Dreaming with ARC

Anonymous Author(s)
Affiliation
Address
email

Abstract

Current machine learning algorithms are highly specialized to whatever it is they are meant to do — e.g. playing chess, picking up objects, or object recognition. How can we extend this to a system that could solve a wide range of problems? We argue that this can be achieved by a modular system — one that can adapt to solving different problems by changing only the modules chosen and the order in which those modules are applied to the problem. The recently introduced ARC (Abstraction and Reasoning Corpus) dataset serves as an excellent test of abstract reasoning. Suited to the modular approach, the tasks depend on a set of human Core Knowledge inbuilt priors. In this paper we implement these priors as the modules of our system. We combine these modules using a neural-guided program synthesis.

1 Introduction

Deep Learning has been hugely successful in recent years, with ever more impressive feats of (super)-human level skills in object recognition [1] and playing games [2–4]. Can this progress continue, if we only work with bigger models, or is there some limit in this skill acquisition? There are at least two reasons to believe we might need a new approach:

- While a given neural network might be highly skilled in any one task, we are still far away from a single model that could excel at a variety of tasks and generalizing beyond those that it has been trained on (even as simple as playing Go on a bigger board size) [5].
- Training (recurrent) neural networks with gradient descent on more abstract tasks, such as checking parity, counting or verifying which pixels are inside/outside a shape, has been known for some time now to fail to generalize beyond the scope of the training set [6–12].

How is that humans exhibit the ability to quickly learn new skills? The dual-process theory of reasoning [13–15] suggests that our abilities stem from the interaction between a fast, associative system (similar to that of modern deep nets) and a slower symbolic system. This highly motivates the need for a neural implementation of symbolic AI, which has seen some recent work, particularly in the space of mathematical reasoning [16–20]. The interaction between the two systems could be implemented by a modular system – having small (neural) modules [21–24, 21] capable of solving specific problems and the ability to combine them using a neural controller. If the modules are together capable of performing an efficient set of complete operations (for example in the sense of being Turing universal), we would expect such a system to be capable of solving general tasks. This is the setting of neurally-guided program synthesis [25–28], which we follow in this paper.

To evaluate the ability of generalizing to new tasks, we want to consider the concept of developer-aware generalization [29] — which measures the ability to solve problems the developer of the algorithm has not encountered before. [29] introduced the ARC (Abstraction and Reasoning Corpus) dataset — a set of IQ-like tasks that are all unique — each task consists of 2-4 training examples and

Submitted to 34th Conference on Neural Information Processing Systems (NeurIPS 2020). Do not distribute.

one or more test examples. Importantly, the tasks never repeat and depend on a set of human Core Knowledge inbuilt priors (such as the notion of objectness, simple arithmetic abilities and geometrical knowledge). This is a highly challenging dataset, with a hard-coded brute force Kaggle-competition winning solution achieving only $\sim 20\%$ performance on a hidden test set [30].

In this paper, we adapt the newly introduced Dreamcoder program synthesis framework to solving the ARC dataset, with the modules of our system (some hard-coded, some in the form of neural nets) given by the set of core priors identified in [29] . We show that such a system is capable of generalizing by learning new primitives, going beyond the limitations of its initial knowledge. We further show that using neural guidance, we can exponentially reduce the search space of programs, from $O(n!)$ programs down to only $O(n)$.

1.1 Introduction to Dreamcoder

The foundation of our modular approach is Dreamcoder, a recent program synthesis approach well suited to the ARC dataset [31]. Dreamcoder represents modules as functional programs. As it solves tasks, it learns new modules through "compression," a process which distills higher-level modules out of existing task solutions. This allows Dreamcoder to solve new tasks that it would not have been able to solve with its original library. Dreamcoder also trains a neural network to learn to recognize which modules are most likely to solve a given task. Together, compression and neural-guided synthesis allow Dreamcoder to gradually acquire expertise in an area. For example, it rediscovers laws of classical physics (including Newton's and Coulomb's laws) from much simpler modules, by compositionally building on concepts from those learned earlier.

As a simple example, given only an addition module, Dreamcoder can learn to solve multiplication tasks through repeated addition. Then, during "compression," it refactors these multiplication programs to express them in terms of a higher-level multiplication module. This new module can be used to solve more difficult tasks such as calculating factorials. Here, we build off of Dreamcoder to solve the ARC dataset, by providing it with the human priors identified in [29] as the basic modules.

2 Modules & Program Synthesis

2.1 Enabling Generalization through Compression

The compression component of Dreamcoder is crucial to our program synthesis approach. After each iteration of attempting to solve ARC tasks, our agent looks at all of the correct programs, notices structures that were similar between different solved programs, and then *re-writes new, higher-level programs based on lower-level programs*. Compression enables our agent to learn new techniques and behaviors based on the tasks it is solving, rather than being limited to the tools the developer provided it with. This type of generalization ability is at the heart of the ARC challenge — creating a machine that quickly learns to solve problems its developers might not have anticipated.

We demonstrate how the synthesizer can create more abstract modules from existing modules in the following experiment. First, we supply our agent with six tasks (meant to be similar to ARC tasks): drawing a line in three different directions, and moving an object in three different directions. The programs synthesized are the following:

```
(lambda (rotate_cw (draw_line_down (rotate_ccw $0)))) // draws line left
(lambda (rotate_cw (move_down (rotate_ccw $0)))) // moves object left
(lambda (rotate_ccw (draw_line_down (rotate_cw $0)))) // draws line right
(lambda (rotate_ccw (move_down (rotate_cw $0)))) // moves object right
(lambda (rotate_cw (rotate_cw (draw_line_down (rotate_cw (rotate_cw $0)))))) // draws line up
(lambda (rotate_cw (rotate_cw (move_down (rotate_cw (rotate_cw $0)))))) // moves object up
```

(a) Training data for drawLineLeft task (b) Training data for moveObjectLeft task

Figure 1: Sample tasks involving applying an action towards the left

After looking at only two different programs each for applying actions in each of the three different directions, the refactoring and compression process results in about ten new modules. Three of the generated modules are the following general "apply action in direction" programs:

```
(lambda (lambda (rotate_cw ($0 (rotate_ccw $1))))) // applies action left
(lambda (lambda (rotate_ccw ($0 (rotate_cw $1))))) // applies action right
(lambda (lambda (rotate_cw (rotate_cw ($0 (rotate_cw (rotate_cw $1))))))) // applies action up
```

Thus, instead of our agent developing tunnel-vision and just becoming more and more suited to doing certain kinds of trained tasks, it is able to generalize knowledge and can then apply this knowledge to other tasks completely unrelated to drawing lines or moving objects.

2.2 Enabling generalization on ARC symmetry tasks

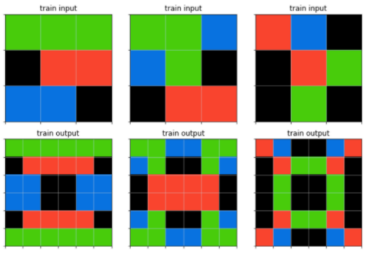
In a second experiment, we demonstrate how compression-based learning enables developer-aware generalization on the ARC dataset. We provide Dreamcoder with a set of five grid-manipulation modules — flipping vertically with `vertical_flip`, rotating clockwise with `rotate_cw`, overlaying two grids with `overlay`, stacking two grids vertically with `stack_vertically`, and getting the left half of a grid with `left_half`. We then train our agent on a subset of 36 ARC training tasks involving symmetry over five iterations of enumeration and compression. During each iteration, our agent attempts to solve all 36 tasks by enumerating possible programs for each task. It then runs compression to create new modules. During the next iteration, the agent tries to solve all tasks again within the same amount of time but equipped with the new modules. In this experiment, our agent solves 16 tasks before any training. After one iteration and new modules, it solves 17 in the same amount of time. After another, it solves 19 tasks, and after the final iteration, it solves 22.

After each iteration, our agent learns new modules which help it solve tasks that were previously too difficult. Thus, the Dreamcoder compression framework enables our agent to learn interpretable, compositional modules not provided by the developer, such as flipping horizontally, rotating counter-clockwise, and stacking grids horizontally. It uses these new modules to solve progressively harder tasks. The most difficult tasks solved involve mirroring the input grid four ways, requiring a synthesized program which is extremely long when expressed in terms of the original modules.

This experiment shows a promising path towards the developer-aware generalization required to succeed on the ARC dataset. In order to solve unknown tasks in the test set, our agent will need to learn from the tasks themselves. As shown in this experiment, Dreamcoder is able to learn new concepts based on tasks given, which enable it to solve more difficult tasks.

Action	Code
mirror across diagonal	`#(lambda (rotate_cw (vertical_flip $0)))`
rotate 180 degrees	`#(lambda (rotate_cw (rotate_cw (input $0))))`
flip horizontally	`#(lambda (rotate_cw (rotate_cw (#(lambda (vertical_flip (input $0))) $0))))`
rotate counterclockwise	`#(lambda (rotate_cw (#(lambda (rotate_cw (rotate_cw (input $0)))) $0)))`
stack grids horizontally	`#(lambda (lambda (#(lambda (rotate_cw (vertical_flip $0))) (stack_vertically (#(lambda (rotate_cw (#(lambda (vertical_flip (input $0))) $0))) $1) (#(lambda (rotate_cw (vertical_flip $0))) $0)))))`

Figure 2: Useful actions learned in the process of solving symmetry tasks. Pound signs represent new modules. New modules may rely on others for construction; e.g. to stack grids horizontally, we reflect each input diagonally, stack vertically, and reflect the vertical stack diagonally.




```
lambda (stack_vertically (#(lambda (lambda (#(lambda (lambda
(#(lambda (rotate_cw (vertical_flip $0))) (stack_vertically
(#(lambda (rotate_cw (#(lambda (vertical_flip (input $0))) $0)))
$1) (#(lambda (rotate_cw (vertical_flip $0))) $0))))) $0 (#(lambda
(rotate_cw (rotate_cw (#(lambda (vertical_flip (input $0))) $0))))
$1)))) $0 $0) (vertical_flip (#(lambda (lambda (#(lambda (lambda
(#(lambda (rotate_cw (vertical_flip $0))) (stack_vertically
(#(lambda (rotate_cw (#(lambda (vertical_flip (input $0))) $0)))
$1) (#(lambda (rotate_cw (vertical_flip $0))) $0))))) $0 (#(lambda
(rotate_cw (rotate_cw (#(lambda (vertical_flip (input $0))) $0))))
$1)))) $0 $0))))
```

Figure 3: One of the four-way mirroring tasks and the program discovered that solves it. The program was discovered only after four iterations of enumeration and compression.

2.3 Neural-guided synthesis

Guiding program enumeration with a neural network is a commonly used program synthesis technique to speed up search, and is included in Dreamcoder's synthesis approach. We showcase the appropriateness of this approach for the ARC dataset by comparing neural-guided synthesis with brute-force enumeration on a set of artificial ARC-like tasks involving sorting blocks of various sizes. Training a neural network improves the space of possible programs considered for a given task from $O(n!)$ to roughly $O(n)$ for a given program length. The network outputs a distribution over the set of modules using a convolutional network over the input/output grids. Doing so exponentially speeds up discovery of task solutions, as shown in the figure below.

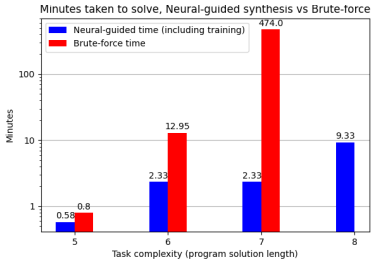


Figure 4: Using a neural net to guide synthesis exponentially improves enumeration time.

3 Discussion

It is useful to compare the learning done in our approach to that done by neural networks. Neural networks can also learn new concepts from training examples, but their internal representation lacks structure which allows them to apply learned concepts compositionally to other tasks. In contrast, modules learned via compression, represented as programs, can naturally be composed and extended to solve harder tasks, while reusing modules between tasks. This constitutes a learning paradigm which we view as essential to human-like reasoning.

Even so, there are aspects of neural network functionality we can take advantage of. Some human priors such as object detection and denoising might be best implemented through neural networks rather than as functional programs. Incorporating neural modules into program synthesis would allow us to benefit from advantages of neural networks, while also benefiting from the advantage of writing modules as programs. This is an important next step of our work. Furthermore, the current neural network that guides program synthesis is currently limited to a bigram distribution over the library. In practice, humans use more sophisticated reasoning when determining the solution to a task, and we would like to incorporate such reasoning into our agent's approach to ARC.

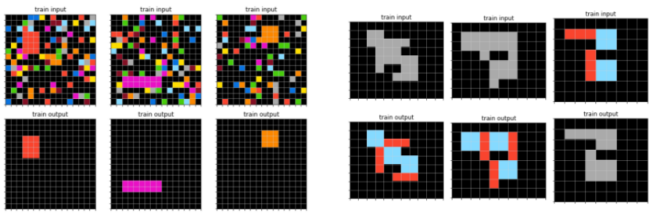


Figure 5: Left: 2 tasks involving denoising. Right: 2 tasks requiring more sophisticated reasoning.

We believe our approach is particularly well-suited to the ARC dataset, which features various symbolic operations combined in varying levels of difficulty, as exemplified in the symmetry tasks which increase in difficulty. In addition, the use of a hidden test set to measure performance means that a successful ARC agent will have to be able to learn concepts beyond what it is given by the developer. Existing approaches to ARC involve carefully crafting a static library of modules which allows difficult programs to be enumerated easily. These approaches are an important baseline, but do not exhibit any sort of learning, unlike the approach here.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[3] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[4] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[5] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning, 2019.

[6] Kimberly M. Villalobos, Vilim Stih, Amineh Ahmadinejad, Shobhita Sundaram, Jamell Dozier, Andrew Francl, Frederico Azevedo, Tomotake Sasaki, and Xavier Boix. Do neural networks for segmentation understand insideness? 04/2020 2020.

[7] Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets, 2015.

[8] Shai Shalev-Shwartz, Ohad Shamir, and Shaked Shammah. Failures of gradient-based deep learning, 2017.

[9] David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models, 2019.

[10] Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion, 2017.

[11] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms, 2016.

[12] Gary Marcus. Deep learning: A critical appraisal, 2018.

[13] Steven A. Sloman. The empirical case for two systems of reasoning. *Psychological Bulletin*, 119:3–22, 1996.

[14] Jonathan Evans. In two minds: Dual-process accounts of reasoning. *Trends in cognitive sciences*, 7:454–9, 11 2003.

[15] Daniel Kahneman. *Thinking, fast and slow*. Farrar, Straus and Giroux, 2011.

[16] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *arXiv preprint arXiv:1912.01412*, 2019.

[17] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems*, pages 8822–8833, 2018.

[18] Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. *arXiv preprint arXiv:1701.06972*, 2017.

[19] Yuhuai Wu, Albert Jiang, Roger Grosse, and Jimmy Ba. Neural theorem proving on inequality problems.

[20] Imanol Schlag, Paul Smolensky, Roland Fernandez, Nebojsa Jojic, Jürgen Schmidhuber, and Jianfeng Gao. Enhancing the transformer with explicit relational encoding for math problem solving. In *Thirty-third Conference on Neural Information Processing Systems (NeurIPS) 2019*, December 2019.

[21] Drew A. Hudson and Christopher D. Manning. Compositional attention networks for machine reasoning, 2018.

[22] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Learning to compose neural networks for question answering. *CoRR*, abs/1601.01705, 2016.

[23] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Deep compositional question answering with neural module networks. *CoRR*, abs/1511.02799, 2015.

[24] Jason Jo, Vikas Verma, and Yoshua Bengio. Modularity matters: Learning invariant relational reasoning tasks, 2018.

[25] Emilio Parisotto, Abdel rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis, 2016.

[26] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

[27] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples, 2018.

[28] Alberto Camacho and Sheila A. McIlraith. Towards neural-guided program synthesis for linear temporal logic specifications, 2019.

[29] François Chollet. On the measure of intelligence, 2019.

[30] top-quarks/arc-solution. `https://github.com/top-quarks/ARC-solution`. Accessed: 2020-10-05.

[31] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning, 2020.