CODEIT: SELF-IMPROVING LANGUAGE MODELS WITH PRIORITIZED HINDSIGHT REPLAY

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models are increasingly solving tasks that are commonly believed to require human-level reasoning ability. However, these models still perform very poorly on benchmarks of general intelligence such as the Abstraction and Reasoning Corpus (ARC). In this paper, we approach ARC as a programming-by-examples problem, and introduce a novel and scalable method for language model self-improvement called Code Iteration (CodeIt). Our method iterates between 1) program sampling and hindsight relabeling, and 2) learning from prioritized experience replay. By relabeling the goal of an episode (i.e., the target program output given input) to the realized output produced by the sampled program, our method effectively deals with the extreme sparsity of rewards in program synthesis. Applying CodeIt to the ARC dataset, we demonstrate that prioritized hindsight replay, along with pre-training and data-augmentation, leads to successful inter-task generalization. CodeIt is the first neuro-symbolic approach that scales to the full ARC evaluation dataset. Our method solves 15% of ARC evaluation tasks, achieving state-of-the-art performance and outperforming existing neural and symbolic baselines.

1 INTRODUCTION

The Abstraction and Reasoning Corpus (ARC) is a general artificial intelligence benchmark targeted at both humans and AI systems (Chollet, 2019). ARC is a challenging benchmark because it contains few-shot example tasks that assume access to the four innate core knowledge systems: objects, actions, number, and space (Spelke & Kinzler, 2007). It was designed to require no knowledge outside of these priors, and so the massive memorization capability of pre-trained language models is of limited use for this problem. Humans are able to solve 80% of (a random subset of) ARC tasks in user studies (Johnson et al., 2021), whereas state-of-the-art neural approaches based on GPT-4 solve only 12% of evaluation tasks (Gendron et al., 2023).

Each ARC task consists of a number of *demonstration examples*, each consisting of an input and output grid, and one or more test inputs for which the corresponding output must be predicted (see Figure 1). Effective agents use abstractions related to the four core knowledge systems, generalize from demonstration to test examples, and generalize between tasks. For example, an agent may infer that adjacent cells (space) of the same color value (number) form an object. An agent may also infer that multiple objects sometimes attract or repel (action). Using these abstractions to reason about the value of the test output, an agent may generalize from the demonstration examples to the test example.

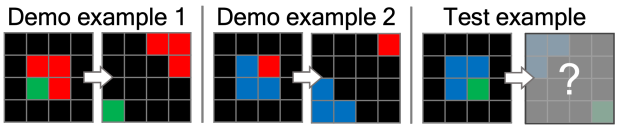


Figure 1: A simplified ARC task. Given two demonstration input-output pairs, the goal is to determine the output grid for the test example, in three attempts or fewer. The size of the grids and the number of demonstration and test examples differs across tasks.

Existing approaches to ARC can be classified as either neural (Gendron et al., 2023; Mirchandani et al., 2023), meaning they directly predict output grids using a neural network, or (neuro-) symbolic (Ainooson et al., 2023; Ferré, 2021; 2023), meaning they first predict a program or other symbolic representation of the mapping between input and output grids, before using it to generate the output grids. Through the use of a well-designed *domain-specific language* (DSL), the symbolic methods can be endowed with prior knowledge analogous to the core knowledge systems found in humans. By combining neural networks and symbolic representations like programs, the system can leverage both prior knowledge and data to solve the ARC tasks.

However, the most effective existing methods, whether neural or symbolic, fail to use experience to generalize between tasks. We propose using Expert Iteration (ExIt) (Anthony et al., 2017) to incorporate experience. ExIt methods do this by alternating between two phases: gathering data with an (often expensive) exploration policy, and improving the policy by training on the newfound experiences. Instead of performing ExIt in the grid space, we take a neuro-symbolic approach and train our model to learn to write programs. This brings us closer to the system that emulates general fluid intelligence described by Chollet (2019): by incorporating new experiences in the form of abstractions.

Recent ExIt approaches employ self-improving language models (Gulcehre et al., 2023; Aksitov et al., 2023; Wang et al., 2023c) to replace the expensive expert by sampling from the policy model and reward-based filtering, saving only trajectories that obtain high reward. This allows them to scale well and benefit from knowledge already captured in the policy. These methods prove effective on program synthesis tasks with natural language specifications (Singh et al., 2023) and code specifications (Haluptzok et al., 2022). However, when solving ARC, agents start ExIt with poor prior knowledge about the search space, as the task is out-of-distribution. Finding a correct program is challenging: positive rewards are extremely sparse. As a result, these methods are sample inefficient in the context of ARC, and programming-by-examples more generally. To enable learning in sparse-reward settings, hindsight relabeling (Andrychowicz et al., 2017) creates artificial expert trajectories post-hoc, and methods that combine ExIt and this technique have improved sample efficiency (Gauthier, 2022; Butt et al., 2022). However, since the relabelled data distribution is constantly changing, there is risk of catastrophic forgetting (French, 1999).

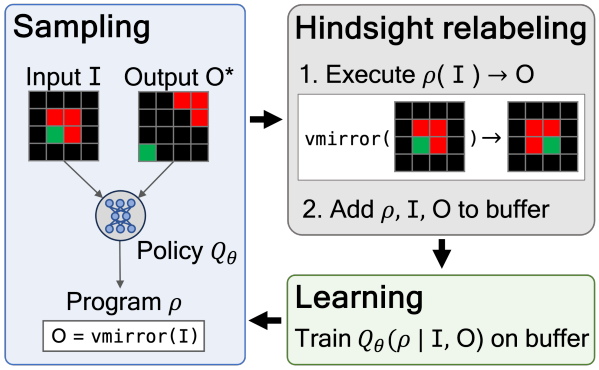


Figure 2: An overview of Code Iteration. In the sampling stage, programs ρ are sampled from the policy Q_θ conditioned on input-output pairs. The program may not produce target output O^* given I, so we use hindsight relabeling: we execute the program, and add the program ρ, inputs I, and *realized* outputs O to the buffer. In the learning stage, we train the policy on samples from the buffer.

In this work, we introduce a novel, scalable expert iteration method for sparse reward settings that does not suffer from catastrophic forgetting. Our method, which we call Code Iteration or *CodeIt* for short, iterates between 1) a sampling and hindsight relabeling stage and 2) a learning stage with prioritized experience replay. We show a visualization in Figure 2. This iterative procedure thus allows us to automatically generate new data without human intervention. Unlike current self-improvement approaches that perform sampling and filtering (Singh et al., 2023), CodeIt learns from all program samples, improving sample efficiency. By prioritizing training on experiences that solve real tasks, we ameliorate the risk of catastrophic forgetting.

CodeIt solves 59 of 400 ARC evaluation tasks, achieving state-of-the-art performance by learning from experiences in the form of abstractions and generalizing to new tasks. We analyze the programs discovered by CodeIt and find that these are on average shorter and use different primitives compared to our custom symbolic baselines. Furthermore, after finding an initial solution, CodeIt continues to improve it over time; shorter solutions are found in 53% of solved ARC tasks, highlighting the ability to perform program refinement. We perform careful ablations to better understand the impact on task performance of key components: ExIt, prioritized hindsight replay, and prior knowledge.

2 METHOD

We approach ARC as a programming-by-examples problem: for a given set of tasks that we call the *search set*, we aim to find programs that correctly match inputs with their respective outputs, and we do so by training a *policy* to produce programs when shown demonstration examples. This is achieved by iterating between two stages: 1) writing programs using a policy and applying hindsight relabeling, and 2) learning from the programs and their input-output examples. We first describe key design choices below, and then explain the iterative procedure.

2.1 DESIGN CHOICES

Programming language We restrict our programming language to the open source domain specific language (DSL) of Hodel (2023). Although various open source DSLs for ARC exist, Hodel designed their DSL using only the ARC training split, whereas some authors incorporate priors from the ARC evaluation split into their DSLs (Icecuber, 2020).

Hodel's DSL contains grid manipulation functions (e.g., `vmirror` or `hmirror`, which mirror the grid along the vertical or horizontal axis), `fill` functions that replace all pixels of a certain color, and functions that return locations of specific pixel groups. See Appendix B.4 for details on the DSL and more example primitives, and see Hodel (2023) for discussion on the DSL's primitives and capability.

Policy Our choice of policy is a pretrained encoder-decoder Large Language Model (LLM). We use the 220 million parameter CodeT5+ (Wang et al., 2023b) model and its default tokenizer, which are pretrained on a diverse set of programming tasks. We input the demonstration examples to the encoder, and let the decoder generate the corresponding program. If necessary, demonstration examples are truncated to fit in the encoder context window.

Grid representation In order to condition the language model policy on input-output grids, we represent them as text. Instead of encoding the grid as a 2-dimensional array, we use an object-centric text representation. Specifically, each color is encoded as an integer, and for each color in the grid we list all the grid cells with that color as $[x, y]$ coordinates. Since the majority of cells belong to the background color, this procedure significantly reduces the number of tokens required to encode the grid (see Figure 11 in Appendix A.3). An example of the sparse grid representation is shown Figure 7 in Appendix A.3 for a toy ARC task.

2.2 THE CODE ITERATION ALGORITHM

We initialize the policy network by training on ground truth data. We then start CodeIt, iterating between *sampling and hindsight relabeling* and *learning*. We refer to one full pass of sampling and learning as a *meta-iteration*. We show the procedure in Fig. 2, and explain each stage in more detail below. For pseudocode, see Appendix A.1.

Initialization We start from a dataset of ARC training tasks and solution programs written in the domain-specific language (DSL) of Hodel (2023), which we call the *training set*. This dataset is expanded by randomly mutating programs (for details of this procedure, see Appendix A.2), resulting in an *augmented training set*.

The initial dataset augmentation step serves multiple purposes. Mixing in mutated programs acts as a form of data augmentation, and is a common approach in policy improvement for program synthesis

(Ellis et al., 2020; Fawzi et al., 2022). Before experiences are sampled from the policy, the model can already learn the DSL syntax, which can be challenging if the training set is small. It also enables the model to learn how to interpret the task demonstration examples before we begin iterative learning, improving the quality of our policy samples in early meta-iterations.

Sampling and hindsight relabeling In the sampling stage, we obtain new programs using the policy Q_θ. Let the *search set* be the set of tasks for which we want to find a corresponding program. For each task in the search set, we convert the demonstration examples' input I and target output O^* from grid to text representation, encode these using the policy, and then autoregressively decode a program: $\rho \sim Q_\theta(\rho|I, O^*)$. We then run the obtained program on the input grids. If the program is syntactically incorrect or the runtime is too high, we discard it. Otherwise, we obtain program outputs $O = \rho(I)$, and can add a new triplet to a replay buffer: the program ρ, the demonstration inputs I, and the realized outputs O (which may or may not match the target outputs O^*). In each sampling stage we repeat this procedure n_ρ times per task, where n_ρ is a hyperparameter.

Replacing the target output by the realized one is a form of hindsight experience replay (Andrychowicz et al., 2017), and ensures that we obtain an experience every time we find a syntactically correct program, thereby preventing stagnation of the buffer. Note that although these programs may not solve the tasks we are interested in, they are always valid in terms of syntax and semantics (correctly mapping $\rho(I) \to O$). They can therefore be used to teach the policy about program syntax and program behaviour, which may lead to positive transfer to the search set. We emphasize that we never add test examples nor performance on the test examples to our buffer, as one should not have access to their target output grid during sampling.

Learning During the learning stage, the policy Q_θ is trained on experiences sampled from the buffer and the augmented training set. These experiences consist of input grids I, output grids O and the corresponding program ρ. The training objective is then a straightforward negative log-likelihood objective:

$$\mathcal{L}(\rho, I, O) = -\log Q_\theta(\rho|I, O). \tag{1}$$

We keep only a single copy of the policy network, and continue to update it during each learning stage. In particular, we do not compare with past versions to guarantee an improvement in the policy before using it in the next sampling stage. Although continual updates could lead to worse performance in the next iteration, we find this is not a problem in practice.

By default, we perform prioritized sampling from the replay buffer (Schaul et al., 2015). For each experience, the priority is proportional to the percentage of demonstration (target) outputs equal to program outputs. Intuitively, this means that programs that solve real ARC tasks' demonstration examples are sampled more often.

3 EXPERIMENTS

In this section, we aim to demonstrate the efficacy of CodeIt, and break down how much different components of the method contribute to the performance. We first tuned hyperparameters on a custom training and validation split (for a description of these parameters and details, see Appendix B). Using these hyperparameters, we benchmark our method on the ARC evaluation split and compare against previous state-of-the-art methods. Finally, we ablate the importance of individual components of CodeIt.

We define *demonstration performance* as the percentage of solved demonstration examples on a given task. We first sort solution programs by demonstration performance, and then by program length, favoring shorter programs. We evaluate the top three programs on the set of test examples. Following ARC evaluation procedure, if at least one of these three programs maps all test example inputs to outputs, the task is solved and *test performance* is 1. We emphasize that the ExIt procedure only makes use of demonstration examples, and that we use test performance for final evaluation only.

Custom baselines We use a random baseline that samples programs line-by-line. At the start of each line, we sample a primitive function from the DSL, then sample arguments given its expected input types. When a variable of type "grid" is created, we end the program with probability 0.8, otherwise we add another line to the program.

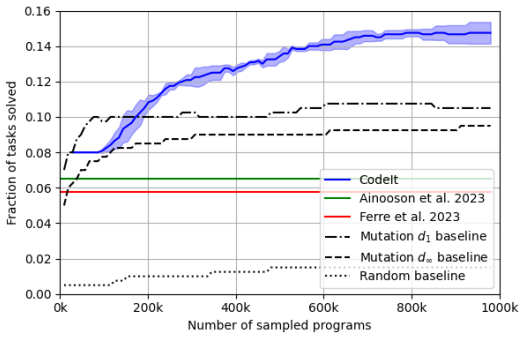


Figure 3: Cumulative performance as function of number of sampled programs for CodeIt and various baselines. We run CodeIt for three random seeds.

Method	policy perf.	cumulative perf.
CodeIt	49/400	59/400
A1: No ExIt	13/400	45/400
A2: No relabeling	24/400	42/400
A3: No priority	38/400	58/400
A4: No pretraining	9/400	35/400
A5: One demo	34/400	51/400
A6: No mutation	17/400	20/400

Table 1: ARC evaluation performance of CodeIt ablations.

We also use a mutation-based baseline. This is a more advanced procedure, designed with the DSL in mind. At every meta-iteration, it mutates the set of training programs provided by Hodel (2023). We use two variations: "d_1" mutates only the initial training set, and "d_∞" can augment newfound programs as well. We provide the exact algorithm in Appendix A.2.

For all three baselines, we sample $n_m = n_\rho \cdot n_{tasks}$ programs per meta-iteration. Here, n_ρ is the desired number of programs per meta-iteration per task, and n_{tasks} the total number of tasks in the population. To strengthen these baselines, we exhaustively evaluate each found program on all inputs in the search set, and check the outputs against ARC output grids.

Baselines from literature We include approaches from literature as baselines as well. A direct comparison is sometimes difficult, as not all baselines apply their method to the full ARC evaluation set: for example, Kolev et al. (2020) and Alford et al. (2021) focus only on a subset of ARC. Additionally, some symbolic methods design a DSL based on both ARC training and evaluation sets and report results on a hidden test set (Icecuber, 2020). We therefore only compare to approaches that report scores on the full ARC evaluation set.

Ainooson et al. (2023) and Ferré (2023) both run a search procedure for a custom DSL on the full set. As Ainooson et al. (2023) report the highest performance the full ARC evaluation set, this is our main symbolic baseline. Although Mirchandani et al. (2023) and Gendron et al. (2023) use a different evaluation protocol, we include these as our main neural baseline, as they are based on powerful LLMs (text-davinci and GPT-4).

3.1 Setup

We initialize our training set with the 400 examples from the ARC training split and the associated solution programs provided by Hodel (2023). We also sample 19,200 programs as additional training data via the mutation procedure outlined in Appendix A.2. We use the programs that are syntactically correct to initialize the augmented training set. We use the 400 ARC evaluation examples as our search set.

In the sampling stage of each meta-iteration, we use temperature sampling with temperature $\tau = 0.95$, and sample up to $n_\rho = 24$ programs per task. This encourages exploration and, as a result, increases the diversity of data added to the replay buffer. We reject policy-sampled programs if they are syntactically incorrect, or if they run for more than 0.25 seconds per program line. All valid programs are added to the replay buffer.

In each learning stage, we start by sampling a set of experiences from the buffer under the distribution given by the priorities. Each meta-iteration, we sample $r_t = 10,000$ experiences from the concatenation of the train set and the augmented train set, and $r_p = 90,000$ experiences from the buffer. The resulting set is used for 1 epoch of training. For a full list of hyperparameters, see Table 3 in the Appendix.

Method	ARC Train Set	ARC Eval Set	ARC Eval 412
Ferré (2021)	29 / 400	6 / 400	-
Ainooson et al. (2023) MLE	70 / 400	17 / 400	-
Ainooson et al. (2023) brute force	104 / 400	26 / 400	-
Ferré (2023)	96 / 400	23 / 400	-
Mirchandani et al. (2023) text-davinci-003	56 / 400*	27 / 400*	-
Gendron et al. (2023) GPT-4	-	-	49 / 412*
Mutation d_1 baseline	-	42 / 400	39 / 412*
Mutation d_∞ baseline	-	38 / 400	36 / 412*
Random baseline	-	6 / 400	7 / 412*
CodeIt	-	**59 / 400**	**59 / 412***

Table 2: Main results on ARC eval set. The evaluation metric is pass@3 by default, * indicates pass@1. To enable comparison to related work of Gendron et al. (2023), we also include pass@1 performance on the ARC Eval set with 412 examples. Our method outperforms all previous baselines. More details on the ARC splits and evaluation procedures can be found in Appendix A.4.

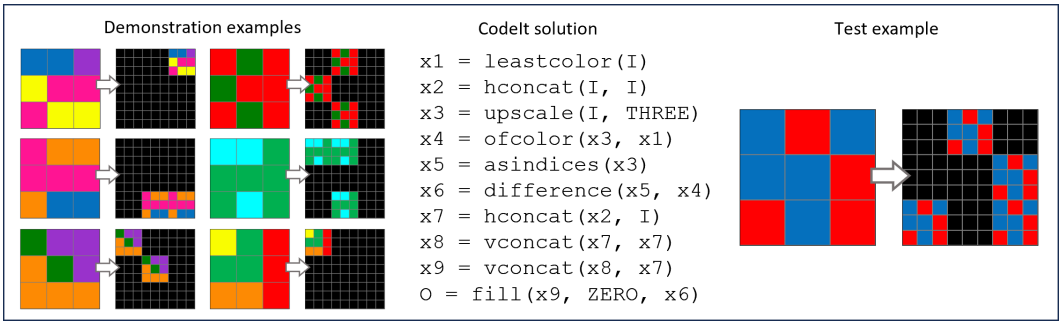


Figure 4: ARC evaluation task 48f8583b and the solution program found by CodeIt.

3.2 Main results on ARC eval set

In Figure 3, we show performance as a function of the number of sampled programs, for CodeIt, our custom baselines, Ainooson et al. (2023) and Ferré (2023). We show *cumulative performance* here, which means that any program in the buffer or augmented train set is considered a solution candidate. For the mutation baselines, we see a rapid performance increase followed by stagnation. In comparison, CodeIt takes several meta-iterations to start generating solutions outside of the augmented train set and then performance rapidly increases. CodeIt quickly outperforms the mutation baseline, indicating that it indeed finds higher-quality samples to train on.

We report final performance of CodeIt after 100 meta-iterations, and the performance of various baselines, in Table 2. To enable comparison to Gendron et al. (2023), we include results on the "ARC Eval 412" set, which treats each test example in the ARC evaluation set as a separate task. Our approach outperforms symbolic approaches (Ainooson et al., 2023; Ferré, 2021; 2023), but also neural approaches based on large language models (Gendron et al., 2023; Mirchandani et al., 2023), achieving state-of-the-art performance on the ARC evaluation set.

For context, we show a solution written by CodeIt for an example task in Figure 4. To further illustrate the differences between the programs found by CodeIt and the mutation baselines, we analyze solutions found by each method in Appendix C.1, including a qualitative comparison in Table 4. One finding is that there are 29 tasks for which CodeIt and the mutation baseline both find a solution, but that there are 23 tasks for which only CodeIt finds a solution, versus 13 for the mutation baseline. For the tasks that both methods solve, CodeIt finds shorter programs on average and uses different primitives. In Appendix C.2, we observe CodeIt refines its initial solution for 53% of solved tasks, producing a shorter solution in a later meta-iteration.

3.3 ABLATIONS

In Figure 5 and 6, we report cumulative performance and policy performance over time for CodeIt and all ablations. In all cases, we initialize the method with the ARC train set, and use the ARC evaluation set as search set. We show the results of ablations at the end of training in Table 1.

A1: No ExIt This ablation removes policy feedback, to isolate the contribution of Expert Iteration. In every meta-iteration, instead of populating the buffer with policy samples, we take the programs generated in that meta-iteration of the mutation d_1 baseline. For each program, we randomly select a task from the search set and perform hindsight relabelling, adding the program, input, output triplet to the buffer. We sample $r_p + r_t = 100,000$ experiences from the concatenation of the train set, the augmented train set and the buffer at each meta-iteration for learning. We see that A1 outperforms the mutation baseline, which means supervised learning from mutation experiences alone does lead to some inter-task generalization. However, cumulative performance is substantially lower than CodeIt. This highlights the importance of policy feedback.

A2: No relabeling We test the effect of hindsight relabeling by only adding experiences to the buffer if the program produces the correct output for all demonstration examples. We train on all experiences in the buffer without prioritized sampling. Although performance increases in early meta-iterations, A2 stagnates after around 30 meta-iterations, indicating that data generated by sampling and filtering alone is not sufficient. Sampling and hindsight relabeling (CodeIt) performs better than sampling and filtering (A2).

A3: No priority To test the hypothesis that prioritized sampling ameliorates catastrophic forgetting, we draw experiences uniformly from the buffer in the learning stage. A3 leads to a small reduction in cumulative performance, but a large reduction in policy performance, indicating that the policy indeed forgets important experiences. Prioritized sampling results in better retention of knowledge.

A4: No pretraining To identify whether our pre-trained policy contains beneficial prior knowledge, we randomly reinitialize the policy's weights at the start of CodeIt. Policy performance shows that performance improvement is much slower. Moreover, inter-task generalization begins later, as shown by the cumulative performance, which only starts increasing after around 50 meta-iterations. Despite the expected slowdown, it is encouraging to see that CodeIt does seem to be able to bootstrap from random weights.

A5: One demo We investigate CodeIt's use of the task representation by decreasing the number of demonstration examples shown to the policy. This results in a significant decrease in both cumulative and policy performance. This indicates CodeIt forms abstractions over multiple demonstration examples.

A6: No mutation In this ablation, we omit the mutation-based training data augmentation step. We observe that taking out mutation-based bootstrapping results in slower training, although performance does increase over time and does not stagnate. We therefore conjecture that mutation-based augmentation is not necessary but still useful.

4 RELATED WORK

4.1 ABSTRACTION AND REASONING CORPUS (ARC)

Various works apply program synthesis approaches to subsets of the ARC dataset. Xu et al. (2022) propose to represent grids as graphs, and applies logical programs to the graph nodes, solving 63 of 160 tasks. Kolev et al. (2020) apply a Differentiable Neural Computer to ARC, solving 78% of tasks with grids of size 10×10 and smaller. Alford et al. (2022) applies DreamCoder (Ellis et al., 2020) and execution-guided synthesis, solving 22 of 36 considered tasks. Park et al. (2023) collects human feedback, then performs behavioral cloning for a subset of ARC tasks using a decision transformer (Chen et al., 2021). However, none of these methods are applied on the full ARC evaluation set, typically due to poor scaling behavior. Unpublished approaches are described in Appendix D.

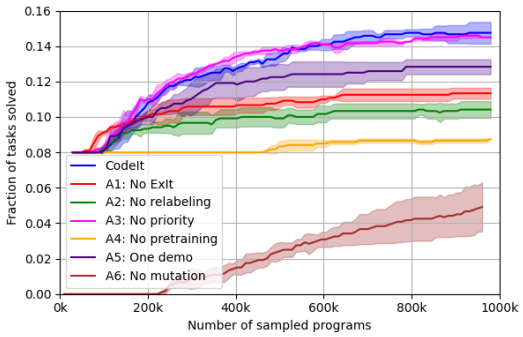


Figure 5: Cumulative performance as function of number of sampled programs for CodeIt and ablations, for three random seeds. For cumulative performance, all programs in the augmented train set and buffer are candidate solutions

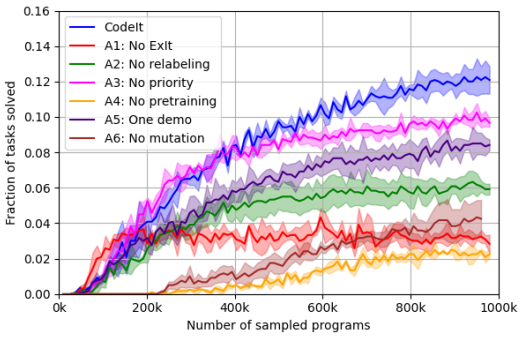


Figure 6: Policy performance as function of number of sampled programs for CodeIt and ablations, for three random seeds. For policy performance, only programs output by the policy in the *current* meta-iteration are candidate solutions.

The few works that do scale to the full evaluation set tend to solve each task in isolation. Ferré (2021) and followup work Ferré (2023) design a custom DSL and perform a fast search for each task. Ainooson et al. (2023) designs a custom DSL as well and obtains best performance with a brute-force search, solving 36 of 400 evaluation tasks. Mirchandani et al. (2023) and Gendron et al. (2023) demonstrate that a pretrained language model with custom tokenizer will output the correct grid after being shown multiple input-output pairs, solving 27 of 400 and 49 of 412 evaluation tasks respectively. Wang et al. (2023a) further augment this approach by generating hypotheses in multiple rounds, although they only show performance on a subset of the ARC training set due to the high monetary cost of querying the language model. In this work, we design a scalable ExIt approach that combines a smaller language model with the higher-level abstraction of a DSL. We also ensure that our approach incorporates experience to benefit from generalization between tasks.

4.2 Expert Iteration

Expert iteration (ExIt) (Anthony et al., 2017) consists of a policy-guided search stage that gathers new experiences, and a learning stage that improves the policy by imitation learning. Commonly used experts tend to be powerful and computationally intensive tree search algorithms such as Monte Carlo Tree Search (Kocsis & Szepesvári, 2006) and greedy search (Daumé et al., 2009). ExIt has achieved superhuman performance include games (Silver et al., 2016; 2018; Anthony et al., 2017) and combinatorial problems such as bin-packing (Laterre et al., 2019). Related work that employs hindsight relabelling in expert iteration are Gauthier & Urban (2022) and Butt et al. (2022).

Applications of ExIt for programming-by-examples (Mankowitz et al., 2023; Ellis et al., 2020) are most relevant to CodeIt. Mankowitz et al. (2023) consider one task only: writing a fast sorting algorithm. For this problem, inter-task generalization is therefore not as important. DreamCoder (Ellis et al., 2020) is most related to our work, since this ExIt method is applied to multiple programming-by-examples tasks. DreamCoder uses a growing DSL to store abstractions, and a computationally intensive search procedure. Instead, CodeIt uses the model to store distilled knowledge, and generates experiences via sampling from the model. Furthermore, DreamCoder filters solutions based on correctness whereas CodeIt uses hindsight relabeling and prioritized experience replay.

4.3 Self Improving Large Language Models

Finetuning pre-trained large language models (LLMs) on synthetic data enables knowledge transfer due to the prior domain knowledge captured in their weights (Butt et al., 2022). Recently, methods that use LLMs to synthesize training data have shown successes in theorem proving (Polu et al., 2022), question answering (Zelikman et al., 2022; Aksitov et al., 2023), mathematical reasoning (Ni et al., 2023), machine translation (Gulcehre et al., 2023), language-to-code generation (Zhou et al., 2023; Singh et al., 2023) and code-to-code generation (Haluptzok et al., 2022).

5 DISCUSSION

Various factors make ARC uniquely challenging for learning-based approaches, for example the limited amount of training data, and the complexity of individual tasks. Another issue is that tasks may differ in number of examples and input dimensionality, which requires agents to reason about concepts at different scales. Here, we provide intuition for why CodeIt works well on this benchmark.

Ablations showed that hindsight relabeling has a large effect on performance. Many expert iteration approaches rely on the emergence of a curriculum of increasingly difficult tasks, even creating a curriculum by comparing the current agent to past versions of itself (Silver et al., 2016; Fawzi et al., 2022) or reward shaping (Laterre et al., 2019; Gulcehre et al., 2023). Hindsight relabeling forms an implicit curriculum (Andrychowicz et al., 2017): initially we collect easy tasks that can be solved in few lines of code, while later on, programs become more complex. This is useful for ARC, where obtaining even one solved task is challenging. As relabeling adds many programs to the buffer, including some that are further away from the target tasks, we used prioritized sampling to avoid catastrophic forgetting.

A potential limitation of CodeIt is that for ARC, it relies on hand-designed components: a DSL, access to an interpreter, and an initial set of ground truth programs. While we do benefit from Hodels expert-designed DSL, we also showed that a neuro-symbolic approach (ablation A1) outperforms a symbolic approach (the mutation baseline), indicating that both DSL and learning contribute to performance. Further, CodeIt outperforms both, indicating that ExIt compounds this effect. We also use a pretrained LLM and mutation procedure to speed up training, but ablations showed that training is possible even without these, albeit at a slower pace. Nevertheless, approaches that can start learning tabula rasa, or form their own DSL (Ellis et al., 2020) remain an important area of research.

For the ARC dataset, it is currently beneficial to incorporate both prior knowledge (via a DSL or pre-trained LLM) and experience (via expert iteration). Chollet (2019) defines the intelligence of a system as "a measure of its skill-acquisition efficiency over a scope of tasks, with respect to priors, experience, and generalization difficulty". Chollet poses that, if two systems are initialized with the same prior knowledge and go through the same amount of experience with respect to a set of unseen tasks, the more intelligent system will combine prior knowledge and its experience more efficiently.

Although many existing approaches incorporate prior knowledge through a programming language or DSL (Ainooson et al., 2023; Ferré, 2023), a pre-trained large language model (Gendron et al., 2023; Mirchandani et al., 2023), or both (Wang et al., 2023a), they cannot incorporate new experience, and therefore do not benefit from inter-task generalization. Alford (2021) proposes an expert iteration method that does learn from experience, but it does not scale well nor benefit from prior knowledge in its policy. We pose that CodeIt is the more effective expert iteration method due to its use of scalable components: pre-trained language models, likelihood-based training, and running programs in interpreters. There is also an implicit relationship between *computational* efficiency and experience: since CodeIt's policy learns on the ARC domain, it is possible to use a much smaller language model than for example Gendron et al. (2023), who use GPT-4 as a policy. This is consistent with LLM literature showing that high quality training data with a curriculum enables smaller LMs to compete with much larger ones on coding tasks (Gunasekar et al., 2023).

6 CONCLUSION

We introduce a novel and scalable method for self-improving language models, *CodeIt*, that uses prioritized hindsight replay. CodeIt achieves state-of-the-art performance on the Abstraction and Reasoning Corpus (ARC) compared to symbolic and neural baselines, solving 59 of 400 evaluation tasks. Ablations show that hindsight relabeling leads to improved sample efficiency resulting in a 40% improvement in performance. We also find that prioritizing important experiences during training ameliorates catastrophic forgetting. Additionally, we observe that CodeIt is able to refine solutions over time, identifying a shorter program for 53% of solved tasks in later iterations. The results demonstrate that our self-improving language model is capable of reasoning in the program space and generalizing between tasks. For the challenging ARC benchmark, both scalability and learning from experience prove to be key components for success.

REFERENCES

James Ainooson, Deepayan Sanyal, Joel P. Michelson, Yuan Yang, and Maithilee Kunda. An approach for solving tasks on the abstract reasoning corpus, 2023.

Renat Aksitov, Sobhan Miryoosefi, Zonglin Li, Daliang Li, Sheila Babayan, Kavya Kopparapu, Zachary Fisher, Ruiqi Guo, Sushant Prakash, Pranesh Srinivasan, Manzil Zaheer, Felix Yu, and Sanjiv Kumar. Rest meets react: Self-improvement for multi-step reasoning llm agent, 2023.

Simon Alford. *A Neurosymbolic Approach to Abstraction and Reasoning*. PhD thesis, Massachusetts Institute of Technology, 2021.

Simon Alford, Anshula Gandhi, Akshay Rangamani, Andrzej Banburski, Tony Wang, Sylee Dandekar, John Chin, Tomaso A. Poggio, and Peter Chin. Neural-guided, bidirectional program search for abstraction and reasoning. *Complex Networks*, 2021.

Simon Alford, Anshula Gandhi, Akshay Rangamani, Andrzej Banburski, Tony Wang, Sylee Dandekar, John Chin, Tomaso Poggio, and Peter Chin. Neural-guided, bidirectional program search for abstraction and reasoning. In *Complex Networks & Their Applications X: Volume 1, Proceedings of the Tenth International Conference on Complex Networks and Their Applications COMPLEX NETWORKS 2021 10*, pp. 657–668. Springer, 2022.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. May 2017.

ARCathon Leaderboard. `https://lab42.global/arcathon/leaderboard/`, 2023. Accessed: 2024-30-01.

Natasha Butt, Auke Wiggers, Taco Cohen, and Max Welling. Program synthesis for integer sequence generation. 2022.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling, 2021. URL `https://arxiv.org/abs/2106.01345`.

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

Hal Daumé, John Langford, and Daniel Marcu. Search-based structured prediction. *Machine learning*, 75:297–325, 2009.

Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Luc Cary, Lucas Morales, Luke B. Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. *CoRR*, abs/2006.08381, 2020. URL `https://arxiv.org/abs/2006.08381`.

Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Barekatain Mohammadamin, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610, 2022. doi: https://doi.org/10.1038/s41586-022-05172-4.

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

Sébastien Ferré. Tackling the abstraction and reasoning corpus (arc) with object-centric models and the mdl principle. *arXiv preprint arXiv:2311.00545*, 2023.

Robert French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3: 128–135, 05 1999. doi: 10.1016/S1364-6613(99)01294-2.

Thibault Gauthier. Program synthesis for the oeis, 2022. URL https://arxiv.org/abs/2202.11908.

Thibault Gauthier and Josef Urban. Learning program synthesis for integer sequences from scratch, 2022. URL https://arxiv.org/abs/2202.11908.

Gaël Gendron, Qiming Bao, Michael Witbrock, and Gillian Dobbie. Large language models are not strong abstract reasoners, 2023.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, et al. Reinforced self-training (rest) for language modeling. *arXiv preprint arXiv:2308.08998*, 2023.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need, 2023.

Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language models can teach themselves to program better. *arXiv preprint arXiv:2207.14502*, 2022.

Michael Hodel. Domain-specific language for the abstraction and reasoning corpus, 2023.

Icecuber. https://www.kaggle.com/competitions/abstraction-and-reasoning-challenge/discussion/154597, 2020. Accessed: 2024-30-01.

Aysja Johnson, Wai Keen Vong, Brenden M. Lake, and Todd M. Gureckis. Fast and flexible: Human program induction in abstract reasoning tasks. *CoRR*, abs/2103.05823, 2021. URL https://arxiv.org/abs/2103.05823.

Kaggle Leaderboard. https://www.kaggle.com/competitions/abstraction-and-reasoning-challenge/code, 2020. Accessed: 2024-30-01.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.

Victor Kolev, Bogdan Georgiev, and Svetlin Penkov. Neural abstract reasoner. *arXiv preprint arXiv:2011.09860*, 2020.

Alexandre Laterre, Yunguan Fu, Mohamed Khalil Jabri, Alain-Sam Cohen, David Kas, Karl Hajjar, Hui Chen, Torbjørn S Dahl, Amine Kerkeni, and Karim Beguir. Ranked reward: enabling self-play reinforcement learning for bin packing. 2019.

Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

Suvir Mirchandani, Fei Xia, Pete Florence, Brian Ichter, Danny Driess, Montserrat Gonzalez Arenas, Kanishka Rao, Dorsa Sadigh, and Andy Zeng. Large language models as general pattern machines. *arXiv preprint arXiv:2307.04721*, 2023.

Ansong Ni, Jeevana Priya Inala, Chenglong Wang, Oleksandr Polozov, Christopher Meek, Dragomir Radev, and Jianfeng Gao. Learning math reasoning from self-sampled correct and partially-correct solutions, 2023.

Jaehyun Park, Jaegyun Im, Sanha Hwang, Mintaek Lim, Sabina Ualibekova, Sejin Kim, and Sundong Kim. Unraveling the arc puzzle: Mimicking human solutions with object-centric decision transformer. *arXiv preprint arXiv:2306.08204*, 2023.

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. *CoRR*, abs/2202.01344, 2022. URL https://arxiv.org/abs/2202.01344.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419): 1140–1144, 2018.

Avi Singh, John D. Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J. Liu, James Harrison, Jaehoon Lee, Kelvin Xu, Aaron Parisi, Abhishek Kumar, Alex Alemi, Alex Rizkowsky, Azade Nova, Ben Adlam, Bernd Bohnet, Gamaleldin Elsayed, Hanie Sedghi, Igor Mordatch, Isabelle Simpson, Izzeddin Gur, Jasper Snoek, Jeffrey Pennington, Jiri Hron, Kathleen Kenealy, Kevin Swersky, Kshiteej Mahajan, Laura Culp, Lechao Xiao, Maxwell L. Bileschi, Noah Constant, Roman Novak, Rosanne Liu, Tris Warkentin, Yundi Qian, Yamini Bansal, Ethan Dyer, Behnam Neyshabur, Jascha Sohl-Dickstein, and Noah Fiedel. Beyond human data: Scaling self-training for problem-solving with language models, 2023.

Elizabeth S. Spelke and Katherine D. Kinzler. Core knowledge. *Developmental Science*, 10 (1):89–96, 2007. doi: https://doi.org/10.1111/j.1467-7687.2007.00569.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-7687.2007.00569.x.

Ruocheng Wang, Eric Zelikman, Gabriel Poesia, Yewen Pu, Nick Haber, and Noah D Goodman. Hypothesis search: Inductive reasoning with language models. *arXiv preprint arXiv:2309.05660*, 2023a.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint*, 2023b.

Ziqi Wang, Le Hou, Tianjian Lu, Yuexin Wu, Yunxuan Li, Hongkun Yu, and Heng Ji. Enable language models to implicitly learn self-improvement from data, 2023c.

Yudong Xu, Elias B. Khalil, and Scott Sanner. Graphs, constraints, and search for the abstraction and reasoning corpus, 2022.

Yudong Xu, Wenhao Li, Pashootan Vaezipoor, Scott Sanner, and Elias B Khalil. Llms and the abstraction and reasoning corpus: Successes, failures, and the importance of object-based representations. *arXiv preprint arXiv:2305.18354*, 2023.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models, 2023.

A METHOD AND EVALUATION DETAILS

A.1 CODEIT ALGORITHM

The pseudo code for the CodeIt procedure is portrayed in Algorithm 1.

Algorithm 1 CodeIt Algorithm

Require: Training set D_{train}, search set D_{test}, policy Q
Ensure: Finetuned policy Q, updated replay buffer R, optimal programs set ρ^*
  $D_{\text{augmented_train}} \leftarrow$ EvolveTrainingTasks(D_{train}) {Evolve training tasks}
  Initialize ρ^* as an empty set {Init set of programs that solve tasks in D_{test}}
  for meta_iter $= 1 \rightarrow 100$ **do**
    # Sampling and hindsight relabeling stage
    for task in D_{test} **do**
      $\{\rho\} \leftarrow Q(\rho|\{I, O\})$ {Sample programs for test tasks}
      for each ρ in $\{\rho\}$ **do**
        if SyntactticallyValid(ρ) **then**
          Add $\{\rho, \{(I^{(i)}, \rho(I^{(i)})), \dots\}\}$ to R {Update the replay buffer with hindsight relabeled tasks}
        end if
        for $(I^{(i)}, O^{(i)})$ in task **do**
          if $\rho(I^{(i)}) = O^{(i)}$ **then**
            Add $\{(\rho, \text{task})\}$ to ρ^* {Update set of programs that solve tasks in D_{test}}
          end if
        end for
      end for
    end for
    # Learning stage
    $D_{\text{sample}} \leftarrow$ SampleFrom($R + D_{\text{augmented_train}} + D_{\text{train}}$) {Sample tasks from the replay buffer}
    Train Q on D_{sample} for 1 epoch {Continual training of the policy}
  end for

Initializing CodeIt Before we start the CodeIt procedure, we expand the training dataset using the first 19,200 mutated tasks from the mutation procedure (see Appendix A.2) used for the mutation d_1 baseline.

A.2 PROGRAM AND TASK MUTATION

Mutation procedure To grow a population of mutated programs with task demonstration inputs corresponding to the original training dataset, we follow the procedure outlined in Algorithm 3. This involves mutating a single task, which is described in Algorithm 2. The mutation is carried out with the hyperparameters $\phi_{\text{var}} = 0.25, \phi_{\text{arg}} = 0.5, \phi_{\text{func}} = 0.25$. With respect to naming notation, d_1 reflects a depth of 1, meaning we only mutate programs from the original training set, and d_∞ reflects a depth of infinity, meaning we can mutate previously mutated programs.

The intuitive explanation of the mutation procedure for a single program is as follows. We pick a random line from a program (L2-3). We then replace either a function call with a function with similar output type (L4-7), or we replace an input argument in the function call (L8-11), or we replace the function call but leave its input variables the same (L12-14).

Mutation baseline For our mutation baseline, we sample mutated programs using the mutation procedure outlined above. For all the mutated programs in the evolved task population, we evaluate each program on the tasks in our search set.

Algorithm 2 MutateProgram

Require: Replacement probabilities ϕ_{var}, ϕ_{arg}, ϕ_{func}, program ρ
Ensure: ρ'
  Initialize $\rho' \leftarrow \rho$ {Copy original program}
  $l \leftarrow$ RandomLineFrom(ρ') {Randomly select a line}
  $p \sim U(0, 1)$
  if $p < \phi_{\text{var}}$ **then**
    $f' \leftarrow$ SampleFunctionWithOutputType(GetTypeOfVariable(l))
    $args' \leftarrow$ SampleArgumentsForFunction(f')
    Replace variable definition $f(args)$ in l with $f'(args')$
  else if $p < (\phi_{\text{var}} + \phi_{\text{arg}})$ **then**
    $a \leftarrow$ RandomArgumentFrom(l)
    $a' \leftarrow$ SampleTermOfType(GetTypeOfArgument(a))
    Replace argument a with a'
  else
    $f' \leftarrow$ SampleFunctionOfType(GetTypeOfFunction(f))
    Replace function f in l with f'
  end if

Algorithm 3 EvolveTrainingTasks

Require: Initial population of training tasks T_{init} (each task is a tuple $(\rho, \mathcal{E})$ where $\mathcal{E} = \{(I^{(i)}, O^{(i)}), \dots\}$), depth
Ensure: Updated task population T' (initialized with T_{init})
  $T \leftarrow T_{\text{init}}$
  $i \leftarrow 0$
  while $i < num_samples$ **do**
    if depth $= 1$ **then**
      $(\rho, \mathcal{E}) \leftarrow$ RandomSelectTask(T_{init}) {Select from initial tasks}
    else
      $(\rho, \mathcal{E}) \leftarrow$ RandomSelectTask(T) {Select from current tasks}
    end if
    $\rho' \leftarrow$ MutateProgram(ρ)
    $\mathcal{E}' \leftarrow \emptyset$ {Initialize mutated task demonstration examples}
    for each $(I^{(k)}, _) \in \mathcal{E}$ **do**
      $O'^{(k)} \leftarrow$ Execute($\rho', I^{(k)}$)
      $\mathcal{E}' \leftarrow \mathcal{E}' \cup \{(I^{(k)}, O'^{(k)})\}$
    end for
    if AreValidGrids(GetAllOutputs($\mathcal{E}'$)) **then**
      $T' \leftarrow T' \cup \{(\rho', \mathcal{E}')\}$ {Add new task to the population}
    end if
    $i \leftarrow i + 1$
  end while

A.3 TASK REPRESENTATION

Grid representation We use a compressed grid representation, mainly to reduce the number of tokens needed to represent each grid. An example of the sparse grid representation is shown in Figure 7 for a toy ARC task.

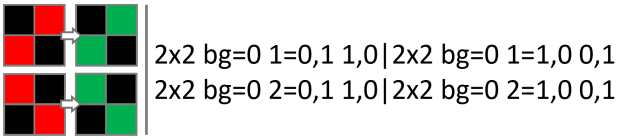


Figure 7: Sparse grid representation of a simplified ARC task.

This object-centric text representation, similar to the one of Xu et al. (2023), works well for sparse grids and is human-interpretable. We do not use a custom tokenizer. A visualization of the number of tokens is shown in Fig. 11, showing that in almost all cases, the sparse grid representation we use leads to a reduction in the number of needed tokens, especially for larger grid sizes.

Truncation We truncate our task demonstration tokens and program tokens such that these sequences fit in our predefined encoder and decoder context windows. For the task demonstration examples, we first order by grid size and divide the encoder context window into two equally sized sections. For task demonstration inputs, we first encode input grids to text as above and then we tokenize using the standard text tokenizer. We truncate these tokens at half the size of the encoder context window. We do the same for the task demonstration outputs and with the exception of also adding an end of sequence token. As a result, even though we aim to show the policy up to ten task demonstration examples, large grids will be cut-off. For programs, we tokenize directly using the standard text tokenizer and truncate at the decoder context window size.

A.4 ARC EVALUATION

Different works use different evaluation procedures to report performance on the ARC evaluation set. We describe two common evaluation settings in more detail below. Unless mentioned otherwise, we always use the first procedure, "ARC Eval Set".

ARC Eval Set This setup is intended as close as possible to the evaluation procedure described by Chollet (2019). Baselines Ferré (2021), Ainooson et al. (2023) follow this procedure, and it is our default setting as well.

The ARC eval set consists of 400 tasks, some of which contain multiple test examples. Common procedure is to report pass@3 performance, meaning the top 3 solutions are selected according to demonstration task performance. If there are ties, we favor the shorter program, under the assumption that shorter programs are more likely to generalize. We then run these programs on all test examples for the task. In some cases, there are multiple test examples per task. We call the task "solved" if all output grids are correct.

ARC Eval 412 This setup is designed to match Gendron et al. (2023). Instead of calling a task with multiple test examples solved if all test outputs are correct, distinct tasks are created - one per test example. This results in a set of 412 evaluation tasks with one test example each. Furthermore, Gendron et al. (2023) uses pass@1, rather than pass@3: only one solution per task is evaluated, and the task is considered solved if the output is correct.

B EXPERIMENT DETAILS

B.1 RESOURCES

Experiments were run for a maximum of 120 hours on a NVIDIA A100 80GB.

B.2 HYPERPARAMETER TUNING

Dataset The ARC benchmark does not contain a validation split. Hence, we use part of the ARC train split for validation during the hyperparameter tuning. In particular, this validation set is the search set that the sampling stage uses as described in 2.2. With this setup we avoid overfitting the hyperparameters to the ARC evaluation split.

We choose the split such that $\mathcal{D}$ and $\mathcal{D}_{\text{valid}}$ contain roughly equally difficult programs by sampling based on program length: $\mathcal{D}$ contains 80% of 2-line programs, 80% of 3-line programs, and so on. This results in 311 examples in $\mathcal{D}$ and 89 examples in $\mathcal{D}_{\text{valid}}$.

Experiments on validation set In these experiments, we initialise our replay buffer with the 311 $\mathcal{D}$ examples, and our search set consists of the 89 $\mathcal{D}_{\text{valid}}$ examples. The aim of these experiments is to find optimal hyper-parameters for search and training. A list of our tuned hyperparameter values and their description is shown in Tab. 3

B.3 HYPERPARAMATERS CHOSEN ON INTERNAL VALIDATION SET

We optimized these parameters on our custom validation set before applying CodeIt to ARC eval.

CodeIt stage	Param	Value	Description
Sampling	n_ρ	24	no. policy samples ρ per task per meta-iteration[1]
	n_m	$19,200$	no. mutated samples for augmented train set[1]
	τ	0.95	sampling temperature
	r_t	$10,000$	number of experiences sampled from augmented train set
	r_p	$90,000$	number of experiences sampled from buffer
Learning	n_ϵ	1	no. train epochs per meta-iteration
	lr	$5e-5$	learning rate

Table 3: Table of hyperparameters.

B.4 DOMAIN SPECIFIC LANGUAGE

We adopt the domain specific language (DSL) of Michael Hodel, made available on GitHub: https://github.com/michaelhodel/arc-dsl. This DSL was designed based on the training set: the (human) designer did not peek at the evaluation set. This is what allows us to run search on ARC eval here. Using a DSL designed for the eval tasks would be cheating, as we would benefit immensely from human insights captured in the primitives. On the other hand, it may mean that some ARC eval programs are not solvable with the current DSL.

The DSL is implemented in https://github.com/michaelhodel/arc-dsl/blob/main/dsl.py. It contains many basic grid manipulation operations, such as rotations (`rot90, rot180, rot270`), mirroring (`dmirror`, `hmirror`, `vmirror`), resizing (`downscale`, `upscale`), or concatenation (`hconcat`, `vconcat`). It also contains functions that perform counting, for example `numcolors` counts the number of colors occurring in an object or grid. For some ARC tasks, identifying the foreground objects and determining how these objects interact is an effective strategy for human test-takers. Therefore, some functions also apply to "objects", which are patches of the same color that stand out from the background. To extract these, the function `objects` returns the set of foreground objects, i.e. those that have a different color than the most common color, assumed to be the background. For a complete list of primitives and their description, we refer the reader to the aforementioned Github page.

Michael Hodel provides hand-designed solution programs for all training tasks in https://github.com/michaelhodel/arc-dsl/blob/main/solvers.py. Some programs are highly complex: for some of the more challenging ARC tasks, we see solutions consisting of up to 58 lines of code (`solve_b775ac94`). We use these 400 solution programs to kickstart CodeIt training.

[1]Note that no. samples here refers to policy and mutation samples before filtering for syntactic correctness.

Method	Number of tasks solved
CodeIt policy only	23
Mutation d_1 only	13
CodeIt policy $\cap$ Mutation d_1	29

Table 4: ARC evaluation tasks solved per method. The top group of two rows show how many tasks were solved by a method, but not by the other. The final row shows tasks solved by both methods.

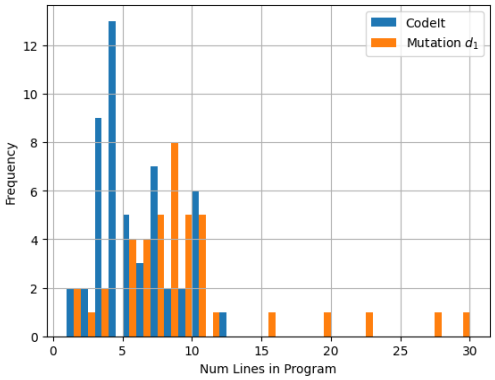


Figure 8: Histogram of number of lines for tasks where both CodeIt and Mutation produced solutions. CodeIt (in blue) typically produces shorter programs than the Mutation baseline (in orange).

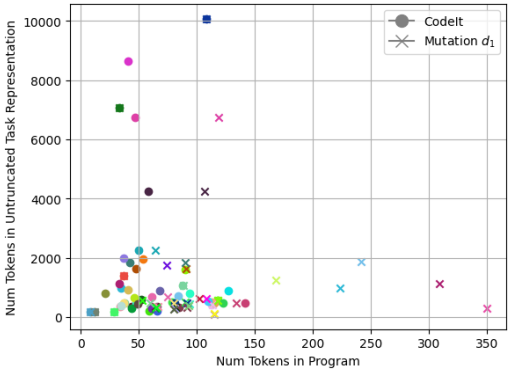


Figure 9: Number of task representation tokens vs number of program tokens. Colors represents the different tasks. We see no obvious correlation between task representation length and program length.

C Program analysis

C.1 CodeIt compared with mutation baselines

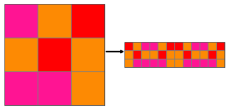
We compare the programs found using our mutation d_1 baseline and the best performing of the three CodeIt runs. Table 4 displays the number of ARC evaluation tasks uniquely solved by each method and the tasks which are solved by multiple methods. CodeIt's policy solves 52 of 400 tasks, 23 of which were not solved by the mutation baseline. In Figures 8 and 9, we select the shortest program that solves an evaluation task for CodeIt and our mutation d_1 baseline, computing the program length and task representation size. Note that CodeIt has an encoder context window size of 1024 and so any tasks which having representations of more than 1024 tokens have been truncated. Overall, CodeIt finds shorter programs as shown in 8. Further, for the same task, CodeIt more often finds shorter programs than our mutation d_1 baseline, as shown in 9 where each color represents a different task. Interestingly, CodeIt does solve some tasks with very large task representations, suggesting in some cases a truncated task representation provides sufficient information to solve the task.

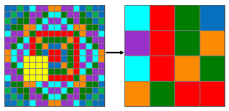
In Table 5, we show a subset of solution programs for ARC eval tasks solved by both CodeIt and our mutation d_1 baseline. We select tasks where the shortest programs differ between the two methods. CodeIt programs appear more concise and use different primitives. Out of the 29 tasks that are solved by both methods, there are 24 shortest programs where method output is different. CodeIt only produces a longer program in 1 out of these 24 cases. The Mutation baseline often includes redundant lines. In addition, for many programs, CodeIt produces a program that is qualitatively better: the solution is less complex, and contains fewer lines overall.

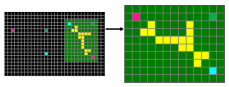
C.2 CodeIt over time

Since we do not have ground truth programs for the ARC evaluation set, we treat the shortest program found with demonstration performance and test performance equal to 1 for each task over all the meta-iterations as a proxy for the ground truth program. To examine how CodeIt solutions change over time, we take the subset of ARC evaluation tasks where the best performing CodeIt run finds

CodeIt Policy	Mutation $d1$	Test Example

`x1 = vmirror(I)`
`x2 = hconcat(x1, I)`
`O = hconcat(x2, x2)`
	`x1 = vmirror(I)`
	`x2 = hconcat(x1, I)`
	`x3 = hmirror(x2)`
	`x4 = vconcat(x2, x3)`
	`x5 = hconcat(x3, x3)`
	`O = hmirror(x5)`
		

`x1 = compress(I)`
`x2 = ofcolor(I, THREE)`
`x3 = rot90(x1)`
`O = subgrid(x2, x3)`
	`x1 = hmirror(I)`
	`x2 = vmirror(I)`
	`x3 = ofcolor(I, THREE)`
	`x4 = subgrid(x3, x1)`
	`x5 = subgrid(x3, x2)`
	`x6 = palette(x4)`
	`x7 = contained(ONE, x6)`
	`O = branch(x7, x5, x4)`
		

`x1 = ofcolor(I, ONE)`
`x2 = subgrid(x1, I)`
`O = cmirror(x2)`
	`x1 = mostcolor(I)`
	`x2 = objects(I, T, F, T)`
	`x3 = replace(I, x1, THREE)`
	`x4 = argmax(x2, size)`
	`x5 = argmin(x2, size)`
	`x6 = position(x4, x5)`
	`x7 = first(x6)`
	`x8 = last(x6)`
	`x9 = subgrid(x4, x3)`
	`x10 = hline(x5)`
	`x11 = hmirror(x9)`
	`x12 = vmirror(x9)`
	`x13 = branch(x10, x11, x12)`
	`x14 = branch(x10, x7, ZERO)`
	`x15 = branch(x10, ZERO, x8)`
	`x16 = asobject(x13)`
	`x17 = matcher(first, THREE)`
	`x18 = compose(flip, x17)`
	`x19 = sfilter(x16, x18)`
	`x20 = ulcorner(x4)`
	`x21 = shape(x4)`
	`x22 = astuple(x14, x15)`
	`x23 = multiply(x21, x22)`
	`x24 = add(x20, x23)`
	`x25 = shift(x19, x24)`
	`x26 = rot270(x11)`
	`O = paint(x26, x25)`
		

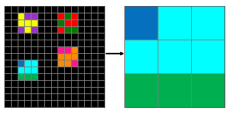
`x1 = objects(I, F, F, T)`
`x2 = argmax(x1, numcolors)`
`O = subgrid(x2, I)`
	`x1 = objects(I, F, F, T)`
	`x2 = leastcolor(I)`
	`x3 = rbind(colorcount, x2)`
	`x4 = argmax(x1, x3)`
	`O = subgrid(x4, I)`
		

Table 5: Selection of shortest programs for ARC evaluation tasks solved by CodeIt policy (left) and the Mutation d_1 baseline (right) for which CodeIt program is shorter.

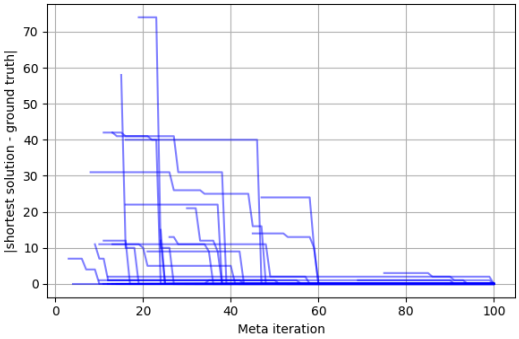


Figure 10: Difference in number of tokens between the shortest solution found per meta-iteration and shortest solution found by the final meta-iteration for best performing CodeIt run.

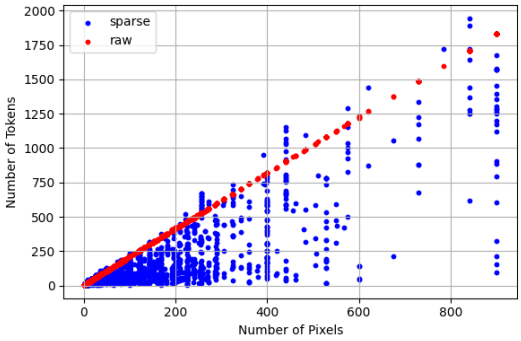


Figure 11: Grid size versus token count for the ARC training data. The sparse grid representation is typically shorter than the raw grid representation.

such programs; this leaves us 45 tasks. We observe that once CodeIt finds a solution, CodeIt often continues to find both longer and shorter solutions in later meta-iterations. We pose that this gives the potential for program refinement, however, since the priority does not incorporate length, there is not explicit bias towards shorter solutions and so both longer and shorter solutions will be learned from. We observe that out of the 45 tasks, the best performing CodeIt run finds shorter solutions over time in 24 tasks as shown in Figure 10.

In Tables 6, we show a selection of examples where the best performing CodeIt run finds a longer solution in an earlier meta-iteration and shorter solution in a later meta-iteration.

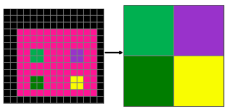
D ARC COMPETITIONS

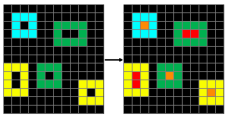
Various unpublished work exist including submissions to ARC challenges as well as a Kaggle competition. These competitions use a private leaderboard, not revealed to participants. This means participants often use the public ARC evaluation set for training or DSL design purposes. For example, the winner of Kaggle 2020 comments that searching in a DSL designed using the training set resulted in low performance, and higher performance was reached after conditioning the DSL on the evaluation tasks (Icecuber, 2020). This makes direct comparisons to methods evaluated on the evaluation set difficult. For reference, we include a summary of competition results in Table 7, however, note that this summary reports performance on the hidden test set, and that competition results cannot not be directly compared to this work and the literature.

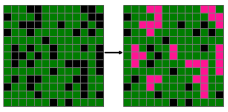
Competition	Winner	Method	Perf.
Kaggle 2020	Icecuber (2020)	Search in eval set DSL*	21%
Kaggle 2020 late	Multiple (Kaggle Leaderboard, 2020)	Ensemble previous entries*	30%
ARCathon 2022	Hodel (2023)	Search in CodeIt DSL	6%
ARCathon 2023	Multiple (ARCathon Leaderboard, 2023)	Unknown	30%

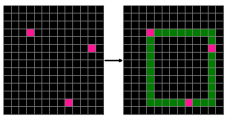
Table 7: Performance on Hidden Test Set for Various ARC Competition Winners. *Method conditions on ARC evaluation set.

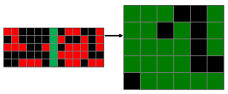
Early Shortest Solution	Later Shortest Solution	Test Example

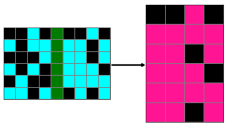
`x1 = ofcolor(I, EIGHT)`
`x2 = replace(I, EIGHT, ZERO)`
`x3 = compress(x2)`
`O = downscale(x3, TWO)`
	`x1 = replace(I, EIGHT, ZERO)`
	`x2 = compress(x1)`
	`O = downscale(x2, TWO)`
		

`x1 = objects(I, T, F, T)`
`x2 = apply(delta, x1)`
`x3 = mfilter(x2, square)`
`x4 = fill(I, FIVE, x3)`
`x5 = objects(x4, F, F, T)`
`x6 = mapply(delta, x5)`
`x7 = fill(x4, SEVEN, x6)`
`O = fill(x7, FIVE, x3)`
	`x1 = objects(I, T, F, T)`
	`x2 = apply(delta, x1)`
	`x3 = mfilter(x2, square)`
	`x4 = fill(I, FIVE, x3)`
	`x5 = objects(x4, F, F, T)`
	`x6 = mapply(delta, x5)`
	`O = fill(x4, SEVEN, x6)`
		

`x1 = objects(I, T, F, F)`
`x2 = colorfilter(x1, ZERO)`
`x3 = sizefilter(x2, ONE)`
`x4 = difference(x2, x3)`
`x5 = merge(x4)`
`O = fill(I, EIGHT, x5)`
	`x1 = objects(I, T, F, T)`
	`x2 = sizefilter(x1, ONE)`
	`x3 = difference(x1, x2)`
	`x4 = merge(x3)`
	`O = fill(I, EIGHT, x4)`
		

`x1 = vmirror(I)`
`x2 = fgpartition(I)`
`x3 = compose(outbox, inbox)`
`x4 = mapply(x3, x2)`
`O = underfill(I, ONE, x4)`
	`x1 = ofcolor(I, EIGHT)`
	`x2 = box(x1)`
	`O = underfill(I, ONE, x2)`
		

`x1 = lefthalf(I)`
`x2 = righthalf(I)`
`x3 = ofcolor(x1, ZERO)`
`x4 = ofcolor(x2, ZERO)`
`x5 = intersection(x3, x4)`
`x6 = shape(x1)`
`x7 = canvas(ONE, x6)`
`O = fill(x7, ZERO, x5)`
	`x1 = lefthalf(I)`
	`x2 = righthalf(I)`
	`x3 = cellwise(x1, x2, ONE)`
	`O = replace(x3, SEVEN, ONE)`
		

`x1 = lefthalf(I)`
`x2 = righthalf(I)`
`x3 = ofcolor(x1, FOUR)`
`x4 = ofcolor(x2, FOUR)`
`x5 = combine(x3, x4)`
`O = fill(x1, EIGHT, x5)`
	`x1 = lefthalf(I)`
	`x2 = righthalf(I)`
	`x3 = cellwise(x1, x2, FOUR)`
	`O = replace(x3, FOUR, EIGHT)`
		

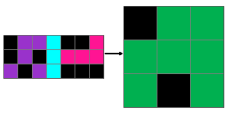
`x1 = lefthalf(I)`
`x2 = righthalf(I)`
`x3 = ofcolor(x1, ZERO)`
`x4 = ofcolor(x2, ZERO)`
`x5 = intersection(x3, x4)`
`x6 = shape(x1)`
`x7 = canvas(TWO, x6)`
`O = fill(x7, ZERO, x5)`
	`x1 = vmirror(I)`
	`x2 = lefthalf(I)`
	`x3 = righthalf(I)`
	`x4 = cellwise(x2, x3, TWO)`
	`O = replace(x4, EIGHT, TWO)`
		

Table 6: Selection of shortest solutions for ARC evaluation tasks solved by CodeIt policy where shorter solutions are found over time.