Neural Networks for Abstraction and Reasoning

Mikel Bober-Irizar [1], Soumya Banerjee [1*]

1 University of Cambridge, Cambridge, United Kingdom

***** Corresponding author email: sb2333@cam.ac.uk

Abstract

For half a century, artificial intelligence research has attempted to reproduce the human qualities of abstraction and reasoning - creating computer systems that can learn new concepts from a minimal set of examples, in settings where humans find this easy. While specific neural networks are able to solve an impressive range of problems, **broad generalisation** to situations outside their training data has proved elusive.

In this work, we look at several novel approaches for solving the Abstraction & Reasoning Corpus (ARC). This is a dataset of abstract visual reasoning tasks introduced to test algorithms on broad generalization. Despite three international competitions with $100,000 in prizes, the best algorithms still fail to solve a majority of ARC tasks. The best solvers today rely on complex hand-crafted rules, without using machine learning at all. We revisit whether recent advances in neural networks allow progress on this task, or whether an entirely different class of models are required.

First, we adapt the DreamCoder neurosymbolic reasoning solver to ARC. DreamCoder automatically writes programs in a bespoke domain-specific language to perform reasoning, using a neural network to mimic human intuition. We present the **Perceptual Abstraction and Reasoning Language** (PeARL) language, which allows DreamCoder to solve ARC tasks, and propose a new recognition model that allows us to significantly improve on the previous best implementation.

We also propose a new encoding and augmentation scheme that allows large language models (LLMs) to solve ARC tasks, and find that the largest models can solve some ARC tasks. LLMs are able to solve a different group of problems to state-of-the-art solvers, and provide an interesting way to complement other approaches.

We perform an ensemble analysis, combining systems to achieve better results than any system alone and analysing individual strengths. However, it is sobering to see that approaches based on neural networks still lag behind existing hand-crafted solvers, and we suggest avenues for future improvements. Our findings with the ensemble model may indicate that a diversity of methods might be necessary to solve problems in ARC. Humans likely employ diverse strategies to solve ARC. Studies involving human participants to identify the strategies they employ to solve ARC could provide valuable insights for future AI approaches. Finally, we publish the `arckit` Python library to make future research on ARC easier.

1 Introduction

For the past fifty years, researchers in the field of artificial intelligence (AI) have been striving to replicate human abilities of abstraction and reasoning - developing computer systems that can learn new concepts from a small number of examples, something that humans find relatively easy [1]. While most recent AI research has focused on *narrow intelligence* (solving specific tasks given large amounts of data), we revisit whether new advances can allow computers to extrapolate to new concepts rather than merely interpolate.

This concept has been referred to as *broad generalisation* [2]: humans do this through abstraction and reasoning; drawing analogies to previous situations and thinking logically. While AI systems such as neural networks excel at a wide range of tasks, from protein folding [3] to playing Go [4], their inability to broadly generalise has precluded deployments in the real world, such as in self-driving cars. To enable safer AI, we must understand how to build systems that can reason in unusual situations.

To systematically build and evaluate computer systems that can solve abstract reasoning problems in a human-like intelligent way, we turn to a concrete benchmark. In 2019, the Abstraction and Reasoning Corpus (ARC) was introduced, as an attempt to codify a benchmark of intelligence [2] - a sort of 'IQ Test' for AI. The ARC contains a series of human-designed tasks on grids, which require learning some transformation from a small number of demonstrations. Despite three international competitions with over $100,000 in prize money, performance on ARC has proved elusive, with no single system surpassing 50% accuracy on the ARC-Easy dataset or 20% on the test set [5].

With only a handful of training examples per ARC task, and 10^{900} possible answers (of which exactly one gains credit), traditional machine learning (ML) methods that require large datasets have so far been unable to make progress. Current state-of-the-art approaches to ARC make use of complex hand-crafted algorithms based on brute force search, without harnessing any ML [6]. This work looks at whether novel machine learning systems that focus on abstraction and reasoning can achieve broad generalisation, using ARC as a focal point.

We investigate two new approaches to ARC, focusing on novel ways to incorporate neural networks to build better abstraction and reasoning solvers.

We adapt the DreamCoder algorithm, a recent state-of-the-art algorithm for program induction, to solve ARC tasks. DreamCoder writes programs in a domain-specific language to solve tasks; we design the **Perceptual Abstraction & Reasoning Language** (PeARL) pure functional domain specific language for this purpose. Our DreamCoder implementation solves 3× more tasks than existing work [7]. We introduce a new framework for solving ARC tasks using large-language models (LLMs), transforming these visual tasks into a textual domain. A detailed evaluation of three model classes [8–10] shows that LLMs can achieve competitive performance with human-crafted systems with the right augmentation and domain transformation. We build an ensemble of multiple ARC solvers that outperforms any individual system, and analyse the diversity and relative strengths of each solver.

We note however that the best systems can currently only solve about 40% of the tasks on the ARC-Hard dataset (which is much lower than human performance), meaning ARC is far from solved. Additionally, the machine-learning based systems in this work still significantly trail the state-of-the-art based on hand-crafted rule-based search [11].

Finally, we publish an open-source Python library for working with ARC to stimulate work in this field.

1.1 Background

Since the very first research on artificial intelligence, analogy-making has been considered central to the notion of intelligence. When presented with novel situations (for example; opening a new type of door, or conversing about a new topic), humans effortlessly solve these situations by creating analogies to previous experiences and concepts.

In 1967, Mikhail Bongard made one of the first attempts at identifying this notion of analogy-making in his book *Pattern Recognition* [1]. He noted how scientists such as Alan Turing have long posited the concept of a thinking machine; but while machines can be built to solve specific tasks (such as solving quadratic equations or playing chess), no progress had been made to imitate or even understand the ability of humans to adapt to new situations. Bongard goes on to suggest that pattern recognition, the ability to recognise situations into objects and classes of objects (concepts), is central to the abilities of human intelligence.

Bongard introduced a set of problems (now known as Bongard Problems) [1], where two sets of shapes are presented, and the task is to identify the common factor that differentiates them (see Supplementary Figure 1). Note that every Bongard problem has a unique transformation, and thus one cannot simply train a classification model to identify a set of transformations - the system must be able to 'invent' them. This rules out classes of models used to solve most AI tasks, which rely on large amounts of training data to spot patterns.

In Douglas Hofstadter's seminal book, *Gödel, Escher, Bach* [12], he writes that *"the skill of solving Bongard problems lies very close to the core of 'pure' intelligence, if there is such a thing"* - this work popularised the Bongard problems and proposed ideas for solving them that are still relevant today.

In the past 50 years, much work has been done to build on these problems. The original collection has been extended to almost 400 Bongard problems by a variety of authors [13]. Many attempts have been made to try and solve Bongard problems computationally, including using neural networks; however, results are still very limited [14–17].

Beyond the difficulties of the task itself, Bongard problems are not well-suited to computer evaluation (discussed further in Section 1.4), so new benchmarks have been proposed in recent years that build on Bongard's ideas. One such benchmark is the Abstraction and Reasoning Corpus.

1.2 The Abstraction and Reasoning Corpus

In 2019, Chollet published *On the Measure of Intelligence* [2], discussing past and future approaches to general artificial intelligence. Chollet notes that while excellent progress has been made in solving specific tasks to approach or surpass human-level (such as detecting cats and playing Go), these models generally require a huge amount of training and are limited to performing well on situations that they were trained on. The failure of neural network models to perform when extrapolating outside the training data has been widely explored [18].

In contrast, humans have an outstanding ability to solve tasks in highly novel situations with little training data, or indeed tasks that no human has ever solved before.

While machine learning models are often claimed to 'generalise', Chollet defines three types of generalisation: local generalisation, where a system can respond to new examples within an existing domain (for example, an image classifier generalising to a test set); broad generalisation, where a system adapts across a wider set of situations including examples which the system's creator could not have foreseen; and extreme generalisation, "adaptation to unknown unknowns across an unknown range of tasks and domains" [2].

Capturing and comparing a systems' abilities to perform these sorts of broad generalisation tasks is inherently difficult. Despite many decades of research, there is no consensus on how to measure intelligence [19], although the extensive field of psychometric testing has proposed many competing tests for humans.

The Abstraction and Reasoning Corpus (ARC), introduced by Chollet in [2], attempts to provide a benchmark for broad generalisation. By formalising a concrete way to measure generalisation ability, the hope is to foster progress in much the same way as ImageNet transformed image classification.

The ARC dataset consists of 900 hand-crafted tasks, each requiring a solver to perform abstract reasoning. In each task, the solver is first presented with some input grids (usually 3-5) and a corresponding set of output grids. Each grid contains pixels of one of 10 colours, represented by integers 0-9 and with a black 'background'. The grid size varies between each task and, indeed, within a task. Supplementary Figure 2 and Supplementary Figure 3 show some example ARC tasks.

Each task represents some common transformation from the input to output grids. A system must reason about the differences in training pairs and abstract a transformation to be applied to new input grids to produce output grids. The system is then presented with one or more test input grids, for which the system can provide up to three predictions. A task is considered solved if any of the three predictions are identical to the correct answer - no partial credit is given for close answers.

The dataset is split into three subsets: a *training set*, *evalution set* and *test set*. The training and evaluation sets each contain 400 unique tasks, and the evaluation set contains tasks that are harder than the training set. The private test set contains a further 100 tasks, which are not publicly available: to evaluate a system on the test set, a researcher must submit code to be executed on an offline system; as

a result we focus on the first two datasets.

Notably, algorithms evaluated in this work use unsupervised learning and do not train on labelled data, meaning we use both datasets exclusively for evaluation. As a result, we refer in this work to these datasets as *ARC-Easy* and *ARC-Hard* respectively.

1.3 Core Knowledge

Each task is designed to take advantage of some of four *Core Knowledge priors* [2]; by defining these explicitly, ARC tries to reduce reliance on whether an algorithm has sufficient 'acquired knowledge' and focus purely on reasoning abilities.

1. **Objectness priors:** Handling objects and their interactions. An algorithm must be able to segment the grid into objects based on space and colour (while accounting for noise and occlusion). Physical contact between objects is a common theme (e.g., 'gravity' transformations and objects growing until they hit other objects).
2. **Goal-directness prior:** Many tasks involve a general notion of 'intentionality', finding the simplest solution to some 'problem' (e.g., drawing the shortest path through a maze rather than a longer one).
3. **Numbers and Counting priors:** Some tasks involve counting and basic arithmetic (such as addition and subtraction on numbers below 10), as well as basic set manipulation (such as sorting objects based on some attribute such as size).
4. **Basic Geometry and Topology priors:** Many ARC tasks rely on geometric transformations, such as translations, rotations, shape scaling, copying objects, and drawing lines.

Some examples of these tasks are also shown in Supplementary Figure 8.

Like Bongard problems, each ARC task is essentially a *few-shot* learning problem. The high dimensionality of the output means that training traditional ML methods on the input/output pairs is impossible; only one of the 10^{900} possible output grids gains credit, with as few as three training examples.

Despite the incredibly challenging nature of the ML problem, *an average human can solve a majority of the tasks in ARC*; this highlights our ability to perform broad generalisation in a way that today's ML systems cannot, and highlights a significant gap in current AI systems.

1.4 Comparison with Bongard problems

While Bongard problems and ARC are designed to test a system or human's ability to perform inductive reasoning in the presence of few examples, there are key differences. ARC was designed for a modern machine learning paradigm. This makes it more amenable to both designing algorithms to solve it and

robustly evaluating those algorithms, compared to Bongard problems and classical psychometric tests designed for humans (such as Raven's Progressive Matrices [20]).

- **Evaluation:** How would we unambiguously determine if a system correctly identified the abstract transformation in a Bongard problem? Determining whether any analogy description is 'correct' could be subjective and time-consuming.

  In ARC, the problem is modified: the system is instead presented with a few examples of a transformation and tasked with applying the transformation to a new input. The output can then be scored algorithmically (the model was successful if it produces a pixel-perfect output).

- **Data size:** While Bongard published 100 of his problems, ARC has 900 total tasks. More tasks means both that learning systems have more training data, and allows for more precise evaluation.

  Additionally, 100 tasks are held back as a private test set: by enforcing that these tasks are unseen, one can truly test 'developer-aware' generalisation [2].

- **Problem format:** Distilling the problem of logical reasoning (rather than interpreting or generating images), ARC presents tasks as coloured pixels on a variable-size grid. This means that systems do not have to rely on a computer-vision shape detector: the problem is distilled as far as possible into one of reasoning.

These attributes make ARC an excellent testbed for ongoing research into the ability of systems to perform abstraction and reasoning. Benchmarks have led to immense progress in other areas of AI research, such as the ImageNet image classification challenge. A similar benchmark for abstraction and reasoning may accelerate research into broad generalization in machines.

1.5 Previous Work

Many attempts have been made to computationally solve ARC, primarily through the Kaggle Abstraction & Reasoning Challenge hosted in 2019 [5] with a $20,000 prize pool, where the current state-of-the-art was set by Johan Sokrates Wind (also known as Icecuber) [6]. Icecuber implements a Domain-Specific Language (DSL) with 142 handcrafted unary functions on grids. At runtime, the functions are greedily composed on the input grids, with the resulting 'pieces' stored in a directed acyclic graph (DAG). Finally, a solver combines pieces from the DAG to get as close as possible to matching the training examples. Supplementary Figure 9 shows an outline of the approach; a detailed description is available in Supplementary Materials.

Xu *et al.* introduced ARGA (Abstract Reasoning with Graph Abstractions) [21]; they extended DSL search by converting ARC grids into an object-graph representation, and operating on these representations instead. Additionally, a series of *constraints* were derived from the training examples to prune the search space, and a 'Tabu list' avoided searching primitives with poor recent performance. Overall, this approach achieved performance comparable to Icecuber limited to a small subset of tasks related to

object manipulation.

While ARC was designed to be a machine learning benchmark, the state-of-the-art solutions all rely on entirely handcrafted methods similar to Icecuber. There have been attempts to use ML, but these have been limited to small subsets of the ARC dataset. For example, Golubev *et al.* solved tasks that rely on **cropping** by extracting features from grids and training a task-specific decision tree classifier to try and predict cropping coordinates (x, y, w, h) for a task's test example [22, 23]. This approach generalised to solve 7% of private test set tasks.

In 2021, Alford *et al.* [7,24] explored the idea of applying the neurosymbolic solver DreamCoder [25] to the ARC dataset, considered much more challenging than previous DreamCoder applications. They selected a subset of 36 ARC-Easy tasks involving symmetry and basic geometric operations (such as rotations, flips and crops), and provided a DSL with 5 basic primitives on grids. They found that DreamCoder could solve some of the tasks and also successfully created new primitives. However, this solution was limited to a small subset of the ARC-Easy dataset, and did not use neural-network-guided search unlike the original neurosymbolic solver DreamCoder paper [25].

In the next section, we review the DreamCoder algorithm in detail, and then revisit whether DreamCoder can be extended to solve much more of ARC by harnessing a more powerful DSL, neural networks and other improvements.

1.6 Neurosymbolic programming with Dreamcoder

Within AI, the field of inductive programming [26] describes algorithms that derive *programs* that explain a series of examples. After 30 years of research, many algorithms and approaches have been proposed across a wide range of applications, with the research area far from being solved. In general, inductive programming provides an encouraging research direction for ARC due to its ability to massively prune the search space from "all possible grids" to just those explainable by a programmatic transformation.

The DreamCoder system [25], was a novel approach to inductive programming, which used neural networks to guide its ability to write programs (neurosymbolic programming).

The DreamCoder algorithm can be broken up into multiple phases, and can be thought of as a *wake-sleep* algorithm. During *waking* phase, a generative model writes programs in a domain-specific language (DSL) that attempt to solve tasks. In the two sleeping phases, the programming language is updated to consolidate new information learned in waking, and a separate *recognition model* is trained which learns to guide the search towards promising programs.

These phases are interleaved in several iterations to allow for self-improvement. The result is that DreamCoder can achieve remarkable performance across several domains, such as list processing, reproducing LOGO drawings, and finding regexes [25].

In this section, we provide a detailed explanation of the DreamCoder algorithm, which we later adapt in

Section 2.1.

1.6.1 Waking phase

DreamCoder, in a general form, attempts to write programs to solve tasks; we must first define what this means. We define a *task* $T(x, y) : p$ as a set of examples (x, y) defined by a program p such that $p(x) = y$. For a list processing task, (x, y) could contain pairs of unsorted and sorted lists, with the correct program p being a sorting algorithm. For tasks without input/output pairs (e.g, generating a program that reproduces a drawing), x can be empty.

The goal of DreamCoder is to inductively reason about the examples in T to produce a candidate program p' which matches the examples. It is important that tasks are *verifiable*: one can check programmatically whether a program is the correct solution to a task. In our list processing example, we can compute whether $p'(x) = y$, even if p is not known; this means that we can apply it to real-world problems where the answer is unknown, but can be verified.

DreamCoder is bootstrapped with a Domain-Specific Language known as a library. Just like any programming language, the library contains a set of functions and values, defined within a functional type-system. Making use of the type-system, we can generate a grammar that recursively defines the set of well-typed programs within the language. For example, consider the following toy language:

Types: `int, str`
Values: `1, 2, 'a', 'b'`
Functions: `add: int -> int -> int`
`concat: str -> str -> str`

In this language, `add 1 2` is a well-typed program but `add 1 'a'` is not, even though it is syntactically correct. With a type-system, one can constrain the search space of programs to valid ones: a powerful tool since the set of syntactically correct programs is commonly astronomically bigger than the set of well-typed programs.

The waking phase of DreamCoder makes great use of this grammar: for each task T, we enumerate a large number of candidate programs $p' \in L$; for each sampled program, we check if it solves the task. As there are infinite possible programs, we must use a heuristic to decide which programs to sample. DreamCoder uses the *Minimum Description Length* (MDL) principle, by computing the entropy of each program and enumerating the programs with the least entropy first. This heuristic is based on the idea that the shortest program that solves a task is the most likely to be the correct one (often compared to Occam's razor).

On its own, the waking phase can be seen as a brute-force search with a powerful and cleverly-defined search space.

1.6.2 Abstraction Sleep

The power of the DreamCoder algorithm comes from the two sleep phases, known as *abstraction sleep* and *dreaming sleep*.

Abstraction sleep considers and manipulates the solutions of tasks solved during waking. First, discovered solution programs (**frontiers**) P are inserted into a *version space*; a data structure that efficiently represents possible refactorings of programs.

The version space represents a large set of programs: for example, the functional program (+ 1 1) can be refactored as ((λ (*x*) (*x* 1 1)) +), and both of these would be included in the version space. All programs within $n = 3$ steps of refactoring are considered.

By representing all frontiers along with possible refactorings in the version space, we can pick out common concepts that occur across multiple programs as new *primitives*. These primitives can then be added to our library for the next iteration of waking, adding common concepts to the DSL (learning by analogy to existing tasks). At each iteration, the k most common concepts that cause the most reduction in total description length are compressed into new primitives.

The effect of abstraction sleep is to dramatically reduce the depth of search (at the cost of a slight increase in breadth). In practice, this abstraction learning can be extremely powerful: in a LOGO drawing task, the learned concepts included drawing polygons with n sides and l side length, or drawing a circle with r radius [25]. The concepts learnt can even include higher-order functions, such as a radial symmetry function that repeats a function multiple times at different angles. At the next iteration of waking, these concepts form longer, more powerful programs than is possible by DSL search on simple operations.

1.6.3 Dreaming Sleep

The final phase of the algorithm is *Dreaming Sleep*. In this phase, we focus on decreasing the breadth of search by intelligently guiding the generative grammar for each task. To do this, we train a neural network *recognition model*, which can directly perform abductive reasoning and infer $T(x, y) \to p$. There are two challenges to overcome in designing such a model.

First, the space of programs is exponentially large, and tasks are very difficult; even a human will often consider and discard multiple candidate hypotheses before arriving at the correct solution. Hence, a neural network which directly predicts the answer is likely to fail.

Instead, the neural network is trained to produce a grammar under which the correct solution p has low entropy. Since programs are enumerated in order of entropy (using iterative deepening), this means that a search guided by the recognition network can find the correct solution faster.

The neural network is made up of a feature extractor which converts a task to a fixed-width feature vector, and a *GrammarNet*, which takes the feature vector and produces a volume Q, where $Q_{ijk}(x)$ is the probability of primitive i being the kth argument to primitive j. From Q, we can construct a

contextual grammar which assigns likelihoods to programs, and sample from this new grammar during waking. This is shown in Supplementary Figure 4.

The second issue is that of training data: with only 800 tasks available and no labelled solutions (p) for any of them, we cannot train a complex neural network on our data. To solve this, DreamCoder uses *dreaming* to generate new training tasks (also called Helmholtz enumeration, inspired by Helmholtz machines [27]). To dream up a labelled training task, we randomly sample a program p^* from our existing probabilistic grammar, and sample some input grids x from the empirical distribution of our tasks; we can now construct a new dreamed task $T^*(x, p^*(x)) : p^*$. To train the recognition model, a constant stream of dreamed tasks and associated correct programs can be used for backpropagation.

The role of the recognition model in DreamCoder is very similar to the use of 'policy networks' in other works such as *AlphaGo* [4], which achieved superhuman performance in the board game Go. In AlphaGo, a Monte-Carlo tree search (MCTS) is used to evaluate possible positions on a board, with a policy network suggesting potentially useful moves to evaluate: the neural network acts to dramatically prune the search space and make search feasible.

In DreamCoder, the enumeration engine acts as the MCTS (evaluating programs matching a recursive grammar), while the recognition model assigns weights to the grammar such that more 'promising' programs are evaluated first. *These networks attempt to to mimic a human's ability of intuition*: when a person solves ARC tasks or plays Go, the vast majority of valid operations are immediately discarded as nonsensical.

2 Methods

2.1 Adaptation of DreamCoder

We adapt DreamCoder as an ARC solver, combining the power of DSL search with neural networks. We build on the reference Python/OCaml implementation by Ellis et al. [28], as well as the work of Alford et al. [29]. To allow DreamCoder to effectively solve ARC tasks, we introduce a new recognition model and domain-specific language (PeARL) for operating on grids, described below. Additional adaptations such as a new evaluation module and multicore search were also implemented, and are open-sourced with this work (Section 2.6). We detail the two primary modifications to the DreamCoder architecture below; Supplementary Figure 11 also gives a top-level view of the modules modified from the reference DreamCoder implementation.

2.2 ARC Recognition Model

A core component of the DreamCoder architecture is the **recognition model**. With many primitives available to our model, a brute-force search is costly and cannot go sufficiently deep to find all solutions within the search space. The recognition model is a neural network which attempts to *guide* the search,

by estimating which programs are most likely to solve a task before enumeration. It is trained during dreaming sleep, and produces a contextual grammar for each task, which assigns a high probability to the correct solution (Section 1.6.3).

The recognition model begins with a feature extractor module which converts a task to a fixed-width feature vector. This allows a different feature extractor to be used for different applications (such as LOGO drawings or list processing [25]). At the same time, a common Grammar Network learns to induce grammars for each task based on extracted features. The end-to-end network can be optimised using gradient descent by making the feature extractor differentiable.

ARC tasks have a very different format to previous applications of DreamCoder and thus necessitate a new feature extractor design. The 2D image-like nature of our grids means that we look to image recognition networks as a base, but there are several design challenges to overcome. We discuss these in turn. The final selected architecture is shown in Supplementary Figure 5.

First, the highly variable grid size precludes the use of convolutional neural networks (CNNs), which rely on a fixed-size image input, or at least a minimum-size input when adaptive or global pooling layers are used [30, 31]. In our case, we need a network that can effectively operate down to 1×1 grids.

One option is to pad all grids to a sufficiently large fixed size, such as 30×30. However, this risks hurting performance on tiny grids when most of the image is padding (most inputs are much smaller than 30×30). Our network is therefore inspired by Fully Convolutional Networks (FCNs) [32], which remove these limitations. We use a series of convolutional layers; each layer has internal padding such that its output size equals its input size.

Instead of downsampling operations usually employed by CNNs and FCNs to extract larger-scale features (which enforce a minimum input size), we use dilated convolutions [33] in later layers of our network. Dilated convolutions leave gaps between sampled pixels; the resulting behaviour is similar to a convolution on a downsampled image. These layers enable multiple scales of contextual information to be incorporated without the parameter explosion associated with large convolutions. The outputs of the dilated convolutions are added to the feature vectors as residuals, meaning the network can choose not to use them (*e.g.* on small grids).

Another unique attribute of ARC is that we have both input and output grids; rather than extracting features from a single grid, the solution to a task depends on the *relationship* between two grids of potentially different sizes. To generate a single set of features for a *task*, we apply the network to each grid followed by an adaptive average-pooling layer to generate two $64 \times 3 \times 3$ feature maps $M(y), M(x)$. The **difference** between these two feature maps $M(y) - M(x)$ is then passed to a linear layer and averaged across training examples to create a single 256-dimension feature vector for an entire task. Using a spatially adaptive layer allows the network to detect when an object has been *translated* between the input and output grid; this was found to improve performance.

2.2.1 Model training

Due to the small model size, we perform all training on CPU. The recognition model is trained at each wake-sleep cycle on Helmholtz-sampled tasks for 360 seconds; around 3,000 random tasks. The Adam optimiser [34] optimises the sum of two loss functions: the **entropy loss**, which is the overall log-likelihood of the program given the generated grammar, and a **classification loss** that treats the grammar network as an N-classifier where N is the number of primitives and minimises binary cross-entropy. Halfway through training, we anneal the learning rate 10× to improve convergence.

Initially, while the model converged to a relatively low loss, its predictions were not useful on actual ARC tasks. Upon inspection, the dreamed programs were extremely complex and too ambiguous for even a human to solve the generated tasks. To combat this, the Helmholtz sampling procedure was modified to limit any sampled program to a maximum depth of 3 primitives, dramatically reducing the loss.

Supplementary Figure 12 shows the training behaviour of the recognition model. We see that when neural-network-derived grammars are used, the entropy of discovered solutions (description lengths) is reduced by about 30% compared to a prior grammar over all tasks. Solutions with lower description lengths take **exponentially less time to find**, so this improvement is dramatic. We also see that the recognition model learns to classify which primitives might be used to solve a given task. We find empirically that our recognition model reduces the number of programs written before a solution is found by approximately 10×.

2.3 The PeARL language

DreamCoder aims to perform program induction within some well-defined language, using guided search to improve efficiency. In principle, DreamCoder could be used to write programs in any language with recursive grammar (even Python).

As discussed in Section 2.5, searching for all possible Python programs (even a guided one) to discover solutions to ARC tasks would be extremely difficult and inefficient. For this reason, we instead build a *domain-specific language* (DSL) in which a much higher proportion of enumerable programs are likely to be useful. Indeed, current state-of-the-art approaches to ARC mostly involve a DSL in some capacity (see Section 1.5).

Following existing DreamCoder DSLs for other domains, we designed the **Perceptual Abstraction & Reasoning Language** (PeARL), a bespoke DSL explicitly designed to represent transformations in ARC tasks. PeARL has two constructs: *types* and *primitives*. Types represent data-types and primitives can represent either a *value* or an n-ary function.

We define the following types in PeARL:

`grid` : A 2D grid of coloured pixels. Each grid contains a size and position, where the position is the original position of that segment (e.g., if the grid has since been cropped).

`size, pos` : The size or position of a grid, represented by a tuple of integers (x, y). When a subgrid is cropped, it retains a position, which can be used for alignment when recombining subgrids.

`colour` : A pixel colour, represented by an integer 0-9, where 0 is black.

`count` : An integer. This type can be used for arithmetic on grid properties, which is useful for some tasks.

A *valid program* in PeARL is any lambda expression of type `grid -> grid` which type-checks. Following existing DreamCoder DSLs, PeARL programs can use any number of primitives, lists, and higher-order functions. The language is entirely defined by a series of Python functions (one for each primitive), where the Python type-annotations implicitly generate the corresponding grammar. For convenience, a `DSL` class was implemented, allowing a one-to-one mapping between Python definitions and PeARL, removing additional boilerplate. This enabled rapid prototyping and experimentation.

For example, here is an implementation of a subset of PeARL with two types and two primitives:

```python
Colour = NewType("Colour", int) # Colours are ints
class Grid: ... # Implementation of the Grid type is more complex

typemap = {Grid: baseType("grid"), # baseType defines a DreamCoder type
           Colour: baseType("colour")}
dsl = DSL(typemap) # Mapping of python types to DreamCoder types

@dsl.primitive
def topCol(g: Grid) -> Colour:
    return np.argmax(np.bincount(g.grid)[1:])+1

@dsl.primitive
def filterCol(g: Grid, c: Colour) -> Grid:
    filtered = g.grid.copy()
    filtered[filtered != c] = 0
    return g.newgrid(filtered)
```

The DSL is populated with types `grid, colour`, as well as two primitives:

```
topCol: grid -> colour
filterCol: grid -> colour -> grid
```

An example of a valid program in this DSL is λ`: filterCol $0 (topCol $0)`, which represents the operation "remove all colours from the grid except the most common one". The argument `$0` represents the input grid to an ARC task, with the result of the function representing the *output* grid of that task.

A given program `f` is **correct** if `f(x) = y` for all (x, y) pairs in a task (such programs are known as **frontiers** for a given task). Hence, the *goal of DreamCoder* is to find frontiers for each task, which can be used to generate predictions on the test examples.

2.4 A framework for large-language models

Language models are a class of models designed to statistically model natural language, being trained to predict the next token (*e.g.* words) in a training corpus. Large language models (LLMs) are characterised by their size (containing tens of billions of parameters) and training on a vast corpus of text (usually scraped from the internet). When fine-tuned using reinforcement learning with human feedback, 'chat' models can be conditioned to respond to instructions in a conversational format [35].

Since the development of GPT-3 in 2020, LLMs have gained popularity at a seismic pace, and the ability of LLMs to write code or solve word puzzles is impressive. Researchers from Microsoft have stated that GPT-4 [9] "could reasonably be viewed as an early (yet still incomplete) version of an artificial general intelligence (AGI) system", and found that it could solve some reasoning tasks.

Further, work from Webb et al. [36] has recently suggested that "large language models such as GPT-3 have acquired an emergent ability to find zero-shot solutions to a broad range of analogy problems". To demonstrate this, they created a digit matrices problem set inspired by Raven's Progressive Matrices. Each problem is governed by a set of transition rules which can be stacked up to depth 3, and they find that GPT-3 performance largely surpasses human performance.

Many researchers also reject these claims, arguing that these examples do not accurately assess reasoning, that LLMs cannot even solve the letter-string analogies attacked by the Copycat computer program 30 years ago [37, 38], and that they still have "a poor grasp of reality" [39].

Given this debate, it is important to evaluate these models and whether they can already be used to solve ARC, which presents a formidable challenge compared to existing reasoning tasks such as in [36]. To do this, we conduct experiments on several state-of-the-art LLMs by a new scheme to translate ARC tasks into a textual domain. We then evaluated them in the same framework as our DreamCoder solution and existing work.

2.4.1 Experimental Setup

We evaluate several large language models (LLMs) on a common testing framework, designed to produce a fair comparison with existing ARC solutions. We test the OpenAI GPT series of models through their API, as well as Meta's LLaMA model series. For LLaMA models, we use GPTQ [40] to quantise the models to 4-bit precision, using a group size of 128. This can degrade performance on standard benchmarks but allows even the largest models to be executed on an NVIDIA A100 GPU.

To encode ARC tasks in a textual format, we convert each grid into integer digits (representing colours), with newlines delimiting rows in the grid. The format is selected to match the tokenisation used by each LLM [41], which means that each grid cell corresponds to one token in the model. This is shown in Supplementary Figure 10. The LLM is fed a prompt explaining the problem, then the inputs and outputs for the training tasks, and the final grid representing the test task must be completed. Some LLMs are fine-tuned for a chat interface rather than text completion; for these models, we represent the input

and output grids as a series of user/assistant messages with an overall system message describing the problem. Note that only problems that fit in 2048 tokens (the context window for most models tested) were attempted; in practice, we find that the largest problems are very difficult for LLMs and so this is likely to only marginally decrease performance. Full details of tokenisation and prompting are available in Supplementary Materials.

To combat the disadvantage that 1D sequence models would necessarily have on a 2D task format, we *augment* each task with a transposed version and a 90-degree rotated version. This allows the LLM to attempt each task in a row-major and column-major format, dramatically improving overall performance.

2.5 Understanding error cases in ARC

Existing approaches to ARC broadly rely on the principle of program induction: generally, a DSL defines the space of possible solutions that an algorithm could ever produce, while some search process tries to induce programs. While DreamCoders model these two components explicitly, the principle applies to almost all current attempts to solve ARC, and similar reasoning algorithms such as Copycat [38].

Furthermore, when a person attempts an ARC task, we see a similar process: one tries to abstract the training tasks to a 'program' (generally in natural language in one's head, for example "rotate by 90 degrees"); human intuition is the search procedure.

To guide our search for improved algorithms, we design a framework for analysing the shortcomings of existing algorithms. In the context of a program induction algorithm, we propose three classes of failure cases:

- **Class 1:** The algorithm did not find a solution *because the solution was not in the search space.* To resolve this, we need to increase the power of the algorithm by expanding the types of operations it knows (or can construct).
- **Class 2:** The algorithm did not find a solution, *even though it was in the search space.* A prolonged search could have found the correct answer, but computational constraints meant it did not. In this case, we could increase available computation or find a way to guide the search towards promising avenues.
- **Class 3:** The algorithm found a candidate solution, *but it did not generalise.* This is characterised by the finding of some rules that solve the training examples but yield an incorrect answer on the test example - essentially a false positive. This is perhaps the hardest case, as it cannot be solved by making the algorithm more complete - instead, we need to either reduce the search or design some notion of how plausible a given solution is so that they can be ranked.

As an extreme example, we can consider an algorithm which enumerates random Python code to solve ARC tasks. This approach can never have a Class 1 error, because all ARC tasks can be solved in Python. However, it is likely to have many Class 3 errors (e.g., generating programs which simply return

the training answers). In practice, we would encounter Class 2 errors because current computers could not hope to enumerate enough programs to find a correct one 'by chance'.

Thus, while in principle a search algorithm sufficiently powerful to solve ARC tasks is easy to design (much like making a Turing-complete programming language is easy) - *the difficulty is how to effectively prune and direct a search* such that it "searches in the right direction" and can solve tasks in a reasonable time. This perhaps approximates something close to human intuition; when presented with an ARC task, a human does not mechanically attempt all possible transformations.

2.6 Software

Throughout this work, the ability to quickly analyse and iterate through ARC tasks is instrumental. To this end, we have built the `arckit` python library, which contains tools to load and manage ARC and evaluate different algorithms. The library comes bundled with the ARC dataset, which can be rapidly loaded in a single line:

```
easy_set, hard_set = arckit.load_data()
```

Each dataset includes 400 `Task`s in a `TaskSet`, allowing them to be looked up by either index or ID. The `Task` class implements a number of features, such as scoring solutions and generating prompts for LLMs.

A primary strength of ARCkit is fast visualisation to understand the dataset and where algorithms fail; ARCkit can automatically produce vector graphics showing grids or tasks, exported in a PDF or SVG format, arranging grids to fit a specified figure size. This functionality was heavily used to generate the figures in this work. Furthermore, any task can be visualised in an interface using the `arctask` command or within Python.

The ARCkit library can be installed with the command `pip install arckit`, and documentation is available at `https://github.com/mxbi/arckit`. Additionally, we make our code for our DreamCoder implementation available from the following repository: `https://github.com/mxbi/dreamcoder-arc`.

3 Results

3.1 Primitive design

The design of primitives for PeARL (Section 2.3) has a big effect on the system's performance and is perhaps the most critical implementation detail. If primitives are too specialised or insufficiently powerful, tasks can be unsolvable. Conversely, if primitives are too elemental or there are too many non-useful primitives, the solutions may be too complex to find by search. Our design must balance these aspects for optimal performance (balancing Type 1 and Type 2 errors as described in Section 2.5).

In existing applications of DreamCoder, the authors used relatively elemental primitives, highlighting the ability of abstraction sleep to discover higher-order behaviour [25]. For example, given list processing

tasks (such as sorting a list or getting the maximum element), basic functional constructs such as `map, fold, cons, >` were defined, from which DreamCoder was able to learn essential constructs like `maximum` and eventually `sort`.

Given the difficulty of ARC tasks, which could limit abstraction sleep's ability to discover operations, a wide range of primitives were implemented in PeARL. To construct the set of primitives in PeARL, we began by implementing basic operations such as rotations, symmetry and composition based on categorising some ARC training tasks in a taxonomy (see Supplementary Figure 8). To supplement this, we ported a number of primitives from Icecuber, which provide a good starting point for 'powerful' primitives such as `gravity`, which are likely to encode core knowledge priors (Section 1.3). Most of the primitives are at least loosely based on ones in Icecuber. In addition, we make use of the functional nature of PeARL to add primitives such as list processing (`mklist, cons`) and higher order functions (`mapSplit8`) which allow DreamCoder to build new concepts during abstraction sleep.

PeARL has 81 unique primitives; below, we detail some broad categories. The full list is given in Supplementary Materials.

Rigid transformation & Cropping Arbitrary rotations, flips and transpositions are all available, as well as cropping and uncropping operations. Grids can be sliced along an axis (for example, `right_half`), and can be repeated or mirrored to solve symmetry tasks (these primitives are inspired by Alford et al. [7]).

Composition Grids can be stacked in an arbitrary order, optionally considering original positions. Pixelwise operations can operate on pairs of grids (e.g., `pixelwise_and`).

Object manipulation A grid can be split into a `list[grid]` in 5 distinct ways: 4/8-connectedness and based on rows, columns and colours. PeARL can apply an arbitrary function to each object in an image with the higher-order function `mapSplit8`. Gravity can be simulated in a 'Tetris-like' manner, and lines can be drawn between objects.

Inspired by Icecuber, a single object can be selected based on many attributes: size, frequency, density, colour and position. Lists of objects can be composed in order of size (`composeGrowing`), and PeARL can define and extend lists functionally (`mklist, lcons`).

Colour manipulation PeARL can erase, filter and remap colours: both targeting a specific colour (`c{1-9}`), as well as dynamically (`{top,rarest}colour`).

Morphology PeARL can draw borders around objects, fill holes inside objects, compress blank spaces and repeated rows/columns.

Counting PeARL can count colours, pixels or objects in a grid (`count{Pixels,Colours,Components}`), and use counts to construct new grids of a specified size (`countTo{X,Y,XY}`).

We note that there is likely to be significant improvement possible from a more advanced or more optimized set of primitives; we did not iterate on the selection in this work.

3.2 Evaluating DreamCoder

To evaluate our DreamCoder solution, we perform two experiments, one on ARC-Easy and one on ARC-Hard. We run DreamCoder for a single wake-sleep cycle, training the recognition model for 40 minutes and attempting each task for 1 CPU-hour.

Overall, DreamCoder enumerated 3.1 billion programs for 800 tasks, discovering frontiers for 75 and 23 tasks in ARC-Easy and ARC-Hard respectively. Of these frontiers, *70 and 18 tasks were solved respectively*, with a Class 3 error rate (false positive solutions – see Section 2.5) of 1.25%. These results improve on those achieved to date by DreamCoder (23 and 4 tasks solved by Alford et al. [7]).

Overall, our implementation of DreamCoder achieves 16.5% accuracy on ARC-Easy and 4.5% on ARC-Hard, improving on other recent systems such as ARGA [21]. However, a large gap remains between DreamCoder and the best hand-crafted solutions such as Icecuber; we discuss this further in Section 3.7.

Figure 1 shows three examples of PeARL solutions written by DreamCoder. DreamCoder can effectively use higher-order functions, types and combine primitives in interesting ways to discover new functionality. In the next section, we look at whether this performance would be achievable with brute force DSL search over PeARL, or whether dreaming and abstraction sleep bring improvements.

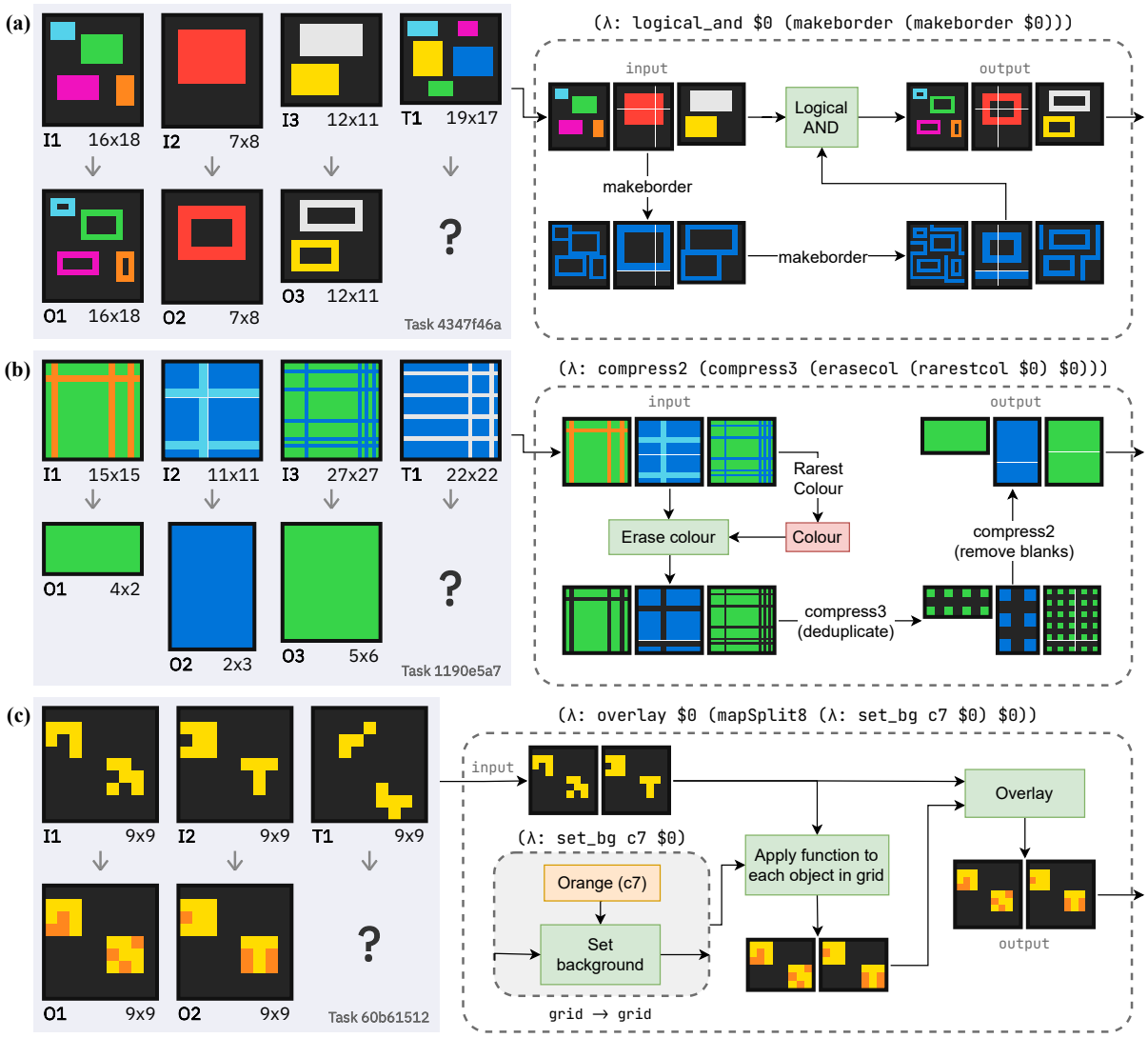


Figure 1. Three ARC tasks solved by DreamCoder (left), with the PeARL programs it generated to solve them illustrated as a computation graph (right), with intermediate outputs shown. **(a)** DreamCoder has no primitive to get the perimeter of objects, so it draws a border around it twice and takes the intersection with the original image. **(b)** Our type-system allows DreamCoder to operate on colours within images. **(c)** DreamCoder can use higher-order functions to apply arbitrary operations to each object in the image.

3.3 Ablation Studies

The DreamCoder algorithm relies on both a good *static* DSL and dynamic learning through abstraction and dreaming sleep. We conducted an ablation study to understand the impact of each learning phase.

We ran three experiments: (i) with the recognition model disabled, (ii) with a simpler context-free

recognition model that only models the probability of each primitive, and (iii) with the full contextual recognition model. The context-free model predicts the probability of individual primitives $f(\cdot)$ appearing in the solution, while the contextual recognition model also predicts the probability of every primitive within every context such as $g(\cdot, f(\cdot))$. We allowed 760 CPU-minutes of recognition model training, and ran enumeration for both 1 CPU-minute and 5 CPU-minutes per task. As the recognition model allows us to find solutions earlier in the search space (by suggesting promising candidates), we want to see if the benefit is diminished by simply allowing the search to run for longer.

The first DreamCoder iteration shows the algorithm's behaviour *before* the abstraction sleep runs, with subsequent iterations incorporating compression.

Supplementary Figure 6 shows the effect of enabling the compression and/or recognition engine on ARC-Easy tasks solved. Compression (abstraction sleep) allows the search to solve a few more tasks by adding more powerful primitives, but the effect is smaller than expected. One explanation is that, unlike the original DreamCoder problems, PeARL was not intentionally designed to include only elementary primitives that compression could build upon, limiting the gain that composition can bring.

On the other hand, enabling the recognition model improved results, solving an additional 24 tasks. *This demonstrates the benefit that neural networks can bring to this problem.* We see that the contextual recognition model slightly outperforms the context-free; the recognition model is to some extent able to learn the relationships between primitives and use this to further narrow the search space. Finally, we see that benefit of enabling the recognition model is much greater than enumerating 5x more programs, which tells us that the recognition model is successful at narrowing down the program search space.

Taking a closer look at the behaviour of abstraction sleep, we can identify why performance sometimes decreases in later iterations. The compression engine emits new primitives combining existing functions: *these new primitives are assigned a lower entropy due to their frequent use in solutions*, and thus end up used in place of simpler counterparts that would solve the same solution - the result is that *solutions to the same tasks tend to get longer* and thus more complicated over time, with more Class 3 errors. This suggests that the Minimum Description Length (MDL) principle used for ranking solutions could be modified to improve performance.

3.3.1 PeARL Ablation

The PeARL domain-specific language used in this work contains 81 primitives (full list in Supplementary Material), which DreamCoder can compose to produce solutions. Some of these primitives are quite powerful, with operations that operate on lists and functions (Section 3.1). To understand the effect of these, we analysed the frequency with which DreamCoder used each of the primitives in the 88 discovered solutions across ARC-Easy and ARC-Hard. Overall, 58 of the 81 primitives were used in solutions (a total of 231 times), whereas 23 primitives were never used. Section 3.3.1 shows the most important primitives, with the full counts given in Supplementary Material.

Primitive	Description	Count
`overlay` (G → G → G)	Overlay two grids transparently. If same size, ignore position	19
`ic_compress2` (G → G)	Remove adjacent duplicate rows and columns	14
`mirrorX` (G → G)	Horizontally mirror grid [abc]→[abccba]	11
`mirrorY` (G → G)	Vertically mirror grid	11
`ic_compress3` (G → G)	Remove any entirely black rows/columns	9
`ic_erasecol` (G → colour → G)	Remove pixels of specific colour	8
`ic_connectX` (G → G)	Join up any objects of the same colour horizontally	8
`ic_connectXY` (G → G)	Join up objects of same colour horizontally & vertically	7
`setcol` (colour → G → G)	Set all non-black pixels to the specified colour	7
`set_bg` (colour → G → G)	Set black pixels to the specified colour	7
`ic_splitall` (G → list[G])	Split grid based on 4-connected objects	7

Table 1. The primitives used most frequently by DreamCoder in successful solutions, along with their type signatures (where `G` denotes `grid`) and descriptions. 19 primitives have at least 5 usages, with 58 total primitives seen in 88 solutions. The base colour values `c[1-9]` were used a total of 25 times. Details for all primitives are given in Supplementary Materials.

Additionally, we performed an ablation by removing all primitives that create or operate on lists as well as higher-order functions, leaving 61 primitives remaining. This new primitive set only contained the `grid` and `colour` types. The reduced PeARL language was able to achieve 48 and 10 solved tasks on ARC-Easy and ARC-Hard respectively, overall a 35% reduction in performance.

3.4 Large language model results

We test a variety of large language models (LLMs) on ARC. Table 2 shows the results on ARC for some popular LLMs. Figure 2 shows the accuracy attained by each LLM on each ARC dataset, broken down into three accuracies: **initial** shows the proportion of tasks which are solved on a first attempt; **augmented** allows three attempts with rotated and transposed tasks following our augmentation scheme. Finally, we consider a 'size accuracy' which shows whether the LLM outputs a grid of the correct size (even if the contents are incorrect).

Several broad conclusions can be drawn from these results. First, we see a prominent performance scaling as the model size increases, especially in the LLaMA models, where progressively larger models are trained similarly. It is likely that this trend continues past currently available models, but the extent to which this continues is unclear and an open research question [43]. We also see that the ARC-Hard dataset is much more challenging, with small models solving almost no tasks.

Overall, the GPT-4 model from OpenAI massively outperforms all other LLMs, solving 21% of easy tasks and 8% of hard tasks. We see that across all models, **our augmentation can double accuracy or more** (Figure 2). The smaller GPT-3 models severely underperform, while large LLaMA models have middling performance.

[1]The size of these models are a trade secret, but GPT-4 likely exceeds 175B parameters.

Table 2. Results on ARC for 8 popular LLMs. **Init.** gives the initial 1-shot accuracy, **Aug.** allows the model three guesses using our augmentation scheme. Size acc gives the proportion of tasks in which a correct-size grid was predicted across both datasets.

			ARC-Easy/400		ARC-Hard/400		Size acc
Series	Model	Params	**Init.**	**Aug.**	**Init.**	**Aug.**	/800
Meta	LLaMA-7B [10]	6.7 B	7	13	0	1	201
	LLaMA-13B [10]	13 B	7	15	1	3	334
	LLaMA-33B [10]	33 B	11	24	3	9	429
	LLaMA-65B [10]	65 B	18	37	5	13	408
OpenAI Complete	GPT-3 Babbage [8]	1.3 B	2	5	0	0	42
	GPT-3 Curie [8]	6.7 B	1	8	2	2	85
OpenAI Chat	GPT-3.5-turbo [42]	-[1]	22	40	4	13	538
	GPT-4 [9]	-[1]	59	85	15	32	567

Finally, we see that the size accuracy gives an easier metric that correlates with overall performance: models that solve very few tasks can be compared by their size accuracy. Figures 3 and 4 qualitatively show the types of errors that underperforming LLMs make.

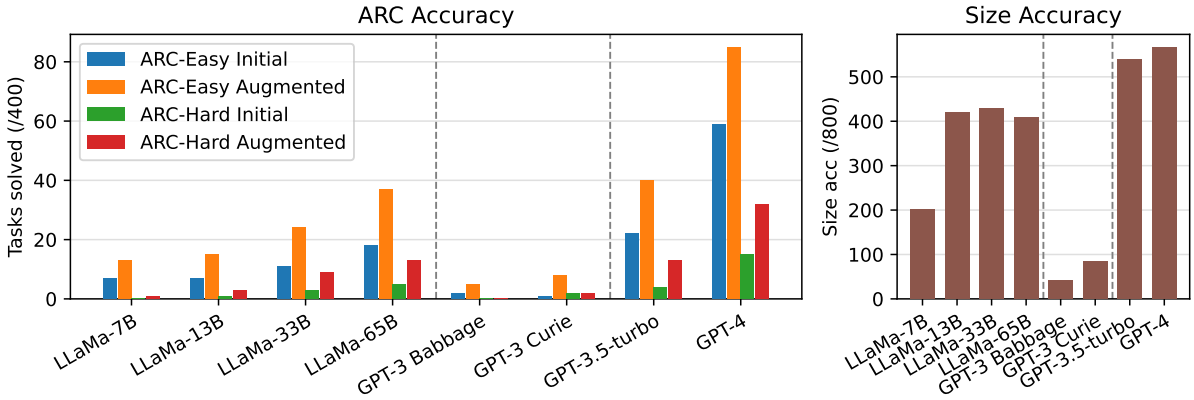


Figure 2. ARC performance of each large language model (LLM), as in Table 2. The number of correctly solved tasks is given for ARC-Easy and ARC-Hard, both on the original task and our 3-shot augmentation scheme. Size accuracy gives the number of tasks for which the LLM was able to output any grid of the correct size across both datasets.

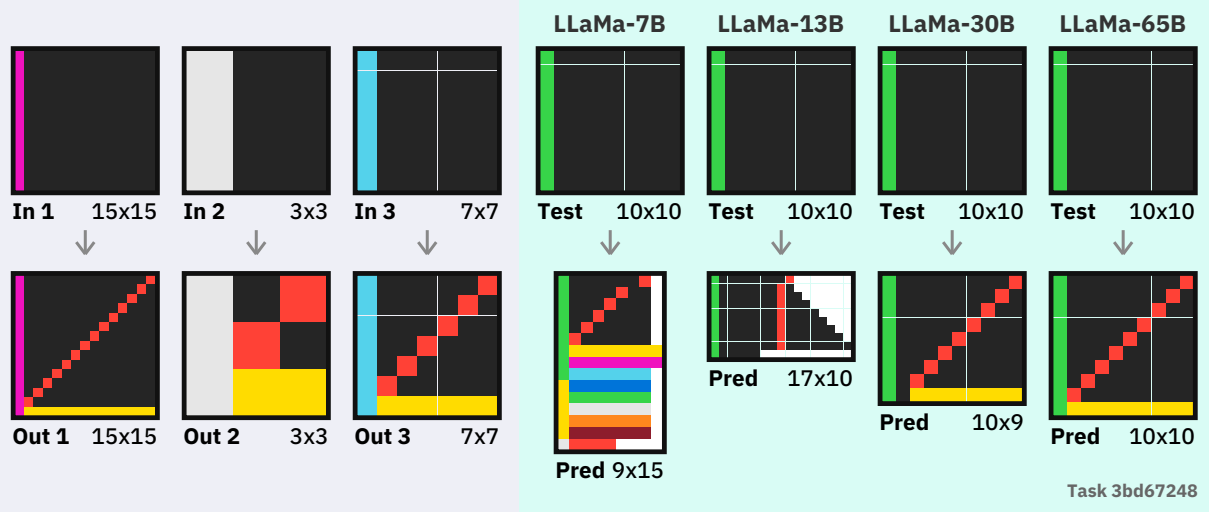


Figure 3. An example task solved by the large-language model LLaMA-65B but not by smaller models; training examples are on the left with predicted answers on the right. We see the smallest two models struggle to produce a coherent answer. The 30B model is almost correct (skipping a row and one yellow pixel), and the largest model solves the task perfectly.

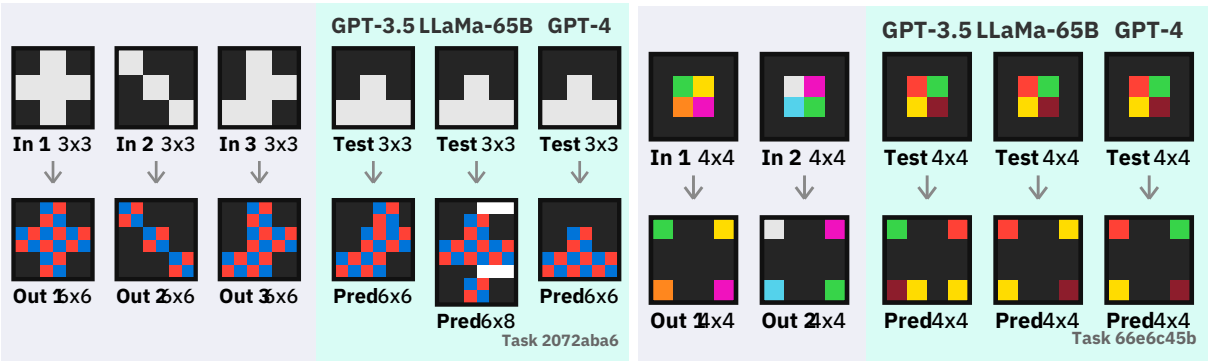


Figure 4. Two ARC-Hard tasks where only GPT-4 produced a correct solution. While the other LLMs often produce somewhat close matches on first inspection, an exactly correct solution is required to solve a task.

Overall, we see that with our encoding and augmentation scheme, GPT-4 achieves slightly better accuracy than our DreamCoder solution (see section 3.2).

3.4.1 Applicability of LLMs to ARC problems

One fair criticism of evaluating LLM abilities on ARC is that these models are designed to be text-based; they read a stream of tokens. However, we argue that this still provides a useful comparison point: there is no known equivalent to LLMs that could support reasoning with visual input and output data. This

is because LLMs derive their abilities from the huge diversity of tasks displayed in their training data (often encompassing a trillion tokens of code, scientific papers, forums, etc.), and performance has been shown to be very dependent on training data [10]. In the visual domain, equivalent datasets that display broad human reasoning capabilities are much harder to collect, although some promising work on Large Vision Models has shown abilities on other reasoning tasks [44].

There are several existing approaches that could potentially improve LLM performance. Prompt engineering involves manipulating the prompt format to improve performance on downstream tasks, and can have a large performance gap (the extent to which this could be done was limited by cost considerations). Additionally, we only evaluate the *zero-shot* performance of LLMs; it is possible that advances such as Low-Rank Adaptation (LoRA) fine-tuning of models [45] could be harnessed to condition an LLM to perform better on ARC-specific tasks.

LLMs can be used to propose hypotheses that can then be refined by task-specific solvers [46]. This has recently been applied to a smaller version of ARC called MiniARC [46]. Several other approaches have been suggested, such as learning rules in multiple stages using LLMs [47], using a multi-agent LLM system [48] or representing ARC problems as graphs and using these as a prompt to an LLM [49].

Textual descriptions of how humans solve problems in ARC could also inform future machine learning approaches [50, 51]. Studying how humans solve problems in ARC and communicate these solutions to each other in natural language [50] may help us design better AI approaches to solve ARC. Accurate reconstructions of of how humans solve problems in ARC [52, 53] can be used to train models [54], which may also open up new approaches to potentially augment human performance with LLMs. Some have suggested that LLMs may be capable of deductive, inductive and abductive learning [55], which is promising.

We discuss some of these further in Section 4.

3.5 Ensemble methods

In this work, we evaluate some very different systems as ARC solvers. One interesting question arises when looking at the differences between these systems: are they solving the same tasks, or does each system have different strengths?

Machine learning research has looked at ensemble approaches, where multiple models are combined to produce predictions better than any single model. The success of these approaches relies on *diversity* of models [56], often measured by looking at the correlation of predictions. More diverse models produce more powerful ensembles, even with disparate individual accuracies.

Since we cannot measure correlation between predictions directly, we propose two approaches to measure the similarity between systems attempting ARC, even when models have very different headline performance. First, the overlap between the set of tasks solved by each system (the Szymkiewicz–Simpson

coefficient [57])

$$\text{Overlap}(A, B) = \frac{|A \cap B|}{\min(|A|, |B|)}$$

tells us the proportion of tasks in the weaker model solved by the stronger model. When one model solves a subset of the tasks of another, the overlap is 1. This can be seen as a performance-normalised similarity metric: it ignores the overall accuracy difference between the models, and lets us determine if one system is a more powerful version of another, or whether they are solving complementary types of tasks (as opposed to more common metrics such as Jaccard index). We also propose the asymmetric Gain measure,

$$\text{Gain}(A, B) = |A \cup B| - |A|,$$

which gives the additional tasks solvable by adding one model to another model - hence, it reflects the *unique skills* of each model; tasks that cannot be solved by the other model in the pair. The asymmetry means that we can see which model performs the best.

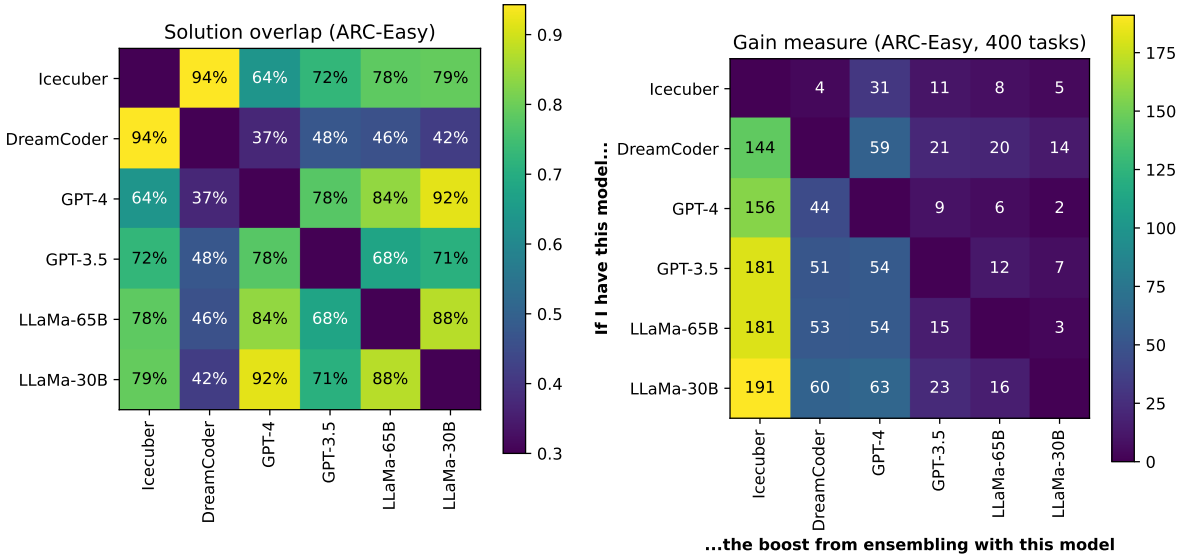


Figure 5. Comparison of the solution similarity of our systems, showing solution overlap (left panel) and gain measure (right panel) between each system: this tells us how similar the predictions are, and how much one could expect to gain by combining systems. Systems are ordered by overall performance, so the top-right triangle in the panel on the right shows the unique skills of the lesser system. The panel on the left shows that DreamCoder and GPT-4 have only 37% overlap between the tasks which they can solve.

Figure 5 shows both metrics for our systems' predictions on the ARC-Easy set, from which we can make

a few key takeaways.

First, we see that LLMs tend to solve different tasks than DSL-based solutions: the overlap between these models is relatively low. In particular, our DreamCoder and GPT-4 solutions, which have comparable accuracy independently, have only a 37% overlap between *which* tasks they can solve - this suggests an ensemble which can effectively select between approaches *could lead to an almost doubling in performance.*

Conversely, we see that different LLMs tend to have similar predictions (with larger models tending to solve a superset of problems solved by their smaller counterparts). Likewise, we see that DreamCoder and Icecuber solve similar types of task: with DreamCoder uniquely solving only 4 tasks and a 94% overlap between them.

3.6 Building a heterogenous high-performance ensemble

For classification and regression tasks, ensembles can be built effectively by a weighted average of each system's predictions [56]. In our case, with only exact solutions scoring credit, interpolating solutions would be unlikely to work well.

Instead, we build a *voting* ensemble of systems: on each task, each system can propose some output grids which they 'vote' for, added to a priority queue. When multiple systems generate the same prediction, votes are summed, increasing their priority. Since ARC allows three guesses, we select the three output grids with the most votes. Another difficulty to overcome is the heterogeneity of our systems. For example, DreamCoder only produces a prediction when it solves all training tasks: a DreamCoder prediction should be prioritised for the final list. On the other hand, when combining LLMs, we would like to provide higher weight to LLMs with better performance.

We assign distinct weights to each algorithm to build our ensembles: 20 to DreamCoder, 3 to each GPT-4 prediction, and $[8, 4, 4]$ to the three IceCuber predictions for each task (based on inspection of the above figures and minimal manual tuning). Additionally, we discard any LLM predictions which are not grids, the default prediction from Icecuber (predicted when no solution was found), and de-duplicate predictions from DreamCoder (as it tends to find equivalent programs: e.g., when functions are commutative under composition, we have $f(g(x))$ and $g(f(x))$). This process allows us to select the three most promising predictions from up to 9. The performance of our ensembles are discussed in the next section.

3.7 Quantitative results across all methods

We now take stock of each system discussed in this report, looking at headline performance on both ARC datasets. For fairness, all systems were evaluated from scratch using a common framework, following the official ARC evaluation procedure. Each system has access to training examples, and can create a maximum of three predictions for each test example. A task is solved if one of the three attempts are **exactly** correct. Figure 6 and Table 3 show these results.

Table 3. Comparison of headline performance for the systems discussed in this work; all systems have three attempts per task. Systems in **bold** were designed in this project. For the sake of brevity, DreamCoder is abbreviated to DC and Abstract Reasoning with Graph Abstractions [21] is abbreviated to ARGA.

Type	System	ARC-Easy/400	ARC-Hard/400
DSL Search (section 1.5)	Icecuber [6]	209	160
	ARGA [21]	49	10
DreamCoder (section 2.1)	Alford et al. [7]	23	2
	Ours (DC)	70	18
LLM (section 2.4)	**LLaMA-65B** [10]	24	9
	GPT-3.5 [42]	40	13
	GPT-4 [9]	85	32
Ensemble (section 3.5)	**DC + GPT-4**	129	35
	Icecuber + **DC + GPT-4**	**228**	**161**

Our new approaches GPT-4 and DreamCoder attain decent performance, improving over ARGA [21]. Notably, DreamCoder solution solves 3× and 9× more ARC-Easy and ARC-Hard tasks respectively than the existing best implementation by Alford *et al.* [7].

Our voting ensemble can combine the strengths of different algorithms, with DreamCoder and GPT-4 ensembling to solve **50% more tasks than either system alone**. Moreover, these systems successfully tackled tasks that stumped Icecuber, the previous state-of-the-art solution. Our final ensemble, which incorporates Icecuber, DreamCoder, and GPT-4, manages to solve 57% and 40% of tasks on the ARC-Easy and ARC-Hard datasets respectively.

Across the board, performance on ARC-Easy correlates well with ARC-Hard, with most systems solving $2-4$ times more easy tasks - a clear outlier here is Icecuber, which solves 160 hard tasks. This could be partially attributed to the fact that **the Icecuber DSL was built on hand-crafted solutions to 100 hard tasks** [6], meaning that the developer designed the language to be good on these tasks.

Collectively, these results show two approaches using neural networks for abstraction and reasoning. We highlight the need for diverse and complementary methods that, when combined, can lead to superior performance.

We note, however, that Icecuber, the current state-of-the-art that uses hand-crafted brute-force search, still retains a commanding lead over neural-network-based solutions.

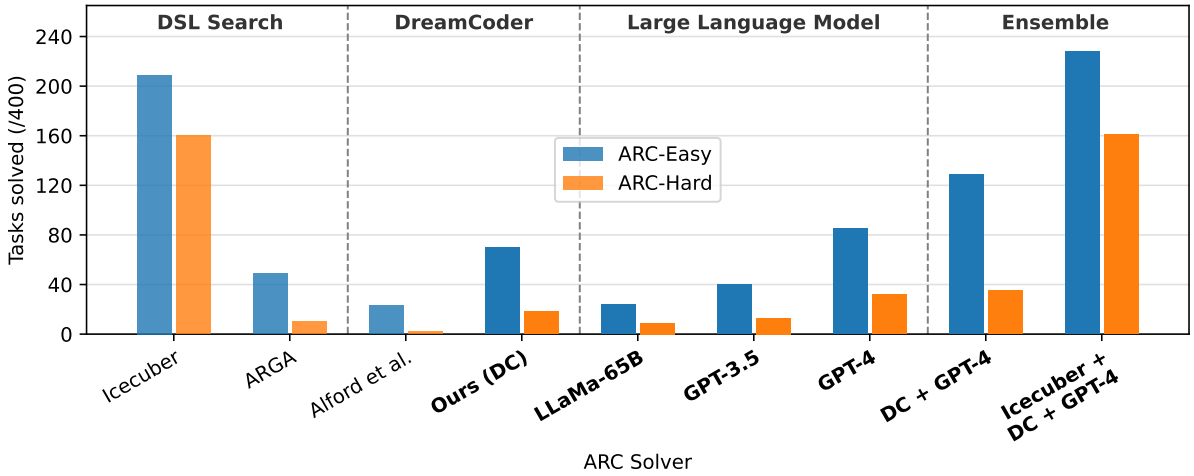


Figure 6. ARC tasks solved by each system in this work on the two available ARC datasets (see Table 3); results from this work in bold. The fully handcrafted Icecuber solver remains the best single solution, with GPT-4 obtaining slightly better performance than DreamCoder. We achieve dramatic gains by ensembling solutions. For the sake of brevity, DreamCoder is abbreviated to DC and Abstract Reasoning with Graph Abstractions [21] is abbreviated to ARGA.

4 Discussion

4.1 Overview

In this work, we have applied a modern machine learning lens to a touchstone problem in AI: searching for systems that can perform *broad generalisation* to abstraction and reasoning tasks. Overall, we find that complex human-crafted solutions based on domain specific languages (DSL) search still provide the best known single-system performance. However, we demonstrate two interesting machine-learning-based solutions, which provide a concrete research direction to make more robust learning machines.

4.2 Limitations of our approach

There are several types of ARC tasks which our DreamCoder adaptation cannot solve. For example, *copy-paste* tasks, which rely on copying a pattern from training outputs to test outputs, cannot be solved. This is because DreamCoder considers each input/output pair in isolation, and cannot copy objects from one grid to another. *In-painting* tasks can likely be solved much more effectively by a dedicated algorithm, which tries to identify the tiling or symmetry pattern required to fill in the missing pixels.

Our method for adapting large-language models to ARC is also exploratory, and there is likely to be significant gains attainable from the use of different models, different prompt techniques (for example that allow LLMs to solve problems larger than their context window), or systems that combine mechanistic reasoning with LLMs. We discuss some avenues in Section 4

Additionally, in this work we report results on the publicly available splits of the ARC dataset (referred

to as ARC-Easy and ARC-Hard), as opposed to the private test set hosted on Kaggle [5]. This creates a risk that the hand-crafted DSL (or indeed, training data for an LLM) is biased towards the training set, and so is a significant limitation of the evaluation in this work. The evaluation environment for the private test set has a strict CPU-time limit as well as no internet access, which would have precluded running DreamCoder or any of the LLMs. In the future, we would like to see a system of improved access to this dataset with more compute availability so that more reliable numbers can be reported.

4.2.1 Qualitative comparison of methods

We conducted an informal analysis of the weaknesses of each algorithm by looking at task successes and failures with reference to the taxonomy in Supplementary Figure 8. Broadly, we found that Icecuber and DreamCoder had very similar strengths and weaknesses: they were very good at solving tasks involving objects, gravity, symmetry and other tasks which solvable within the DSL, even if the solution involved composing. The DreamCoder DSL is more narrow than that of Icecuber, and so many of the tasks solved by Icecuber were simply not possible in the DreamCoder DSL, suggesting that DSL design is extremely important for program-synthesis approaches. There were a few cases of DreamCoder solving tasks that should have been solvable by Icecuber given sufficient search depth, suggesting that a guided search is useful. DSL-based approaches are unable to solve tasks such as those require copying objects or colours between input and output examples (such as lookup tables), or those requiring learning a more complex pattern (in-painting).

On the other hand, LLMs were very good at tasks where the input and output are very similar (such as inpainting and denoising), and those where the sizes of grids are small. This makes sense as larger problems can be 100x longer than smaller problems, and reasoning over longer inputs can be a challenge for LLMs, as well as producing longer outputs with no mistakes. For a DSL approach, the grid size does not affect the difficulty of the problem. They also struggled at tasks that required 2D manipulation, likely due to the way that LLMs operate on 1D sequences (as discussed in Section 3.4.1). Interestingly, there were no obvious categories where LLMs were perfect or totally fail.

In Supplementary Figure 7, we provide several examples of exemplar tasks that IceCuber, DreamCoder and GPT-4 could solve respectively that the others could not.

4.3 Future work

Research on computational abstraction and reasoning is far from complete, so we consider some future directions suggested by our conclusions.

While we improve the accuracy of DreamCoder, it is still far behind the handcrafted Icecuber DSL search [6]. One future improvement would be to extend DreamCoder with the unique features of Icecuber, such as colour normalisation and, in particular, the greedy stacker, which combines intermediate solutions working backwards from the output grid. Execution-guided program synthesis would be one way to

achieve a similar effect, where intermediate solutions are scored based on their similarity to the final output and used to guide the search, replacing all-or-nothing evaluation.

There are several research avenues that could improve LLM performance. We only evaluate the in-context learning performance of LLMs; advances such as low-rank adaptation fine-tuning [45] could precondition a model further on ARC priors. Additionally, chain-of-thought prompting and similar techniques, where the LLM is first required to explain the training examples before applying it to the test example, may also improve performance [58, 59].

Early experiments have been made with Large Vision Models [44] which suggest that visual datasets are sufficient to train a model with emergent reasoning abilities that could be applied to ARC. While these models warrant investigation, challenges are likely to include the requirement for pixel-perfect outputs as well as collecting a suitable mix of training data.

Finally, an ARC2 dataset is in development by Chollet and Lab42 [60], aiming to crowd-source 5,000 tasks. ARC2 aims to be more diverse, with "no specific task type used more than once". This is likely to make ARC even harder, and require greater generalisation abilities.

4.4 Concluding remarks

Our implementation of DreamCoder achieves a 3.5× improvement in tasks solved across ARC over the previous best implementation [7,24], through a wide range of improvements, such as the PeARL language and the use of a neural-network recognition model (section 2.1).

Additionally, we introduced a framework to transform visual ARC problems into a text completion domain suitable for large language models (LLMs), and found that the largest available LLMs trained across huge corpora have sufficiently transferable abilities to begin to solve ARC problems; their accuracy can also be effectively doubled with suitable augmentation. Our results suggest that even larger models could continue to improve attainable accuracy on ARC (section 2.4). While previous solutions largely rely on human-designed priors in the form of a DSL, the allure of LLMs is that they instead use neural-networks in an end-to-end fashion, and all reasoning priors come from training data.

We also show the effectiveness of ensembling solutions: while each algorithm may specialise on a certain type of task, systems with low false-positive rates can be effectively combined (section 3.5). Rather than trying to create a system that *beats* Icecuber, a useful direction would be to try to build systems that complement the state-of-the-art, leading to a multipronged solution with better generalisation. In particular, we already find that LLMs solve many problems not suitable for domain-specific languages.

It is sobering to observe that both the previous state-of-the-art solutions and the approaches developed in this work achieve only modest accuracy on the ARC dataset (approximately 40% of tasks solved on the ARC-Hard dataset), which is still much less than what humans can solve [2, 50, 53, 61]. This suggests that current machine learning methods may not yet be powerful enough to develop complex concepts and

abstractions for the hardest tasks. Significantly more work will be required before we can consider ARC solved. Our ensemble analysis indicates that a diversity of methods might be necessary to solve problems in ARC.

We hope our work will serve as the foundation for future approaches. Neuro-symbolic methods such as DreamCoder may benefit from many improvements, such as execution-guided synthesis [62] and a stronger DSL mimicking Icecuber [11]. Likewise, there are many potential avenues to improve LLM performance, such as chain-of-thought prompting methods [58, 59], fine-tuning [45] or constrained generation. Furthermore, studying how humans solve ARC tasks [50, 63] could inform the development of future AI systems. Mimicking the diverse strategies humans use to solve ARC may prove helpful in designing next-generation abstraction and reasoning algorithms.

Finally, we see that broad generalisation is still an incredibly challenging goal for AI, even for the most advanced systems. ARC is proving to be an informative benchmark; still unsolved, and motivating research after four years, while at the same time well-defined and objectively evaluable. We hope that research on ARC will one day bring about a broad generalisation machine, in the way that MNIST and ImageNet brought about the rise of modern deep neural networks.

Declarations

Acknowledgements

We acknowledge the help and support of the Accelerate team. We are especially grateful to Neil Lawrence and Carl Henrik Ek for fruitful discussions and feedback.

Funding statement

SB acknowledges funding from the Accelerate Programme for Scientific Discovery Research Fellowship. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript. The views expressed are those of the authors and not necessarily those of the funders.

Conflicts of interests

All authors declare they have no conflicts of interest to disclose.

Ethics

No ethics approval was necessary.

Data availability

All data generated or analysed during this study are included in this published article and its supplementary information files.

Author contributions

MBI carried out the implementation and analysis, participated in the design of the study and wrote the manuscript. SB carried out the analysis, participated in the design of the study and wrote the manuscript. SB directed the study. All authors gave final approval for publication.

References

1. Bongard MM, Hawkins JK, Cheron T (1968) Pattern recognition. Spartan Books.
2. Chollet F (2019). On the measure of intelligence. ArXiv: 1911.01547.
3. Jumper J, Evans R, Pritzel A, Green T, Figurnov M, et al. (2021) Highly accurate protein structure prediction with AlphaFold. Nature 596: 583–589.
4. Silver D, Huang A, Maddison CJ, Guez A, Sifre L, et al. (2016) Mastering the game of Go with deep neural networks and tree search. Nature 529: 484–489.
5. Chollet F, Tong K, Reade W, Elliott J (2020). Abstraction and reasoning challenge. URL `https://kaggle.com/competitions/abstraction-and-reasoning-challenge`.
6. Wind JS (2022). 1st place solution + code and official documentation. Kaggle Forums. URL `https://www.kaggle.com/c/abstraction-and-reasoning-challenge/discussion/154597`.
7. Alford S (2021) A Neurosymbolic Approach to Abstraction and Reasoning. Master's thesis, Massachusetts Institute of Technology.
8. Brown TB, Mann B, Ryder N, Subbiah M, Kaplan J, et al. (2020) Language models are few-shot learners. In: Larochelle H, Ranzato M, Hadsell R, Balcan M, Lin H, editors, 33rd Annual Conference on Neural Information Processing Systems (NeurIPS).
9. OpenAI (2023). GPT-4 technical report. ArXiv:2303.08774.
10. Touvron H, Lavril T, Izacard G, Martinet X, Lachaux MA, et al. (2023). LLaMA: Open and efficient foundation language models. ArXiv:2302.13971.
11. Wind JS. DSL solution to the ARC challenge. URL `https://github.com/top-quarks/ARC-solution`.
12. Hofstadter DR (1979) Gödel, Escher, Bach: an Eternal Golden Braid. Basic Books New York.

13. Foundalis H (2007). Index of bongard problems. URL https://www.foundalis.com/res/bps/bpidx.htm.

14. Halme A (2020). Bongard solvers. URL https://notes.fringeling.com/BongardSolvers/.

15. Kharagorgiev S (2018). Solving Bongard problems with deep learning. URL https://k10v.github.io/2018/02/25/Solving-Bongard-problems-with-deep-learning/.

16. Depeweg S, Rothkopf CA, Jäkel F (2018). Solving Bongard problems with a visual language and pragmatic reasoning. ArXiv:1804.04452.

17. Foundalis HE (2006) Phaeaco: A Cognitive Architecture Inspired by Bongard's problems. Ph.D. thesis, Indiana University.

18. Geirhos R, Temme CRM, Rauber J, Schütt HH, Bethge M, et al. (2018) Generalisation in humans and deep neural networks. In: Bengio S, Wallach HM, Larochelle H, Grauman K, Cesa-Bianchi N, et al., editors, 31st Annual Conference on Neural Information Processing Systems (NeurIPS) 2018. pp. 7549–7561.

19. Legg S, Hutter M (2007). A collection of definitions of intelligence. ArXiv:0706.3639.

20. Raven JC (1936) Mental tests used in genetic studies: The performance of related individuals on tests mainly educative and mainly reproductive. Master's thesis, University of London.

21. Xu Y, Khalil EB, Sanner S (2022). Graphs, constraints, and search for the abstraction and reasoning corpus. ArXiv:arXiv.2210.09880.

22. Golubev V (2019). 3rd place[short preview+code]. Kaggle Forums. URL https://www.kaggle.com/competitions/abstraction-and-reasoning-challenge/discussion/154305.

23. Golubev V (2019). 7 solved tasks via trees. Kaggle Kernels. URL https://www.kaggle.com/code/golubev/7-solved-tasks-via-trees/notebook.

24. Banburski A, Ghandi A, Alford S, Dandekar S, Chin P, et al. (2020) Dreaming with ARC. Technical report, Center for Brains, Minds and Machines (CBMM). URL https://dspace.mit.edu/handle/1721.1/128607.

25. Ellis K, Wong C, Nye M, Sable-Meyer M, Cary L, et al. (2020). DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. ArXiv:2006.08381.

26. Muggleton SH (1991) Inductive logic programming. New Generation Computing 8: 295–318.

27. Dayan P, Hinton GE, Neal RM, Zemel RS (1995) The Helmholtz machine. Neural Computation 7: 889–904.

28. Ellis K (2022). Dreamcoder official code. GitHub. URL https://github.com/ellisk42/ec.

29. Alford S (2021). bidir-synth: DreamCoder and bidirectional program synthesis on ARC. URL https://github.com/simonalford42/bidir-synth.git.

30. He K, Zhang X, Ren S, Sun J (2015) Spatial pyramid pooling in deep convolutional networks for visual recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence 37: 1904–1916.

31. Lin M, Chen Q, Yan S (2014) Network in network. In: Bengio Y, LeCun Y, editors, 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014.

32. Long J, Shelhamer E, Darrell T (2015) Fully convolutional networks for semantic segmentation. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015. IEEE Computer Society, pp. 3431–3440.

33. Yu F, Koltun V (2016) Multi-scale context aggregation by dilated convolutions. In: Bengio Y, LeCun Y, editors, 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016.

34. Kingma DP, Ba J (2015) Adam: A method for stochastic optimization. In: Bengio Y, LeCun Y, editors, 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015. URL http://arxiv.org/abs/1412.6980.

35. Ouyang L, Wu J, Jiang X, Almeida D, Wainwright CL, et al. (2022). Training language models to follow instructions with human feedback. ArXiv: 2203.02155.

36. Webb T, Holyoak KJ, Lu H (2022). Emergent Analogical Reasoning in Large Language Models. ArXiv:2212.09196.

37. Mitchell M (2020). Can GPT-3 make analogies? URL https://medium.com/@melaniemitchell.me/can-gpt-3-make-analogies-16436605c446.

38. Mitchell M (1993) Analogy-Making as Perception. MIT Press.

39. Marcus G, Davis E (2020) GPT-3, Bloviator: OpenAI's language generator has no idea what it's talking about. MIT Technology Review .

40. Frantar E, Ashkboos S, Hoefler T, Alistarh D (2023). GPTQ: Accurate post-training quantization for generative pre-trained transformers. ArXiv: 2210.17323.

41. OpenAI (2023). TikToken. URL https://github.com/openai/tiktoken.

42. OpenAI (2023). Introducing ChatGPT. URL https://openai.com/blog/chatgpt.

43. Hoffmann J, Borgeaud S, Mensch A, Buchatskaya E, Cai T, et al. (2022) Training compute-optimal large language models. ArXiv: 2203.15556.

44. Bai Y, Geng X, Mangalam K, Bar A, Yuille A, et al. (2023). Sequential modeling enables scalable learning for large vision models. ArXiv:2312.00785.

45. Hu EJ, Shen Y, Wallis P, Allen-Zhu Z, Li Y, et al. (2022) LoRA: Low-rank adaptation of large language models. In: The Tenth International Conference on Learning Representations, ICLR.

46. Qiu L, Jiang L, Lu X, Sclar M, Pyatkin V, et al. (2023). Phenomenal yet puzzling: Testing inductive reasoning capabilities of language models with hypothesis refinement. ArXiv:2310.08559.

47. Zhu Z, Xue Y, Chen X, Zhou D, Tang J, et al. (2023). Large language models can learn rules. ArXiv:2310.07064.

48. Tan J, Min C, Motani M (2023). Large language model (LLM) as a system of multiple expert agents: An approach to solve the abstraction and reasoning corpus (ARC) challenge. ArXiv:2310.05146.

49. Xu Y, Khalil EB, Sanner S (2023) Graphs, constraints, and search for the abstraction and reasoning corpus. Proceedings of the AAAI Conference on Artificial Intelligence 37: 4115-4122.

50. Acquaviva S, Pu Y, Kryven M, Sechopoulos T, Wong C, et al. (2021). Communicating natural programs to humans and machines. ArXiv:2106.07824.

51. Wang R, Zelikman E, Poesia G, Pu Y, Haber N, et al. (2023). Hypothesis search: Inductive reasoning with language models. ArXiv:2309.05660.

52. Kim S, Phunyaphibarn P, Ahn D, Kim S (2022) Playgrounds for abstraction and reasoning. In: NeurIPS Workshop on Neuro Causal and Symbolic AI.

53. Johnson A, Vong WK, Lake BM, Gureckis TM (2021) Fast and flexible: Human program induction in abstract reasoning tasks. Proceedings of the 43rd Annual Meeting of the Cognitive Science Society: Comparative Cognition: Animal Minds, CogSci 2021 : 2471-2477.

54. Park J, Im J, Hwang S, Lim M, Ualibekova S, et al. (2023). Unraveling the arc puzzle: Mimicking human solutions with object-centric decision transformer. ArXiv:2306.08204.

55. Liu E, Neubig G, Andreas J (2024). An incomplete loop: Deductive, inductive, and abductive learning in large language models. ArXiv:2404.03028.

56. Bober-Irizar M, Husain S, Ong EJ, Bober M (2017) Cultivating DNN diversity for large scale video labelling. In: Conference on Computer Vision and Pattern Recognition, CVPR 2017, Youtube-8M Workshop.

57. Vijaymeena M, Kavitha K (2016) A survey on similarity measures in text mining. Machine Learning and Applications: An International Journal 3: 19–28.

58. Wei J, Wang X, Schuurmans D, Bosma M, Ichter B, et al. (2022) Chain-of-thought prompting elicits reasoning in large language models. In: Thirty-sixth Conference on Neural Information Processing Systems (NeurIPS).

59. Yao S, Yu D, Zhao J, Shafran I, Griffiths TL, et al. (2023). Tree of thoughts: Deliberate problem solving with large language models. ArXiv:2305.10601.

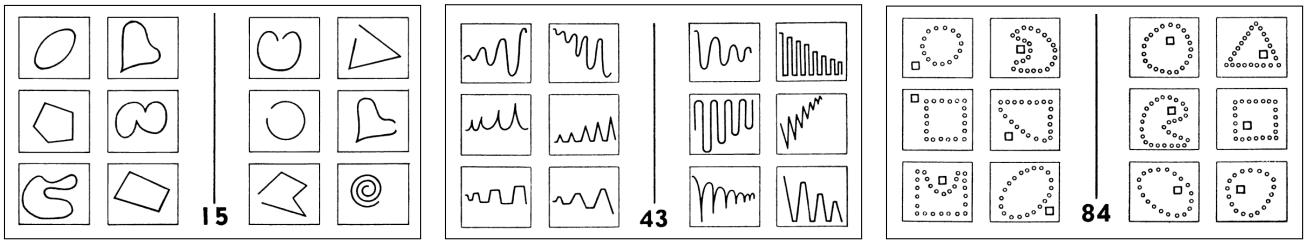
60. Lab42 (2023). ARCreate: Crowdsourcing ARC 2. URL `https://arc-editor.lab42.global/`.

61. Mitchell M, Palmarini AB, Moskvichev A (2023). Comparing Humans, GPT-4, and GPT-4V on abstraction and reasoning tasks. ArXiv:2311.09247v2.

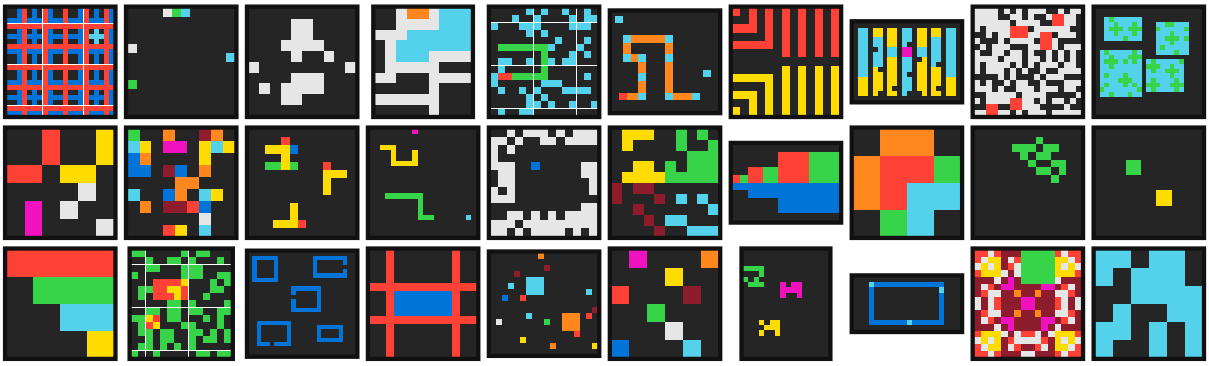
62. Chen X, Liu C, Song D (2019) Execution-guided neural program synthesis. In: International Conference on Learning Representations. URL `https://openreview.net/forum?id=H1gfOiAqYm`.

63. Peterson JC, Bourgin DD, Agrawal M, Reichman D, Griffiths TL (2021) Using large-scale experiments and machine learning to discover theories of human decision-making. Science 372: 1209-1214.

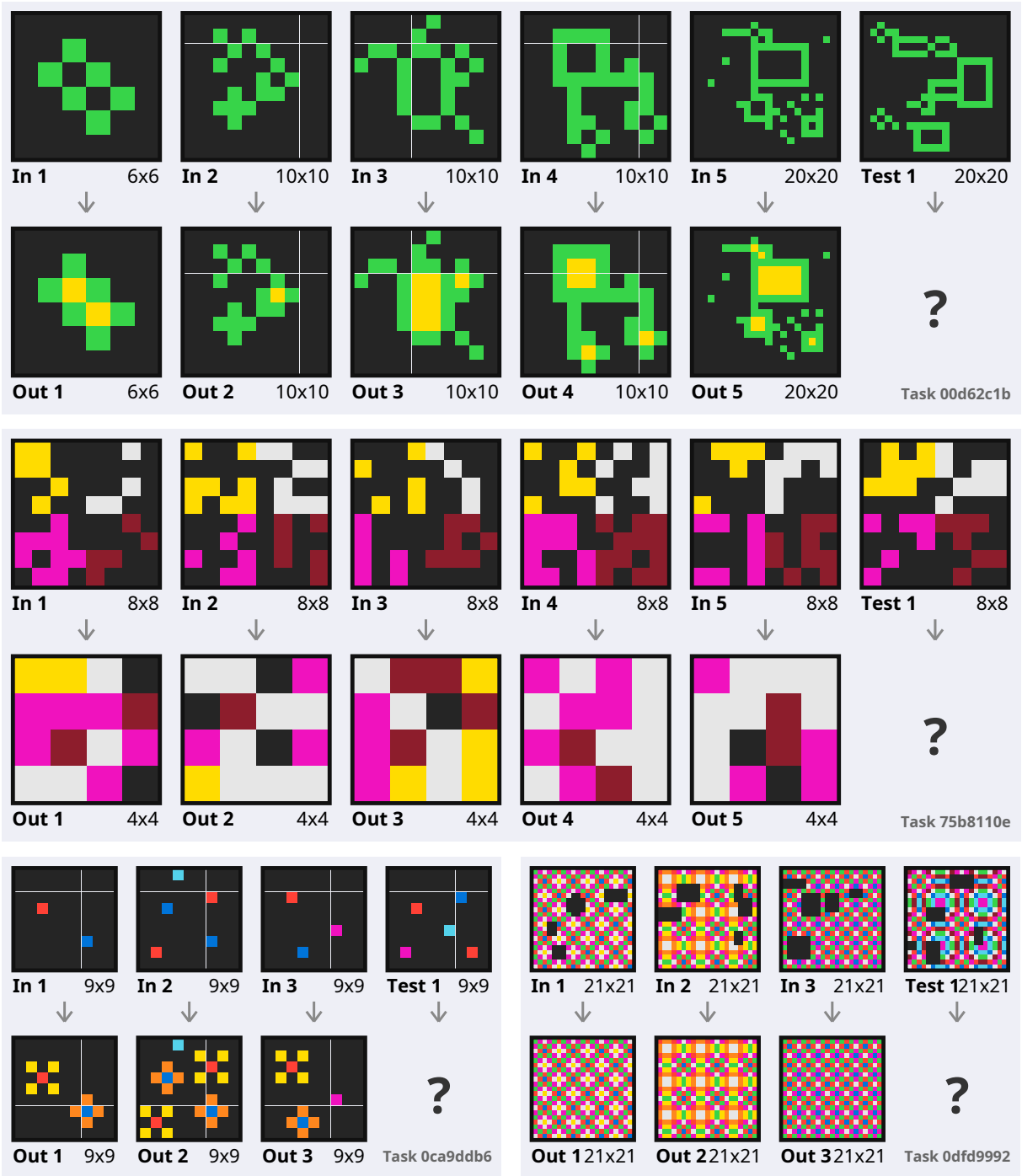
Supplementary Material: Neural networks for abstraction and reasoning: Towards broad generalization in machines



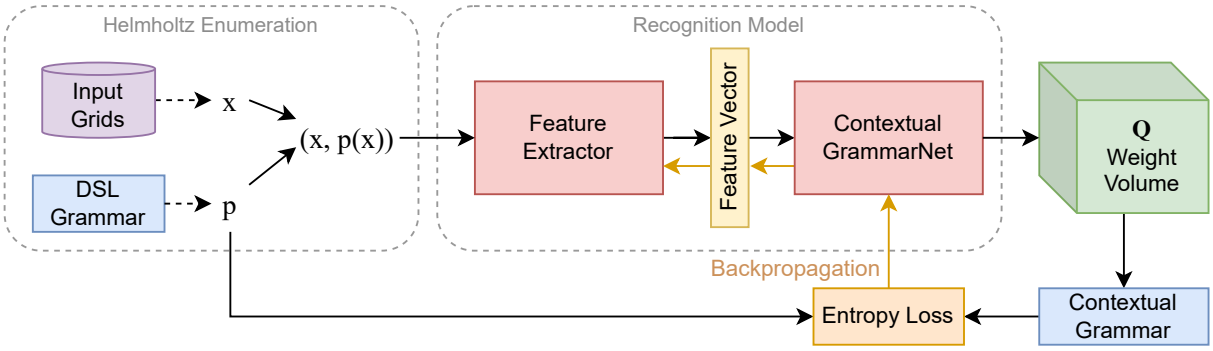
Supplementary Figure 1. Three of Bongard's original problems from 1967. The task is to identify the difference between the two sets, where each problem encapsulates a different concept. **(15)** Set A contains closed shapes while Set B contains open ones. **(43)** Set A contains waves of increasing amplitude; set B decreasing amplitude. **(84)** Set A has a square inside the perimeter defined by connecting the dots; set B outside. Reproduced from [1].



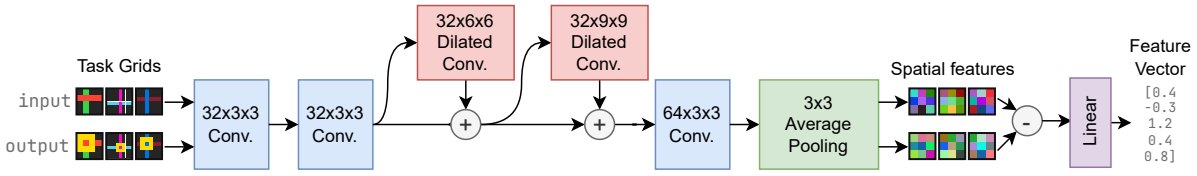
Supplementary Figure 2. The ARC dataset contains 900 handcrafted abstraction and reasoning tasks based on colourful grids. Each task tests new abilities which must be inferred from few examples.



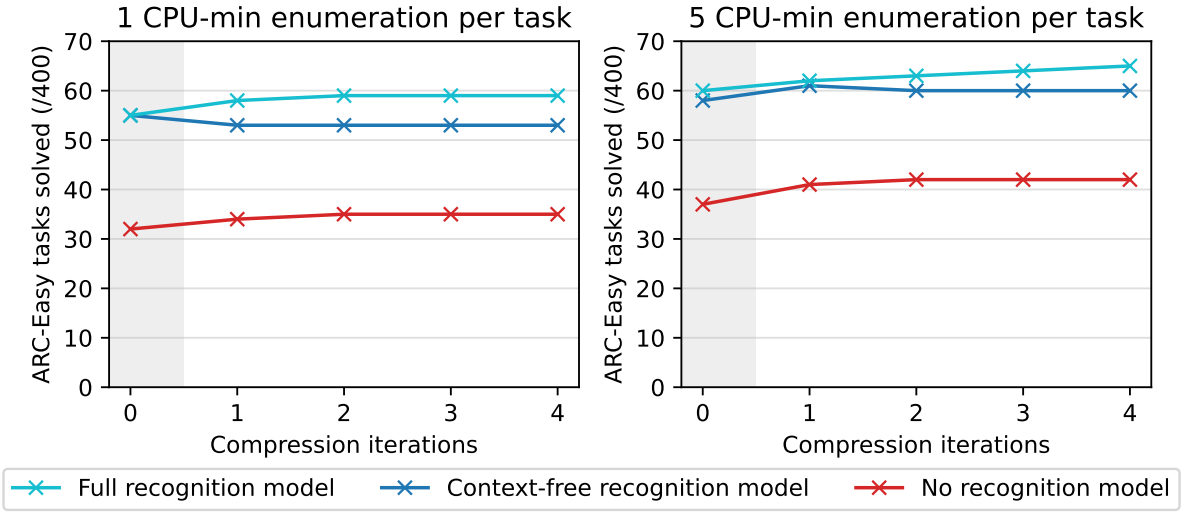
Supplementary Figure 3. Four examples of tasks from the ARC-Easy dataset. Each task requires recognising new patterns and concepts from the training pairs (In/Out) to predict the output of the test examples from the inputs. The tasks have been hand-designed to test a different set of concepts.



Supplementary Figure 4. The training process for the recognition model. Helmholtz enumeration generates an unlimited stream of dreamed tasks, and the recognition model attempts to produce a contextual grammar which assigns a low entropy to the correct solution for the dreamt tasks. The model outputs a weight volume Q, where $Q_{ijk}(x)$ is the probability of primitive i being the kth argument to primitive j for a given task. Backpropagation optimises the neural network end-to-end.



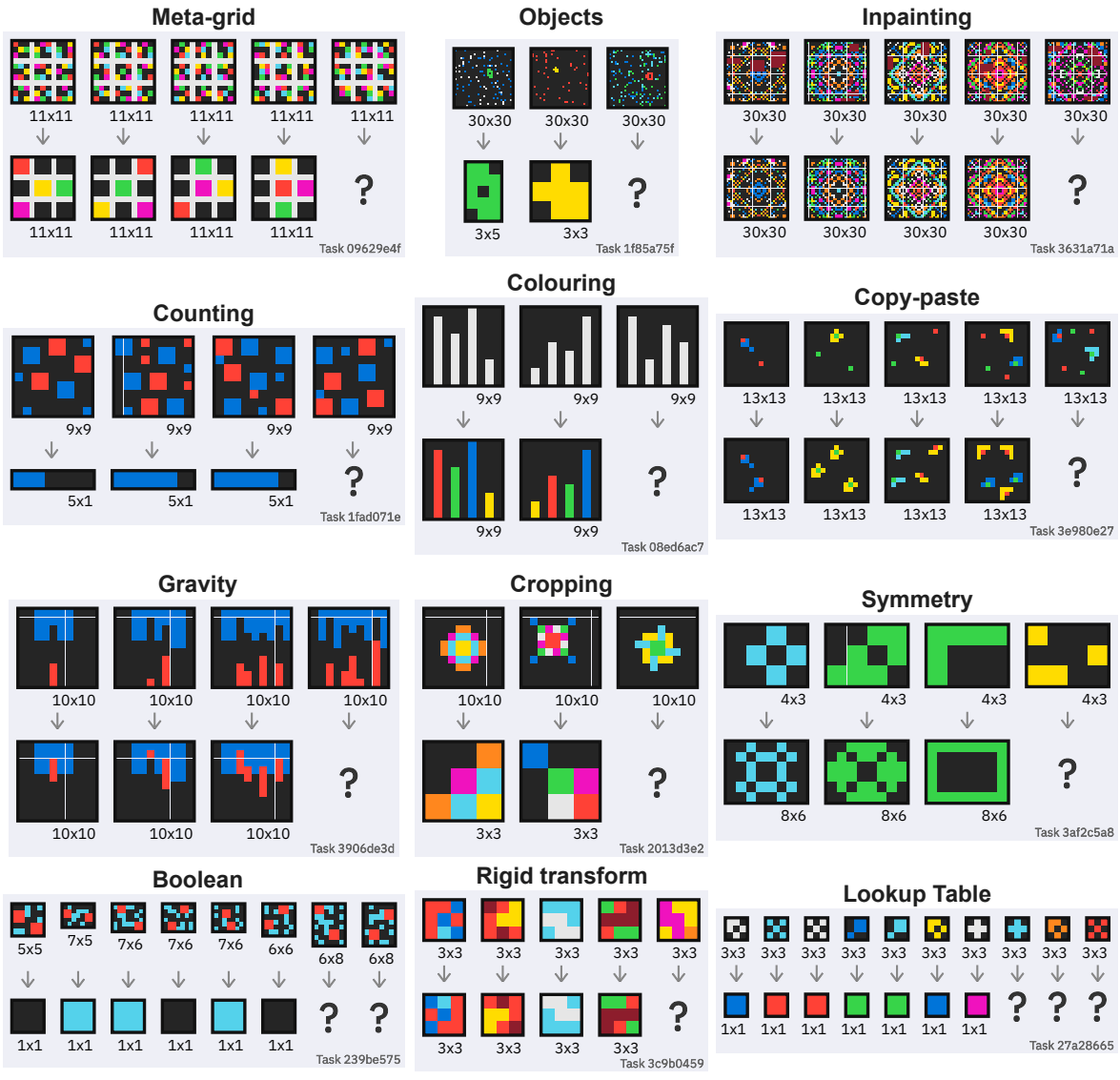
Supplementary Figure 5. The architecture of the fully-convolutional feature extractor. Each grid within an ARC task is operated on by a series of fully-convolutional layers to produce spacial features; the residual between input and output features is turned into a vector that describes the task. The grid size is maintained as long as possible, and the differential of input and output grids are used as features. Residual dilated convolutions ensure multi-scale context can be used. This architecture allows 1x1 up to 30x30 grids to map to the same vector space, as ARC tasks do not have a fixed grid size.



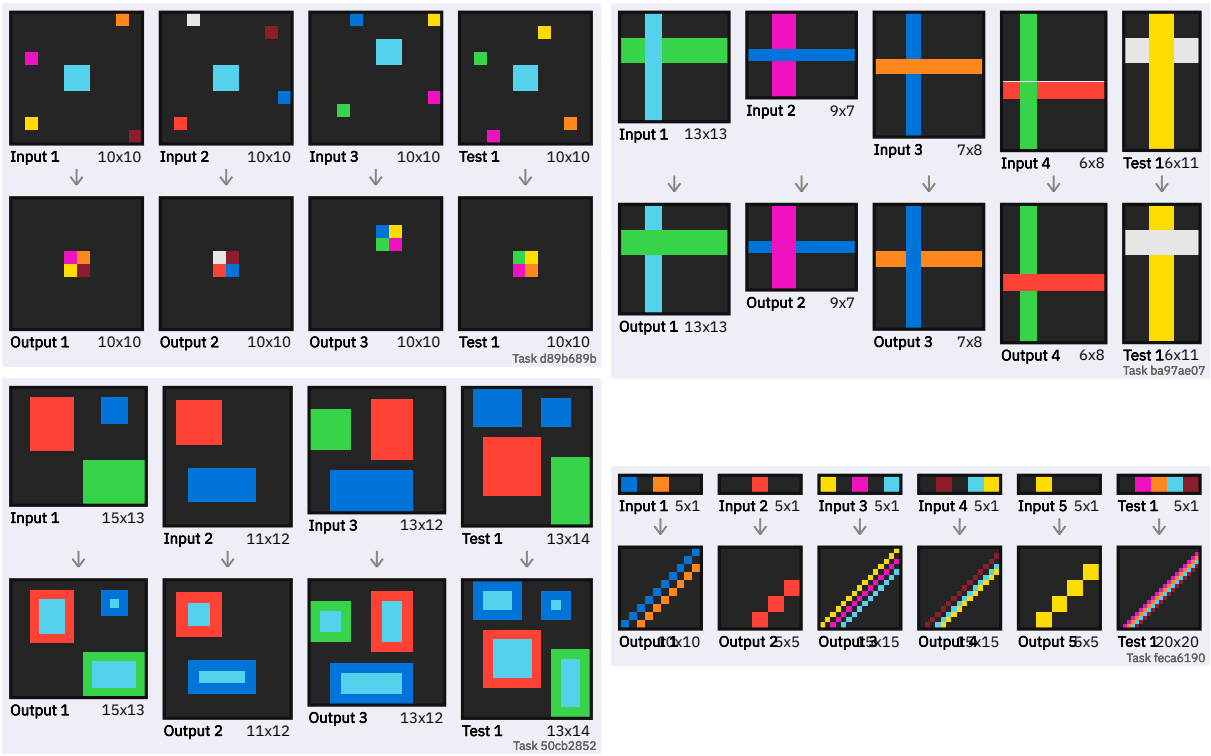
Supplementary Figure 6. The number of ARC-Easy tasks solved by our DreamCoder implementation while ablating two key components. The addition of a neural network recognition model (abstraction sleep) almost doubles the number of solved tasks, while using compression to create new primitives by composing existing ones (dreaming sleep) has a much smaller performance uplift. We also show results with 5x higher enumeration time, showing the effect of a deeper search.

A taxonomy of tasks in ARC

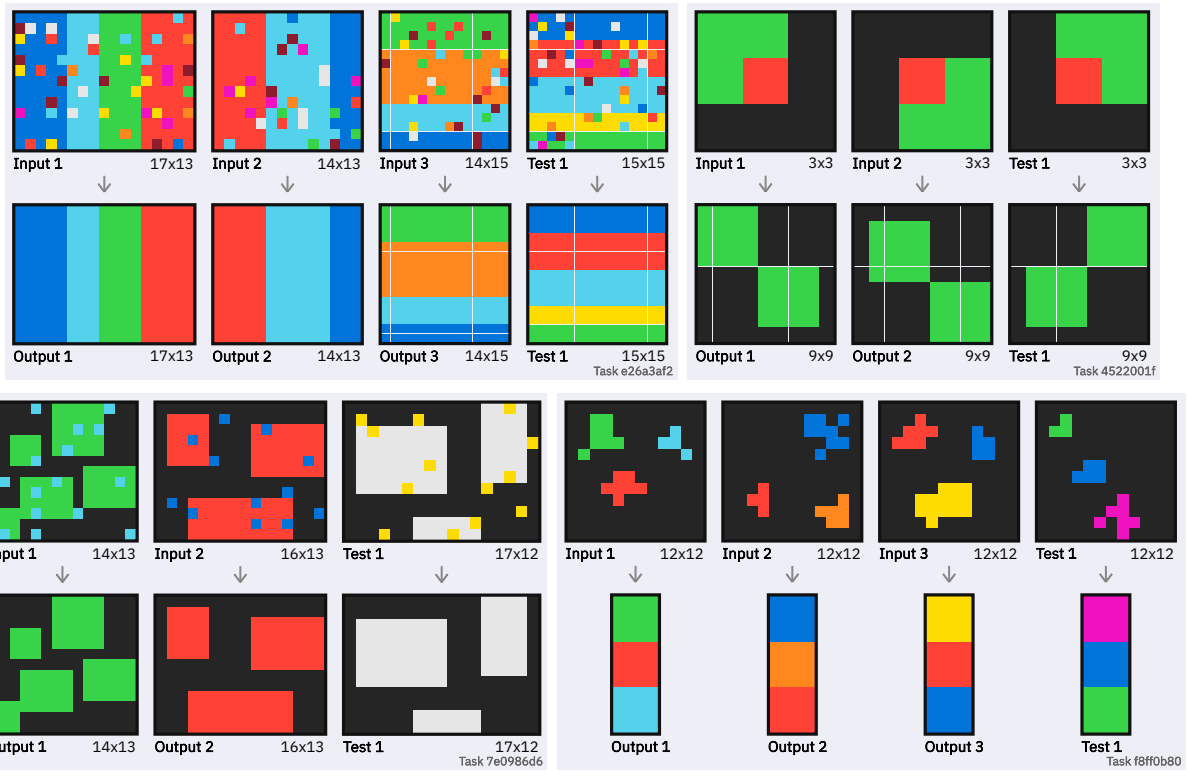
The first step to constructing an ARC solver is to understand the nature of the problem: this helps motivate later design choices. To do this in a systematic way, we built a *taxonomy* of ARC tasks. Some example tasks in this taxonomy are shown in Supplementary Figure 8. Understanding common concepts is vital to designing a good domain-specific language (DSL) to solve tasks that involve these concepts (see Section 3.1).



Supplementary Figure 8. Example tasks for 12 taxonomy tags: the diversity of ARC is apparent. Many tasks combine multiple of these concepts or other concepts not shown.

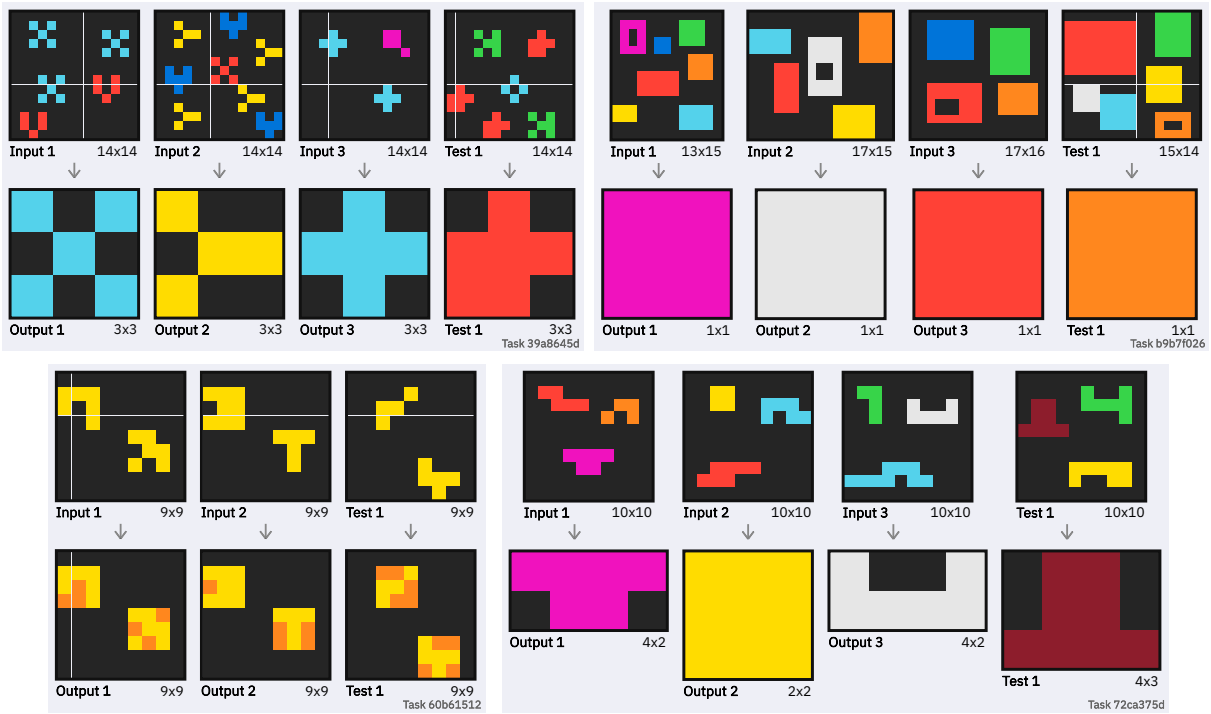


(a) Tasks only solved by Icecuber



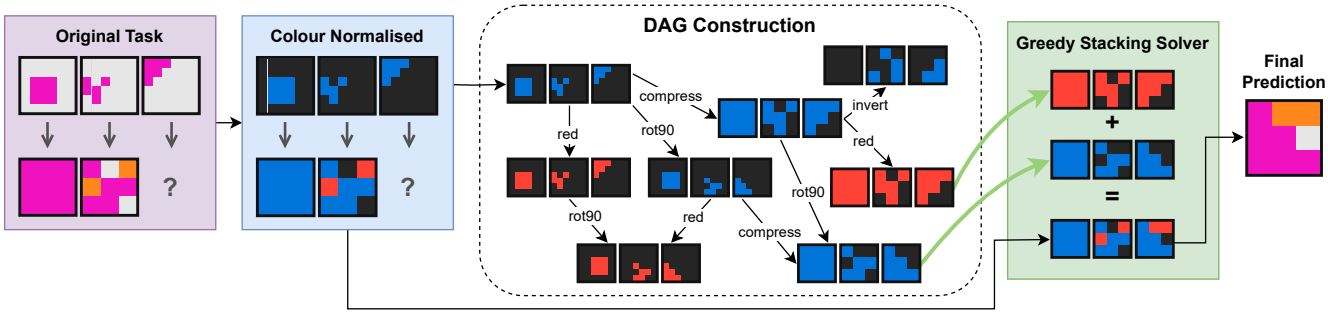
(b) Tasks only solved by GPT-4

Supplementary Figure 7. Randomly selected examples of tasks that only Icecuber, GPT-4 and DreamCoder could solve (respectively). Continued on next page.



(c) Tasks only solved by DreamCoder. Clockwise from top-left, the programs that DreamCoder wrote to solve the tasks were: `(pickcommon (split8 $0))`, `(ic_compress2 (ic_compress3 (ic_connectX $0)))`, `(pickcommon (ic_splitall (mirrorX $0)))`, `(overlay $0 (mapSplit8 (lambda (set_bg c7 $0)) $0))`

Supplementary Figure 7. Randomly selected examples of tasks that only Icecuber, GPT-4 and DreamCoder could solve (respectively).



Supplementary Figure 9. Logical diagram of the Icecuber algorithm. Input grids for a task are combined to form 'pieces'. A brute-force search over unary functions on pieces forms a directed acyclic graph (DAG) of potential outputs, until outputs are found that match the task's outputs. If no exact match is found, a greedy stacker combines DAG nodes to produce the final result, minimising pixel distance to the training outputs.

Icecuber: DSL Search

The existing state-of-the-art approach for ARC was introduced by Johan Sokrates Wind (a.k.a **Icecuber**), achieving a 20.6% private test set accuracy [6] in the first Kaggle competition, and has yet to be surpassed.

Icecuber implements a Domain-Specific Language (DSL) in C++, with 42 image transformation functions with 142 total variants [11] (for example, the `eraseColor` function has variants for 10 colours). Each function is a unary transformation from one grid to another, such as cropping, filling in the interior of objects, or re-colouring a grid.

The DSL is combined with a highly efficient brute-force search written in C++, stacking up to 4 unary functions for each task (Supplementary Figure 9). Rather than storing the programs that generate each specific outcome, each function is applied to the entire set of training and test inputs at once (the starting piece), creating a new piece stored in a directed acyclic graph (DAG). After all programs are enumerated, this DAG contains as many as 10^7 pieces (candidate output grids for each input grid). The DAG allows for de-duplication of outputs, which helps with memory usage and performance (*e.g.* `rotate180(rotate180(grid))` points back to the input piece in the DAG, and therefore is not enumerated any further).

Each piece can then be checked against the training data: if all the training inputs are transformed correctly, we have found a candidate solution to our task (and the corresponding test output is already in the piece).

When none of the candidates exactly match the training outputs, the **greedy stacker** can instead be employed. The greedy stacker works by choosing a random subset of training examples, and **selects pieces from the DAG that most closely match the unexplained cells in the grid**, composing them with transparency to minimise the Hamming distance. This allows bidirectional search: finding intermediate states that compose to the output. Finally, solutions are ranked by a complexity heuristic

and the top three solutions are submitted.

Several additional tricks are employed, such as normalising colours in tasks, and a module which guesses the output grid size before enumeration begins.

By limiting the DSL to unary functions only, the search space size can be greatly limited, and becomes amenable to brute-force search. A more complicated DSL that can arbitrarily combine grids would suffer from super-exponential growth, dramatically reducing the maximum depth possible. However, the greedy stacking solver is able to overcome the drawbacks of this by combining grids *after enumeration*, reducing the overall computational complexity.

Prompting LLMs to complete ARC tasks

Tokenisation

For each type of LLM, we use a different grid-encoding scheme to match the tokeniser used by that LLM. An example of how how the LLM 'sees' an ARC problem is shown in Supplementary Figure 10.

LLaMA We encode grids with no spaces between digits, and with newlines between rows. The LLaMA tokeniser creates a separate token per numerical digit.

OpenAI Completion These models use the GPT-2 tokeniser, which has unique tokens for each digit preceded by a space, so we add a space before each cell to maintain one token per grid.

OpenAI Chat These models use the CL100K tokeniser, which **has no unique tokens for digits followed/led by static characters**; therefore, we do not have a way to force each cell to be a single token. We use a tokenisation with no spaces and therefore allow multiple digits to be merged into one token.

```
We are playing a game which involves transformaing a 2D input grid of digits into an output grid of digits. Every below pair
 of grids contains the same transformation (e.g. rotation, symmetry, manipulation of objects). Each Input grid is followed
 by an Output grid which applies the same transformation as previous Input/Output pairs. One such example is below.
Input 1:
 0 0 6
 0 4 0
 3 0 0
Output 1:
 0 0 6
```

Supplementary Figure 10. An example of how the LLM 'sees' an ARC task, with tokenisation highlighted (in this case using OpenAI Completion tokenisation). The task is structured so that each grid cell is a single token, with the test output left as a text completion.

Prompting

Completion prompt format We use the following prompt for completion models:

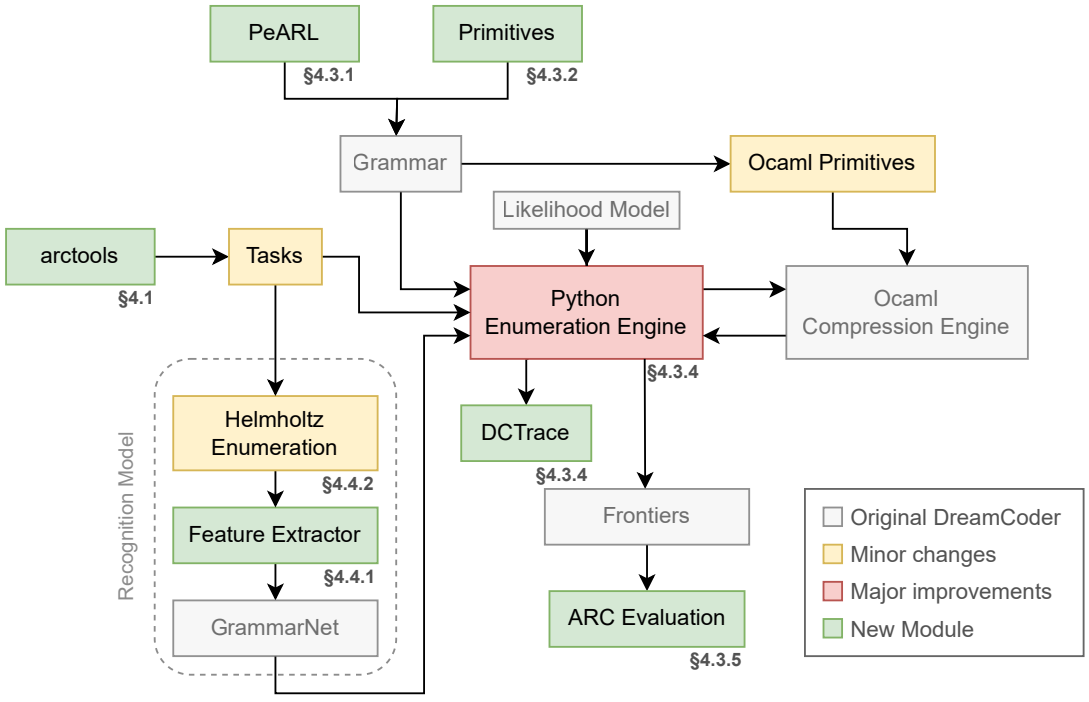
```
We are playing a game which involves transforming a 2D input grid of digits into an output grid of
digits. Every below pair of grids contains the same transformation (e.g. rotation, symmetry, manipulation
of objects). Each Input grid is followed by an Output grid which applies the same transformation as
previous Input/Output pairs. One such example is below.
```

Chat prompt format Chat models are more complex and use a *system* message followed by interleaved user and assistant messages. We use the following system message:

```
We are playing a game which involves transforming an input grid of digits into an output grid of digits.
In general, digits form objects in 2D and the task is to perform some spatial transformation of these
objects to go from the input grid to the output grid. All the information about the transformation is
contained within the input pairs themselves, and your answer will only be correct if the output grid is
exactly correct, so this is what I expect from you. I will begin by giving you several examples of
input-output pairs. You will then be given a new input grid, and you must provide the corresponding
output grid.
```

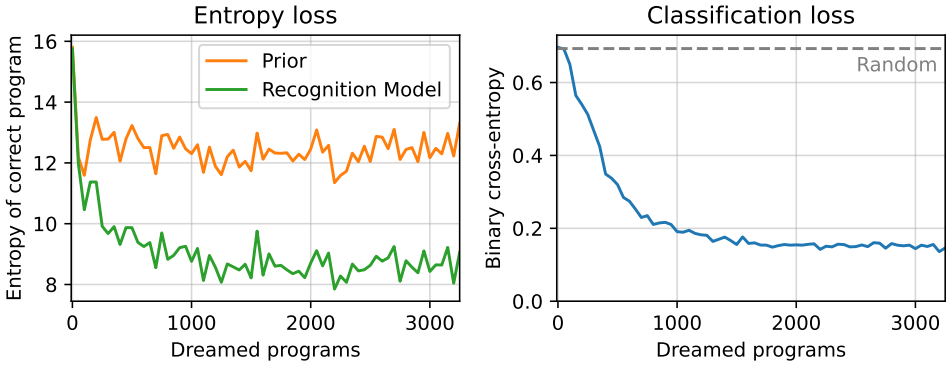
We then interleave the input and output grids as User/Assistant messages, so that the chat model can follow the behaviour of what are presented as its previous responses.

DreamCoder Software Architecture



Supplementary Figure 11. An overview of the DreamCoder software architecture, highlighting the primary modules used, and whether they are new or existing modules.

Abstraction Sleep Results



Supplementary Figure 12. Evaluation metrics for the abstraction sleep phase. Entropy loss gives the average entropy of the solutions generated under the neural-network grammar. In addition, we record the binary cross-entropy loss of classifying which primitives are used in a given task. We see that our recognition model is able to guide the search towards the correct solution; this translates to solving more ARC tasks within a given enumeration time limit. A more detailed explanation of these metrics is given below.

Explanation of Entropy loss and Classification loss In Supplementary Figure 12, we provide two metrics. As an illustrative example, suppose that we have three primitives A, B, C and that the correct program is (λ: `B(A($0))`. First, the recognition model produces probabilities of each primitive being contained in the solution $[0.8, 0.7, 0.2]$. The classification loss is the binary cross-entropy of these predictions with $[1, 1, 0]$ (in this case ≈ 0.39).

On the other hand, entropy is defined as $-\log_2(P(p|R))$, where $P(p|R)$ is the likelihood of correct program p being sampled under this recognition model R. This takes into account both the primitives and their configuration ($B(A(\cdot))$ vs. $A(B(\cdot))$), and is the more meaningful metric for actual performance.

So, a 4 bit reduction in entropy means that the correct solution is $2^4 = 16$ times more likely to be sampled during enumeration, and thus that 16 fewer programs need to be sampled to find the solution (although this is on dreamed tasks, the reduction on ARC tasks is approximately 10x). Higher-entropy programs take exponentially longer to find.

PeARL Primitives

The following list provides a complete description of the primitives available in PeARL, along with the number of times they were used in the 88 tasks solved by DreamCoder. A number of these primitives were adapted from the Icecuber C++ DSL [6]. We use `G` to denote the `grid` type.

Primitive (Count)	**Type**	**Description**
Rigid transformations		
`rot90` (1)	(G → G)	Rotate 90° clockwise
`rot180` (6)	(G → G)	Rotate 180°
`rot270` (1)	(G → G)	Rotate 90° counter-clockwise
`flipx` (3)	(G → G)	Horizontal flip
`flipy` (4)	(G → G)	Vertical flip
`swapxy` (2)	(G → G)	Transpose
Cropping		
`left_half` (6)	(G → G)	Crop the left half of the grid (floor division)
`right_half` (2)	(G → G)	Crop the right half of the grid (floor division)
`top_half` (4)	(G → G)	Crop the top half of the grid (floor division)
`bottom_half` (0)	(G → G)	Crop the bottom half of the grid (floor division)
Uncropping		
`repeatX` (2)	(G → G)	Stack two copies of grid horizontally
`repeatY` (0)	(G → G)	Stack two copies of grid vertically
`mirrorX` (11)	(G → G)	Horizontally mirror grid [abc]->[abccba]
`mirrorY` (11)	(G → G)	Vertically mirror grid
`ic_embed` (4)	(G → G → G)	Embeds a grid into a larger hull defined by a 2nd argument (zero-padded)
Colour manipulation		
`c[1-9]` (25 total)	(colour)	colour values (9 primitives in total)
`topcol` (6)	(G → colour)	The most common non-black colour
`rarestcol` (5)	(G → colour)	The least common non-black colour
`ic_filtercol` (4)	(colour → G → G)	Retains only pixels with the specified colour
`ic_erasecol` (8)	(colour → G → G)	Removes any pixels with the specified colour
`setcol` (7)	(colour → G → G)	Set all non-black pixels to the specified colour
`set_bg` (7)	(colour → G → G)	Set black pixels to the specified colour
`get_bg` (1)	(colour → G → G)	Return grid of background pixels in grid in specified colour
`ic_invert` (2)	(G → G)	Replaces black with the `topcol`, replaces colours with black
`colourHull` (1)	(colour → G → G)	Set every pixel to a colour
Position manipulation		
`getpos` (0)	(G → pos)	Get the position of a cropped grid (default 0,0)
`getsize` (0)	(G → size)	Get the size of the grid
`ic_toorigin` (0)	(G → G)	Reset a grid's position to $(0, 0)$
Morphology		
`fillobj` (5)	(colour → G → G)	Fill each closed object's interior with a specified colour
`ic_fill` (1)	(G → G)	`fillobj` coloured blue
`ic_interior` (0)	(G → G)	Return interior of closed objects only, coloured `topcol`
`ic_center` (1)	(G → G)	Create a grid of $w/2, h/2$ with centred position, coloured blue
`ic_makeborder` (6)	(G → G)	Draw border around objects image in blue (border only)
`ic_spread` (0)	(G → G)	Each black cell is coloured with its closest neighbour colour
`ic_spread_minor` (0)	(G → G)	`ic_spread` ignoring the most common colour.
Counting		
`countPixels` (1)	(G → count)	Return the number of non-black pixels in the input grid
`countcolours` (2)	(G → count)	The number of non-black colours in the input grid
`countComponents` (0)	(G → count)	The number of 4-connected objects in the image
`countToXY` (2)	(count → colour → G)	Draw a new grid of $n \times n$ with the specified colour
`countToX` (0)	(count → colour → G)	Draw a new grid of $n \times 1$ with the specified colour
`countToY` (1)	(count → colour → G)	Draw a new grid of $1 \times n$ with the specified colour
Compression		
`ic_compress2` (14)	(G → G)	Remove rows/columns which are duplicates of preceeding rows/cols
`ic_compress3` (9)	(G → G)	Remove any entirely black rows/columns

Primitive	Type	Description
Drawing		
`ic_connectX` (8)	(G → G)	Join up any objects of the same colour horizontally
`ic_connectXY` (7)	(G → G)	Connect in both X and Y
List creation		
`ic_splitcols` (1)	(G → list(G))	Split a grid based on colours
`ic_splitall` (7)	(G → list(G))	Split grid based on 4-connected objects
`split8` (4)	(G → list(G))	Split grid based on 8-connected objects
`ic_splitcolumns` (1)	(G → list(G))	Create $1 \times n$ grids per column
`ic_splitrows` (0)	(G → list(G))	Create $n \times 1$ grids per row
List reduction		
`pickcommon` (3)	(list(G) → G)	If there are repeated grids, return the most common
`ic_pickunique` (3)	(list(G) → G)	If there is one unique grid, return it
`pickmax_count` (0)	(list(G) → G)	Return grid with the most coloured cells
`pickmax_neg_count` (0)	(list(G) → G)	Return grid with fewest coloured cells
`pickmax_size` (0)	(list(G) → G)	Return grid with largest area
`pickmax_neg_size` (0)	(list(G) → G)	Return grid with smallest area
`pickmax_cols` (3)	(list(G) → G)	Return grid with most colours
`pickmax_interior_count` (2)	(list(G) → G)	Return the grid with the most empty interior holes
`pickmax_neg_interior_count` (0)	(list(G) → G)	Return the grid with the fewest empty interior holes
`pickmax_x_pos` (0)	(list(G) → G)	Return right-most grid
`pickmax_x_neg` (0)	(list(G) → G)	Return left-most grid
`pickmax_y_pos` (0)	(list(G) → G)	Return uppermost grid
`pickmax_y_neg` (0)	(list(G) → G)	Return lowermost grid
List processing		
`mklist` (0)	(G → G → list(G))	Initialise list from two elements
`lcons` (0)	(G → list(G) → list(G))	List cons
Composition		
`ic_composegrowing` (2)	(list(G) → G)	Overlay grids from largest to smallest, taking into account position
`overlay` (19)	(G → G → G)	Overlay two grids transparently. If same size, ignore position
`logical_and` (1)	(G → G → G)	Pixel-wise AND between two grids. Uses colour of first grid
Higher-order functions		
`mapSplit8` (2)	((G → G) → G → G)	Apply a ((G → G)) lambda to all objects individually
Gravity		
`gravity_down` (2)	(G → G)	Move all objects down with gravity and collisions
`gravity_up` (0)	(G → G)	Move all objects up with gravity and collisions
`gravity_left` (0)	(G → G)	Move all objects left with gravity and collisions
`gravity_right` (1)	(G → G)	Move all objects right with gravity and collisions