding, just as in the phenomenon of mode collapse in variational autoencoders. (van den Oord et al., 2018) We believe that the KL divergence may upper bound the information content of the gradient training signal for parts of the network that process the encoded information. Thus, when a tensor in z falls to zero KL, the network stops learning to use its encoded information, and the KL is no longer incentivized to recover. If we artificially hold the KL above zero for an extended period of training, then the network may learn to make use the tensor's information, incentivizing the KL to stay above zero when released again.

We implemented a mechanism to keep the KL above a minimum threshold so that the network always learns to use that information, but we do not believe the network learns fast enough for this to be useful, as we have never seen a tensor recover before. Therefore, it might be useful to explore different ways to schedule this KL floor to start high and decay to zero, to allow learning when the KL is forced to be high, and to leave the KL unaffected later on in learning. This might cause training results to be more consistent across runs.

J.4 Regularization

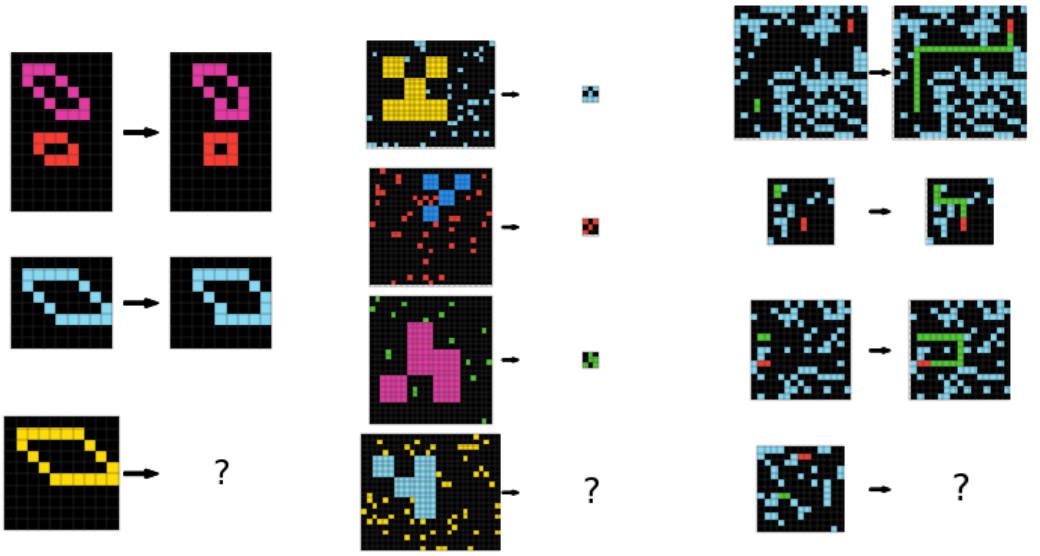
In template Algorithm 1, we do not consider reducing the description length of θ. If we were to minimize θ's description as well, this would produce an extra KL term to the loss in CompressARC (Algorithm 3), and the KL term would simplify to an L2 regularization on θ under certain reasonable limits. It is somewhat reckless for us to neglect θ in our work due to the sheer number of bits θ contributes, and making this change may improve our results.

K Additional Details about the ARC-AGI Benchmark

Hidden Rules: Figure 13 shows three examples of ARC-AGI-1 training puzzles. For every puzzle, there is a hidden rule that maps each input grid to each output grid. There are 400 training puzzles and they are easier to solve than the 400 evaluation puzzles. The training set is intended to help teach your system the following general themes which underlie the hidden rules in the evaluation set:

- **Objectness**: Objects persist and cannot appear or disappear without reason. Objects can interact or not depending on the circumstances.
- **Goal-directedness**: Objects can be animate or inanimate. Some objects are "agents" - they have intentions and they pursue goals.
- **Numbers & counting**: Objects can be counted or sorted by their shape, appearance, or movement using basic mathematics like addition, subtraction, and comparison.

- **Basic geometry & topology**: Objects can be shapes like rectangles, triangles, and circles which can be mirrored, rotated, translated, deformed, combined, repeated, etc. Differences in distances can be detected.



(a) **Hidden rule:** Shift every object right by one pixel, except the bottom/right edges of the object.

(b) **Hidden rule:** Shrink the big object and set its color to the scattered dots' color.

(c) **Hidden rule:** Extend the green line to meet the red line by turning when hitting a wall.

Figure 13: Three example ARC-AGI-1 puzzles.

Scoring: You are given some number of examples of input-to-output mappings, and you get **two attempts** to guess the output grid for a given input grid, without being told the hidden rule. If either guess is correct, then you score 1 for that puzzle, else you score 0. Some puzzles have more than one input/output pair that you have to guess, in which case the score for that puzzle may be in between.

Scoring Environment: The competitions launched by the ARC Prize Foundation have been restricted to 12 hours of compute per solution submission, in a constrained environment with no internet access. This is where a hidden semi-private evaluation set is used to score solutions. The scores we report are on the public evaluation set, which is of the same difficulty as the semi-private evaluation set, which we had no access to when we performed this work.

Comparisons and Baselines: The scores we listed for reasoning models were achieved with compute budgets well over the limits of the constrained environment. Otherwise, all other solutions we mention are scored on the semi-private evaluation set within the competition constraints.

The puzzles are designed so that **humans can reasonably find the answer, but machines should have more difficulty**. The average human can solve 76.2% of the training set, and a human expert can solve 98.5%. (LeGris et al., 2024) Current methods for solving ARC-AGI focus primarily on tokenizing the puzzles and arranging them in a sequence to prompt an LLM for a solution, or code that computes a solution. (Greenblatt, 2024) Top methods typically fine-tune on augmented training puzzles and larger alternative synthetic puzzle datasets (Li et al., 2024a) and use test-time training (Sun et al., 2020; Barbadillo, 2024). Reasoning LLMs have managed to close to 100% on the semi-private evaluation set, albeit with astronomical amounts of compute. (Chollet, 2024)

L Additional Case Studies

Below, we show two additional puzzles and a dissection of CompressARC's solution to them.

L.1 Case Study: Bounding Box

Puzzle 6d75e8bb is part of the training split, see Figure 14.

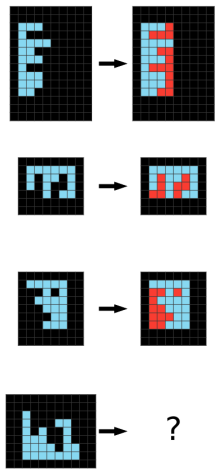


Figure 14: Bounding Box: Puzzle 6d75e8bb from the training split.

L.1.1 Watching the Network Learn: Bounding Box

Human Solution: We first realize that the input is red and black, and the output is also red and black, but some of the black pixels are replaced by light blue pixels. We see that the red shape remains unaffected. We notice that the light blue box surrounds the red shape, and finally that it is the smallest possible surrounding box that contains the red shape. At this point, we copy the input over to the answer grid, then we figure out the horizontal and vertical extent of the red shape, and color all of the non-red pixels within that extent as light blue.

CompressARC Solution: See Table 5

L.1.2 Solution Analysis: Bounding Box

Figure 15 shows the amount of contained information in every tensor within z.

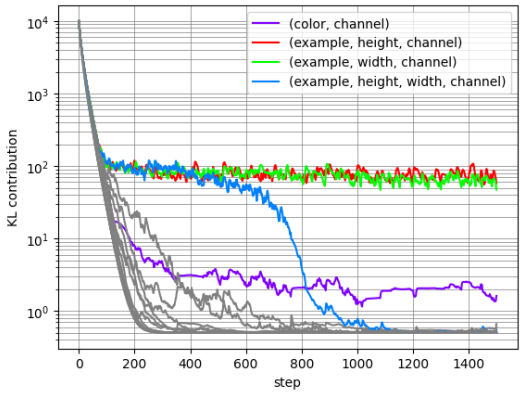
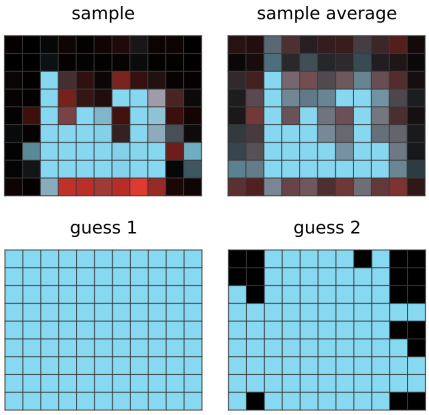
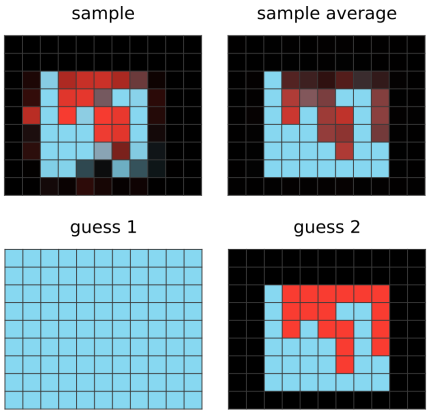
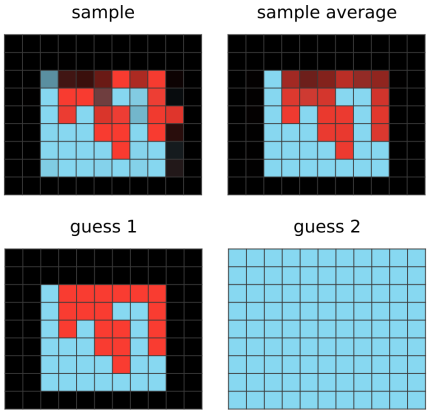


Figure 15: Breaking down the KL loss during training into contributions from each individual shaped tensor in the multitensor z.

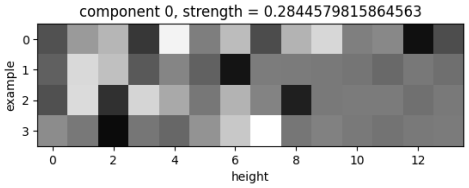
All the tensors in z fall to zero information content during training, except for three tensors. From 600-1000 steps, we see the $(\texttt{example}, \texttt{height}, \texttt{width}, \texttt{channel})$ tensor suffer a massive drop in information content, with no change in the outputted answer. We believe it was being used to identify the light blue pixels in the input, but this information then got memorized by the nonlinear portions

Table 5: CompressARC learning the solution for Bounding Box, over time.

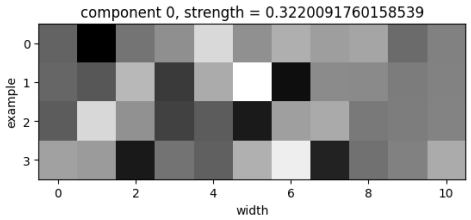
Learning steps	What is CompressARC doing?	Sampled solution guess
50	The average of sampled outputs shows that light blue pixels in the input are generally preserved in the output. However, black pixels in the input are haphazardly and randomly colored light blue and red. CompressARC does not seem to know that the colored input/output pixels lie within some kind of bounding box, or that the bounding box is the same for the input and output grids.	
100	The average of sampled outputs shows red pixels confined to an imaginary rectangle surrounding the light blue pixels. CompressARC seems to have perceived that other examples use a common bounding box for the input and output pixels, but is not completely sure about where the boundary lies and what colors go inside the box in the output. Nevertheless, guess 2 (the second most frequently sampled output) shows that the correct answer is already being sampled quite often now.	
150	The average of sampled outputs shows almost all of the pixels in the imaginary bounding box to be colored red. CompressARC has figured out the answer, and further training only refines the answer.	

of the network, using the (`example`, `height`, `channel`) and (`example`, `width`, `channel`) as positional encodings.

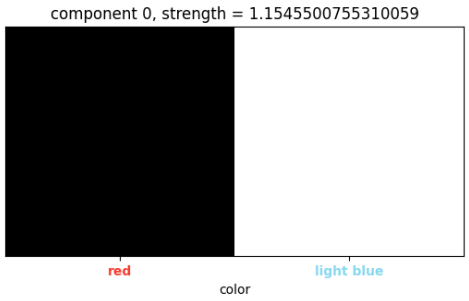
Figure 16 shows the average output of the decoding layer for these tensors to see what information is stored there.



(a) (**`example`**, **`height`**, **`channel`**) **tensor.** The first principal component is 771 times stronger than the second principal component. **A brighter pixel indicates a row with more light blue pixels.** It is unclear how CompressARC knows where the borders of the bounding box are.



(b) (**`example`**, **`width`**, **`channel`**) **tensor.** The first principal component is 550 times stronger than the second principal component. **A darker pixel indicates a column with more light blue pixels.** It is unclear how CompressARC knows where the borders of the bounding box are.



(c) (**`color`**, **`channel`**) **tensor.** This tensor serves to distinguish the roles of the two colors apart.

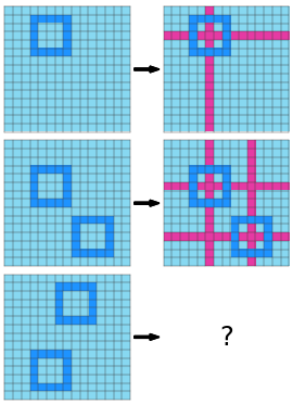
Figure 16: Mechanistic explanation of what information ends up being stored in the learned latents, and how it can be used to reconstruct the puzzle contents.

L.2 Case Study: Center Cross

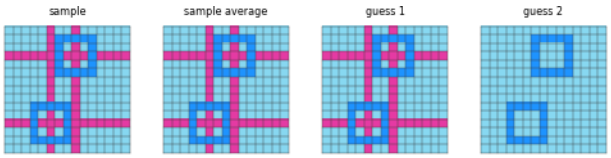
Puzzle 41e4d17e is part of the training split, see Figure 17a.

Human Solution: We first notice that the input consists of blue "bubble" shapes (really they are just squares, but the fact that they're blue reminds us of bubbles) on a light blue background and the output has the same. But in the output, there are now magenta rays emanating from the center of each bubble. We copy the input over to the answer grid, and then draw magenta rays starting from the center of each bubble out to the edge in every cardinal direction. At this point, we submit our answer and find that it is wrong, and we notice that in the given demonstrations, the blue bubble color is drawn on top of the magenta rays, and we have drawn the rays on top of the bubbles instead. So, we pick up the blue color and correct each point where a ray pierces a bubble, back to blue.

CompressARC Solution: We don't show CompressARC's solution evolving over time because we think it is uninteresting; instead will describe. We don't see much change in CompressARC's answer over time during learning. It starts by copying over the input grid, and at some point, magenta rows and columns start to appear, and they slowly settle on the correct positions. At no point does CompressARC mistakenly draw the rays on top of the bubbles; it has always had the order correct.



(a) The puzzle.



(b) CompressARC's solution.

Figure 17: Center Cross: Puzzle 41e4d17e from the training split.

L.2.1 Solution Analysis: Center Cross

Figure 18 shows another plot of the amount of information in every tensor in z. The only surviving tensors are the $(\texttt{color}, \texttt{channel})$ and $(\texttt{example}, \texttt{height}, \texttt{width}, \texttt{channel})$ tensors, which are interpreted in Figure 19.

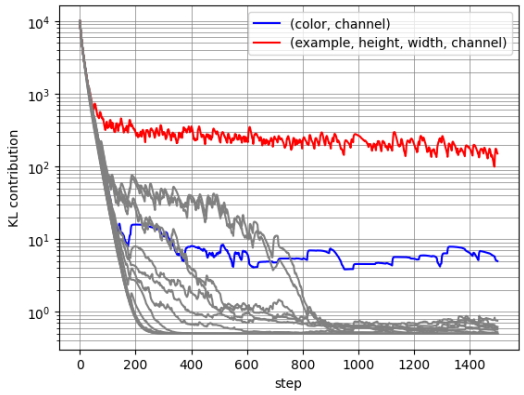
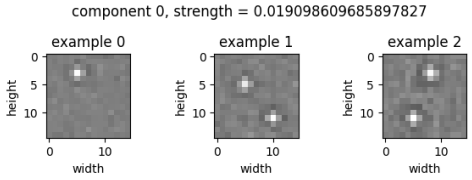
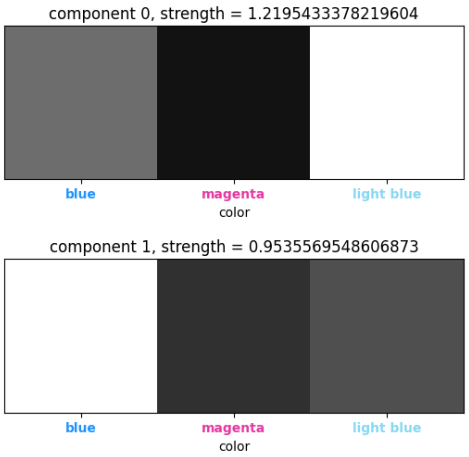


Figure 18: Breaking down the KL loss during training into contributions from each individual shaped tensor in the multitensor z.



(a) (`example`, `height`, `width`, `channel`) **tensor.** The top principal component is 2496 times stronger than the second principal component. **This tensor codes for the centers of the bubbles.** In the KL contribution plot, we can see that the information content of this tensor is decreasing over time. Likely, CompressARC is in the process of eliminating the plus shaped representation, and replacing it with a pixel instead, which takes fewer bits.

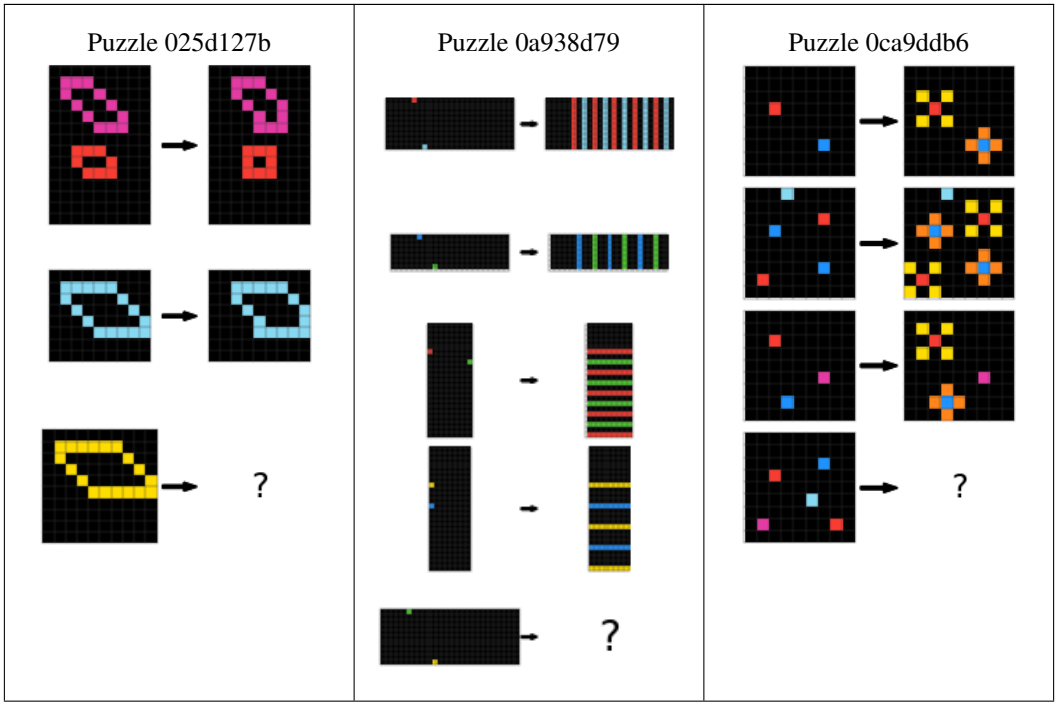


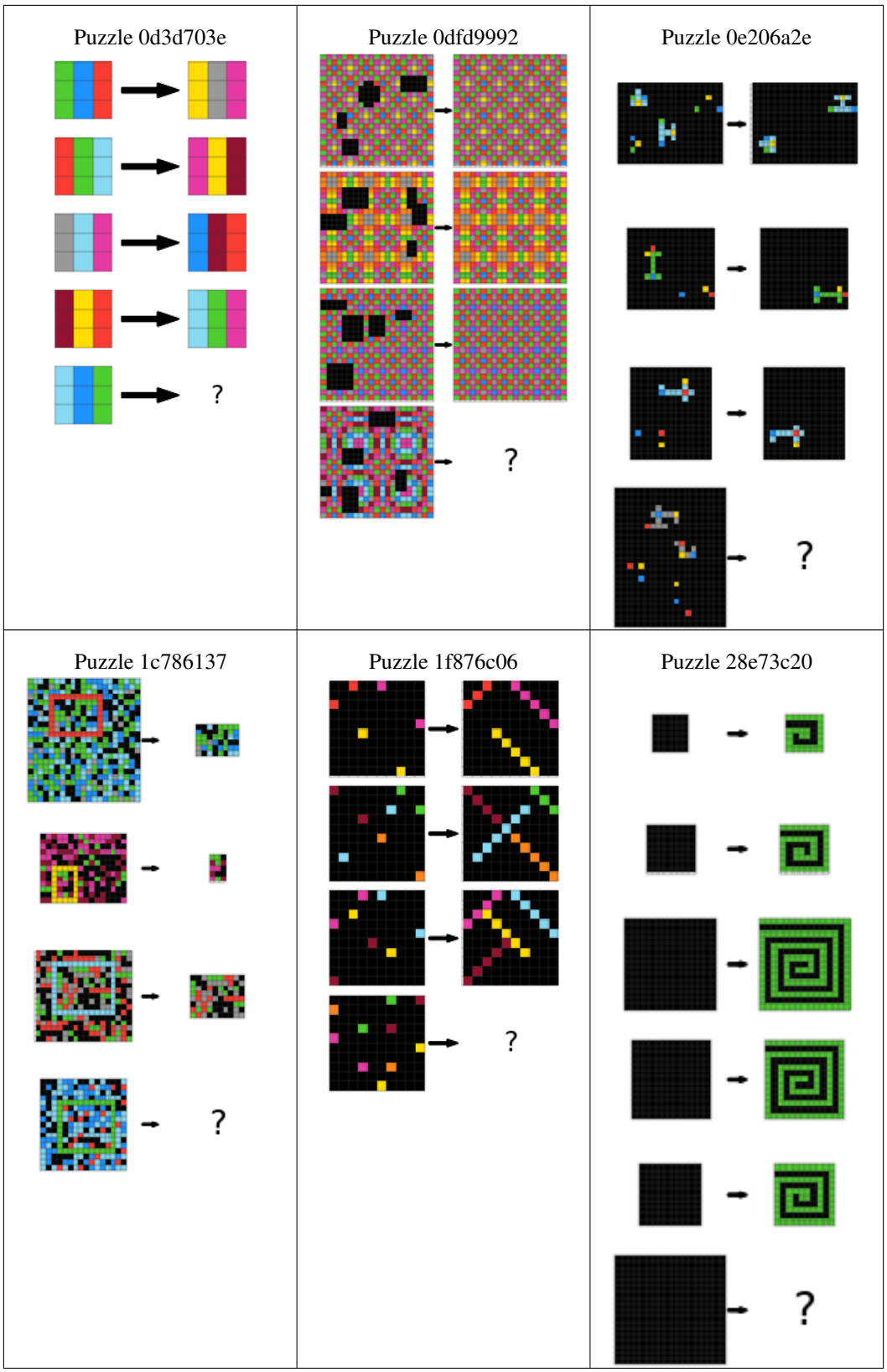
(b) (`color`, `channel`) **tensor.** This tensor just serves to distinguish the individual roles of the colors in the puzzle.

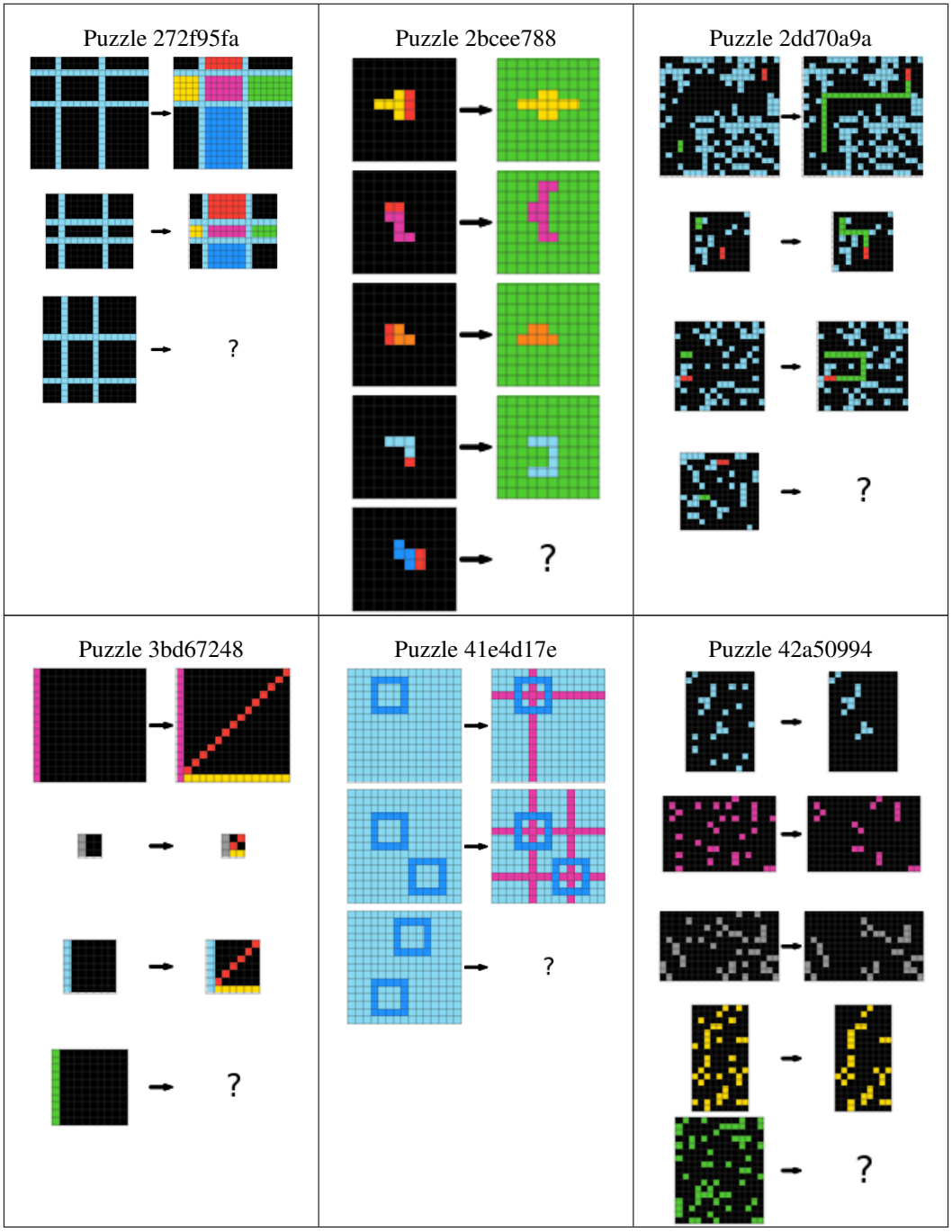
Figure 19: Mechanistic explanation of what information ends up being stored in the learned latents, and how it can be used to reconstruct the puzzle contents.

M List of Mentioned ARC-AGI-1 Puzzles

See Table 6 below.




Puzzle 0d3d703e
?
Puzzle 0dfd9992
?
Puzzle 0e206a2e
?
Puzzle 1c786137
?
Puzzle 1f876c06
?
Puzzle 28e73c20
?


Puzzle 272f95fa
?
Puzzle 2bcee788
?
Puzzle 2dd70a9a
?
Puzzle 3bd67248
?
Puzzle 41e4d17e
?
Puzzle 42a50994
?

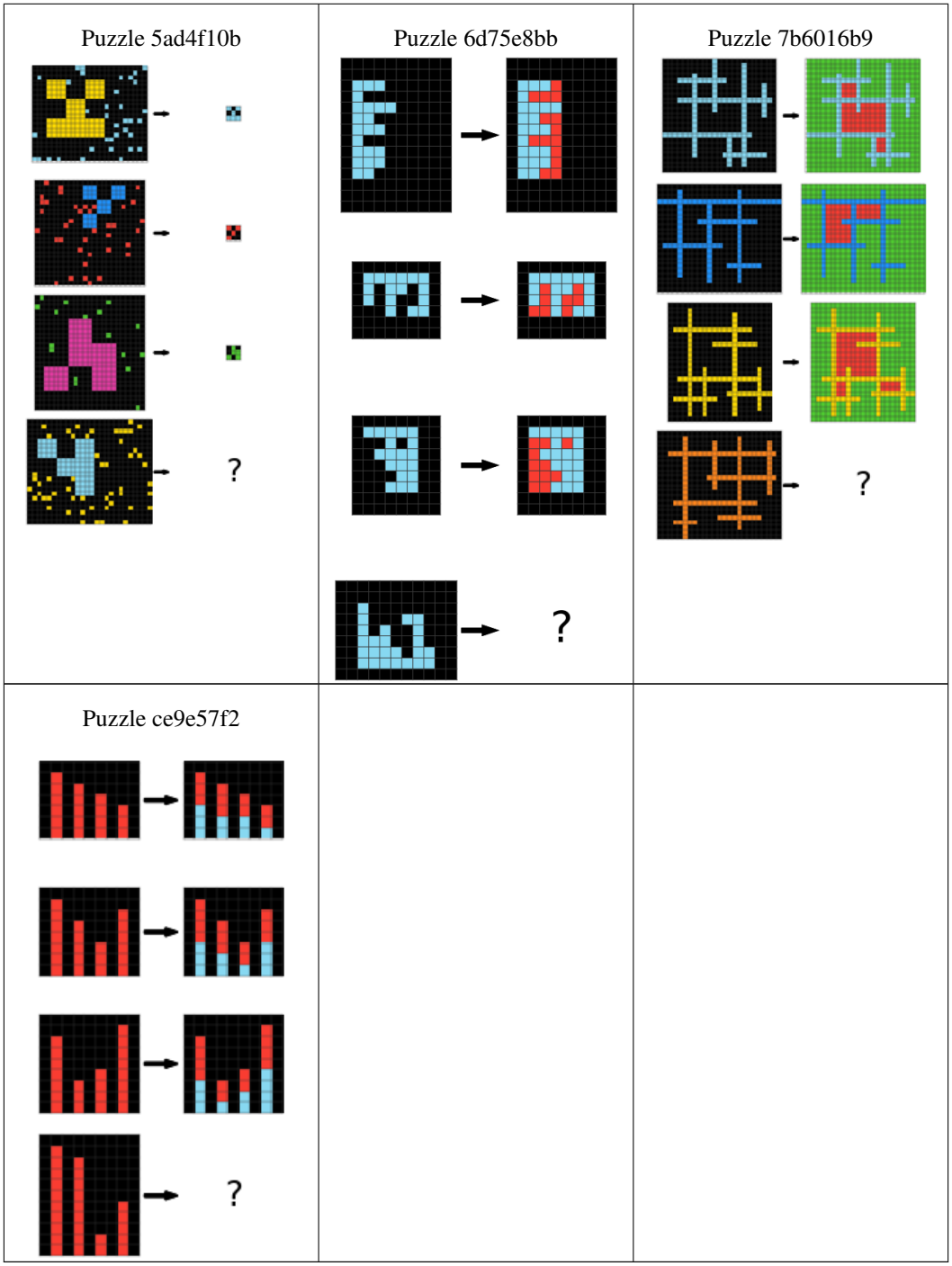


Table 6: List of Mentioned ARC-AGI=1 Puzzles. All these puzzles are part of the training split.

N CODE

Code for this project is provided in the supplemental materials.