Neural-Guided, Bidirectional Program Search for Abstraction and Reasoning

Simon Alford[1(✉)], Anshula Gandhi[1], Akshay Rangamani[1], Andrzej Banburski[1], Tony Wang[1], Sylee Dandekar[2], John Chin[1], Tomaso Poggio[1], and Peter Chin[3]

[1] Massachusetts Institute of Technology, Cambridge, MA 02139, USA
[2] Raytheon BBN Technologies, Cambridge, MA 02138, USA
[3] Boston University, Boston, MA 02215, USA

Abstract. One of the challenges facing artificial intelligence research today is designing systems capable of utilizing systematic reasoning to generalize to new tasks. The Abstraction and Reasoning Corpus (ARC) measures such a capability through a set of visual reasoning tasks. In this paper we report incremental progress on ARC and lay the foundations for two approaches to abstraction and reasoning not based in brute-force search. We first apply an existing program synthesis system called DreamCoder to create symbolic abstractions out of tasks solved so far, and show how it enables solving of progressively more challenging ARC tasks. Second, we design a reasoning algorithm motivated by the way humans approach ARC. Our algorithm constructs a search graph and reasons over this graph structure to discover task solutions. More specifically, we extend existing execution-guided program synthesis approaches with deductive reasoning based on function inverse semantics to enable a neural-guided bidirectional search algorithm. We demonstrate the effectiveness of the algorithm on three domains: ARC, 24-Game tasks, and a 'double-and-add' arithmetic puzzle.

Keywords: Abstraction · Reasoning · Program synthesis · Neural networks

1 Introduction

The growth and tremendous success of deep learning has catapulted us past many benchmarks of artificial intelligence. Reaching human and superhuman performance in object recognition, language generation and translation, and complex games such as Go and Starcraft has pushed the boundaries of what humans can do and machines cannot [7,12,13,16,19,22]. To continue to make progress, we must identify and work towards reducing the gaps between human and machine intelligence.

The Abstraction and Reasoning Corpus (ARC), introduced by François Chollet in 2019, captures an important aspect of human intelligence that our current systems are unable to do: the ability to systematically and flexibly generalize to

R. M. Benito et al. (Eds.): COMPLEX NETWORKS 2021, SCI 1072, pp. 657–668, 2022.
https://doi.org/10.1007/978-3-030-93409-5_54

new domains [6]. Chollet argues that intelligence must be measured not as skill in a particular task, but as *skill-acquisition efficiency*. General intelligent systems must also have *developer-aware generalization*, i.e. be able to solve problems the developer of the system has not encountered before or anticipated.

ARC consists of training, evaluation, and private test sets of 400, 400, and 200 tasks. Each task consists of 2–4 training examples and one or more test examples. Each training example is an input/output pair of grids. To solve a task, an agent must determine the relationship between input and output grids in the training examples, and use this to produce the correct output grid for each of the test examples, for which the agent is only given the input grid. Each task is thus a few-shot learning problem, for which the solution is symbolic and rule-based (Fig. 1).

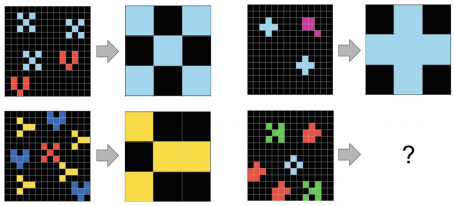


Fig. 1. An example ARC task with three training examples and one test example. The solution might be described as "find the most common object in the input grid".

The tasks are unique and constructed by hand so as to prevent the reverse engineering of any synthetic generation process. They are designed to depend on a set of human Core Knowledge inbuilt priors such as objectness, simple arithmetic abilities, symmetry, and goal-directedness. The evaluation and private test sets are designed such that a solution tailored to the training set is unlikely to transfer to the evaluation or test sets. Chollet hosted a Kaggle-competition for ARC and the winning solution, a hard-coded brute force approach, achieved only ∼20% performance on the private test set [2].

In this paper we report incremental progress on ARC and lay the foundation for several approaches to abstraction and reasoning not based in brute-force search. We approach ARC as a program synthesis benchmark, solving tasks by writing programs that convert input grids to output grids. In Sect. 2 we outline an approach to abstraction by applying DreamCoder [11]. We show this approach enables learning new concepts that aid in generalization as well as the solving of progressively more challenging tasks. In Sect. 3 we describe a novel program synthesis approach motivated by the way humans approach ARC that captures the reasoning required to search for ARC task solutions. Our algorithm constructs a search graph and reasons over this graph structure to discover task solutions. More specifically, we extend existing execution-guided program synthesis approaches [10,25] with deductive reasoning based on function inverse semantics [18] to enable a neural-guided bidirectional search algorithm.

We evaluate our approach on three domains: ARC tasks, '24 Game' problems, and a simple 'double-and-add' challenge. These experiments show the benefits of bidirectional search over baselines and the potential for further progress on ARC. In Sect. 4 we discuss related work, progress on ARC, and future directions.

2 Abstraction Using DreamCoder

We frame the problem as a search problem over the space of programs expressible in some domain specific language (DSL). One way a learning agent can achieve *developer aware generalization* (in the sense of [6]) is to identify frequently occurring patterns of computation and form abstractions from them. These abstractions enable searching for more complex programs more quickly.

In this section we use DreamCoder [11], a recent tool for program synthesis, to form abstractions. We first show how DreamCoder's compression algorithm enables learning generalizations of concepts seen in training. Second, we run DreamCoder on ARC to show how forming new abstractions enables the agent to solve progressively more challenging tasks.

2.1 Warmup: Forming Abstractions

To show how DreamCoder can form more abstract concepts from existing ones, we supply our agent with six synthetic tasks (meant to be similar to ARC tasks): drawing a line in three different directions, and moving an object in three different directions. See Fig. 2 for a visualization of these tasks.

We solve these tasks with four primitives: rotate clockwise and counterclockwise, draw a line down, and move an object down. The programs synthesized are the following:

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

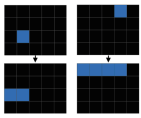
After running the compression algorithm, the agent creates the following new abstractions:

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

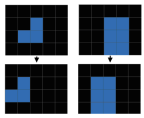
Importantly, the abstractions formed are more general than the original primitives given. This can help enable systematic generalization on further tasks.

2.2 Enabling Generalization on ARC Symmetry Tasks

In a second experiment, we demonstrate how compression-based learning enables developer-aware generalization on ARC. We provide DreamCoder with a set of five grid-manipulation operations: flipping vertically with `vertical_flip`, rotating clockwise with `rotate_cw`, overlaying two grids with `overlay`, stacking two



(a) An example "draw line left" task



(b) An example "move object left" task

Fig. 2. Sample tasks involving applying an action left.

grids vertically with `vertical_stack`, and getting the left half of a grid with `left_half`. We then train our agent on a subset of 36 ARC 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 abstractions. During the next iteration, the agent repeats its search equipped with the new abstractions. In this experiment, our agent initially solves 16 tasks. After one iteration, it solves 17 in the same amount of time. After another, it solves 19 tasks, and after the final iteration, it solves 22 tasks. Table 1 shows some of the new abstractions learned by DreamCoder's compression algorithm such as flipping horizontally, and stacking grids horizontally. The program solutions for the final tasks solved, shown in Fig. 3, could not be feasibly discovered without the use of abstractions to reduce the search time.

2.3 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, functions learned via compression, represented as programs, can naturally be composed and extended to solve harder tasks, while reusing concepts between tasks. This constitutes a learning paradigm which we view as essential to human-like reasoning.

Table 1. Useful actions learned in the process of solving symmetry tasks. Pound signs represent abstractions. Abstractions 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.

Action	Code
Mirror across diagonal	`#(lambda (rotate_cw (vertical_flip $0)))`
Rotate 180°	`#(lambda (rotate_cw (rotate_cw $0)))`
Flip horizontally	`#(lambda (rotate_cw (rotate_cw (vflip $0))))`
Rotate counterclockwise	`#(lambda (rotate_cw (#(lambda (rotate_cw (rotate_cw $0))) $0)))`
Stack grids horizontally	`#(lambda (lambda (#(lambda (rotate_cw (vertical_flip $0))) (stack_vertically (#(lambda (rotate_cw (#(lambda (vertical_flip $0)) $0))) $1) (#(lambda (rotate_cw (vertical_flip $0))) $0)))))`

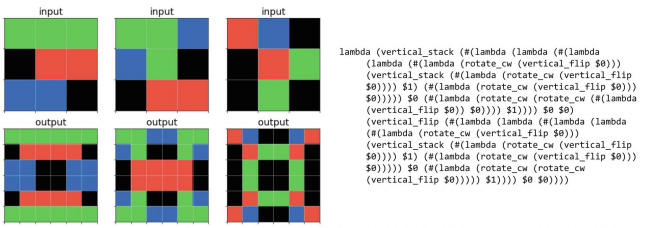


Fig. 3. One of the four-way mirroring tasks and the program discovered that solves it written in terms of the original primitives. The program was discovered only after four iterations of enumeration and compression.

There is a caveat of the approach shown here. Abstraction as shown uses a simple enumerative search. DreamCoder uses a form of neural-guided program synthesis, predicting a distribution over functions to search over, but this guidance is too weak to scale to the complexity of ARC tasks. In the next section, we show the type of reasoning required for ARC and design an approach to exhibit this reasoning.

3 Bidirectional, Neural-Guided Program Search

In Subsect. 3.1 we first motivate and describe our bidirectional, neural-guided search algorithm. Then in Subsect. 3.2 we present experiments and results using this approach.

3.1 Algorithm Description

In this section we describe our reasoning approach for ARC. We first give a motivating example of human reasoning on ARC, explain how to approximate it with execution-guided synthesis, then incorporate inverse semantics to create a bidirectional, neural-guided search algorithm.

Motivating Example. Solving ARC tasks fundamentally consists of a search for valid solutions. To make this search tractable, our agent needs the ability to reason towards solutions. ARC tasks feature rich visual queues that guide us towards solutions. Without enabling our agent to take full advantage of these queues, the search over possible programs becomes impossibly large. The process of discovering the solution to an ARC task often consists of several discrete steps of reasoning before discovering the solution. How can we design an approach to search that searches for ARC solutions in the same manner as humans?

As a motivating example, let us consider solving task 303 in Fig. 4. The reasoning steps to come to a solution might look something like this:

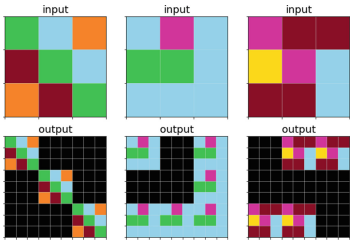


Fig. 4. Task 303.

1. **Notice** that the output grid consists of copies of the 3×3 input grid, arranged in a certain arrangement among a 9×9 grid.
2. **New question:** Where should we place the input grid copies?
3. **Notice** that the placements match the arrangement of a different color's pixels for each grid. For example, in the first example, the diagonal of grids in the output matches the green pixels in the input.
4. **New question:** What color should we arrange our grid copies along?
5. **Solution:** The color matched is the most common color in the grid.

Notice the way discovering a solution involves combining sequential insights and problem reductions. Systematizing a form of reasoning for ARC that emulates this reasoning will be based on a combination of execution-guided program synthesis and inverse semantics.

Extending Execution-Guided Synthesis. Execution-guided program synthesis [5,10] is a form of program synthesis where one executes partial programs to produce intermediate outputs, which are used to guide the construction of the full program. Intermediate evaluations provide the opportunity for step-by-step reasoning: instead of coming to the answer at once, one can construct it piece by piece. Humans could be said to make use of the same thing: for instance, it much easier to write out the result of a multiplication digit by digit, instead of conducting the full calculation in one's head. The form of execution-guided synthesis we apply to ARC is most similar to the 'REPL' approach of [10]. An example applying the technique to ARC is shown in Fig. 5.

Existing execution-based synthesis approaches are limited to bottom-up enumeration: the leaves of the program are constructed (and evaluated) first. In contrast, the steps for solving task 303 involve proposing a function that is used to produce the output grid, and deducing the inputs required to correctly produce the output as new intermediate targets before discovering the complete program.

This form of deductive reasoning involves evaluating function in reverse. It is best exemplified in the FlashMeta system [18], which leverages the inverse semantics of operators to deduce one or more inputs of a function given the output target and one or more inputs. We incorporate this type of reasoning into an extension of execution-guided program synthesis.

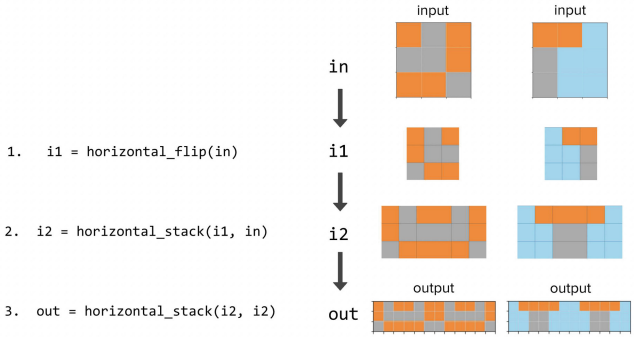


Fig. 5. Solving ARC task 138 from the evaluation set with execution-guided synthesis. Conditioned on the input and output grids, the agent chooses to flip the input horizontally in step one. This action is executed to produce intermediate value `i1`. Next, the agent chooses to horizontally stack the intermediate value with the input grid, producing another value `i2`. Last, the agent horizontally stacks this value `i2` with itself, correctly producing the output grid for each example and solving the task.

Deductive Reasoning via Inverse Semantics. For our purposes, we can consider two cases. The simplest case is when the function is invertible. In this case, we can evaluate the inverse to produce two new targets for the search, as shown in Fig. 6. In the second case, the function is *conditionally* invertible: given the output and one or more inputs to a function, one can deduce the remaining inputs needed to produce the output via this function. Many functions are conditionally invertible; perhaps the most familiar family is arithmetic operators: if we know $1 + x = 5$, we can deduce that $x = 4$. An example relevant to ARC is shown in Fig. 6. Using conditional inverses, it is possible to formalize the reasoning described for task 303.

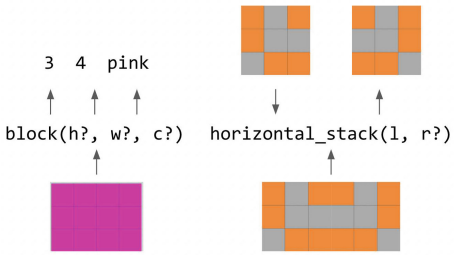


Fig. 6. *Left:* the function `block` is directly invertible: given the output, we can deduce the inputs. *Right:* the function `horizontal_stack` (horizontal stack) is conditionally invertible: given the output and one input, we can deduce the other input.

Bidirectional, Neural-Guided Program Search. To extend execution-guided synthesis to a bidirectional algorithm with inverse and conditional inverse functions, we extend the environment of [10], approaching the synthesis task via reinforcement learning.

The setup takes place in a Markov Decision Process. The current state is a graph of nodes. Each node represents either an input value, the output value, or an intermediate value resulting from the execution of an operation in the forwards or backwards direction. A node is *grounded* if there is a valid program to create that node from the operations applied so far. In general, grounded nodes correspond to those from the forwards, i.e. bottom-up program enumeration, direction of search, while ungrounded nodes correspond to those from the backwards direction, i.e. top-down program enumeration.

An *operation* is a function from the grammar along with a designation of being applied in forwards, inverse, or as a conditional inverse (and if as a conditional inverse, conditioned on which input arguments). There are three types of operations: forward operations, inverse operations, and conditional inverse operations. A forward operation applies the function to a set of grounded inputs to produce a new grounded node. An invertible operation takes an ungrounded output and produces a new ungrounded target node such that grounding the target node will cause the output node to be grounded as well. A conditionally invertible operation takes an ungrounded output and one or more grounded input nodes, and produces a new ungrounded target node such that grounding the target node will cause the output node to be grounded as well. All invertible and conditionally invertible operations have a corresponding forward operation.

Solving a given task thus consists of an episode in the MDP. Actions in the MDP correspond to a choice of operation and the choice of arguments for that operation. Each action applies a function in either the forward or backward direction. Intuitively, this executes a bidirectional search to try to connect the grounded nodes on one side with the ungrounded output node on the other. We give reward R for solving the task and a penalty of -1 for choosing an action corresponding to an invalid operation.

Like [5,10,25], we train with a combination of supervised training on randomly generated programs fine-tuning with reinforcement learning algorithm REINFORCE. To generate random bidirectional programs for supervised training, we first create a random program, and construct an execution trace for it by probabilistically converting inverting function applications from the root. Network architecture is held the same from [10], with task-dependent embedding network nodes of the bidirectional graph, a DeepSet network [24] to encode the graph into a single embedding and choose a function to apply, and a pointer network [23] for choosing function arguments.

3.2 Experiments

We evaluate our bidirectional algorithm in three settings: solving ARC symmetry tasks, solving arithmetic puzzles from the '24 Game' family, and solving 'double-and-add' puzzles. As a baseline, we compare bidirectional synthesis with

a forward-only baseline which only allows application of operations in the forwards direction like existing approaches.

ARC Symmetry Tasks. As a proof of concept, we evaluate the bidirectional algorithm on a set of 18 ARC symmetry tasks—a subset of those used in Sect. 2. We use a DSL of six operations: stacking two grids horizontally or vertically, rotating clockwise or counterclockwise, and flipping a grid horizontally or vertically. The rotation and flip functions are directly invertible, while the stacking operations are conditionally invertible. We use a convolutional neural network to embed grid example sets. We train on a set of randomly generated programs evaluated on random input grids from the ARC training set, and fine-tune with REINFORCE before sampling rollouts for thirty minutes on all tasks at once. The agent is able to solve 14 of 18 tasks, including one of the "four-way mirror" tasks. In this experiment, bidirectional performed equally to the forward-only baseline.

24 Game. Next, we compare the performance of bidirectional search with the forward-only baseline by tasking our agent with solving "24 Game" problems. A 24 Game consists of four input numbers, one through nine. To solve the task, one must use each number once in an expression that creates twenty four using $+, -, \times, \div$. For example, given 8, 1, 3, and 2, a solution is $24 = (2 - 1) \times 3 \times 8$. To solve these tasks bidirectionally, we can use the conditional inverse of each arithmetic operator in addition to forward arithmetic operations.[1]

First we conduct supervised pretraining on all depths at once. These programs may create any number as a target, not just 24, with the maximum allowed integer 100, and no negative or nonintegral numbers. We then fine-tune on different depths with REINFORCE for 10,000 epochs of batch size 1000. We measure performance by percent of episodes solved in the last 1,000 epochs of training.

Results are shown in Table 2. Bidirectional synthesis outperforms the forward-only baseline across all depths. This supports our thesis, but is suspicious: as we should expect to see identical accuracy for depth one tasks, when only a single action is needed. Accuracy remains fairly high as depth increases, because depth does not necessarily imply program length: as many as 40% of depth four tasks remain solvable in fewer than four actions.

Double-and-add. Last, we include results on a 'double-and-add' task to better show the advantage of bidirectional search. Given a target number, one must reach it starting from the number two by repeatedly adding one or doubling the number. For example, $7 = 1 + 2 * (1 + 2)$. This task, akin to the method for exponentiation by repeated squaring, is much easier solved in a top-down fashion: the choice of adding one or doubling boils down to whether the target is even or odd. Here we have two forward operations, each of which are directly invertible. On a training set of five thousand numbers sampled between one and five million, and a held out set of five hundred numbers, our bidirectional model

[1] We relax the rule that each input is used exactly once.

Table 2. Percent of tasks solved for 24 Game, measured by percent of episodes solved in the last 1000 epochs of RL fine-tuning. Forward-only denotes only using forward operations. Bidirectional includes conditional-inverse operations. Average over three runs with stdev shown.

Depth	1	2	3	4
Forward-only	87.22 ± 0.64	84.29 ± 1.6	75.88 ± 3.6	67.04 ± 1.0
Bidirectional	95.2 ± 0.66	92.9 ± 2.1	87.7 ± 1.1	85.3 ± 1.9

achieves 100% evaluation accuracy after a single epoch of supervised training. In contrast, the forward-only model fails to solve the held-out tasks, due to the difficulty "seeing" the solution from the source, see Fig. 7.

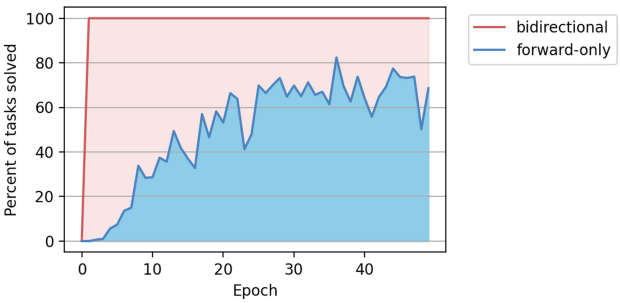


Fig. 7. Percent of tasks solved for bidirectional and forward-only agents trained on the double-and-add task. The bidirectional agent achieves 100% accuracy after a single epoch of training. After fifty epochs of training, forward-only converges without solving the held-out tasks.

4 Discussion

Related Work. Our work builds off and is inspired by a long line of progress in neural program synthesis [3,4,8,17,21], execution-guided synthesis [5,10,26], and deep reinforcement learning for search [15,19].

Bidirectional, neural-guided program search is made possible primarily due to the inverse semantics of FlashMeta [18]. The concept of bidirectional programming, inverse semantics, and program inversion has been present throughout the history of program synthesis [9,14,20], but the way in which inverse evaluation is used here is most similar to FlashMeta.

ARC. To date, there are no prominent learning-based approaches to ARC that have proven more successful than the Kaggle-winning brute-force approach [2]. Other Kaggle approaches include genetic programming and cellular automata, but all essentially rely on brute force search over a DSL of operations combined

with ARC-specific tricks, without any substantial learning [1]. The few-shot nature and large search space for ARC make it a very challenging benchmark, and progress scaling program synthesis algorithms is likely needed to enable further progress. We hope our progress reported here inspires and enables further progress on ARC.

Future Work. The next step of our work is to combine the two approaches to create a unified approach. This can be done by using the bidirectional search algorithm to solve tasks, then create new operations out of abstractions base on tasks solved. To fill out the learning approach, we can consider including the ability to synthesize inverse and conditional inverse operations for newly learned abstractions, perhaps as its own synthesis problem. Our approach remains to be scaled up to a full DSL capable of solving ARC. Incorporating more sophisticated inverse semantics and type-directed search are important components of the full bidirectional approach.

References

1. Abstraction and reasoning challenge—kaggle (2020). https://www.kaggle.com/c/abstraction-and-reasoning-challenge/leaderboard
2. top-quarks/arc-solution (2020). https://github.com/top-quarks/ARC-solution. Accessed 05 Oct 2020
3. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: DeepCoder: learning to write programs (2016)
4. Cai, J., Shin, R., Song, D.: Making neural programming architectures generalize via recursion (2017)
5. Chen, X., Liu, C., Song, D.: Execution-guided neural program synthesis. In: International Conference on Learning Representations (2018)
6. Chollet, F.: On the measure of intelligence (2019)
7. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding (2019)
8. Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.R., Kohli, P.: RobustFill: neural program learning under noisy I/O (2017)
9. Dijkstra, E.W.: Program Inversion. In: Dijkstra, E.W. (ed.) Selected Writings on Computing: A personal Perspective. Texts and Monographs in Computer Science, pp. 351–354. Springer, New York (1982). https://doi.org/10.1007/978-1-4612-5695-3_63
10. Ellis, K., Nye, M., Pu, Y., Sosa, F., Tenenbaum, J., Solar-Lezama, A.: Write, execute, assess: program synthesis with a REPL (2019)
11. Ellis, K., et al..: DreamCoder: growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning (2020)
12. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition (2015)
13. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
14. Lubin, J., Collins, N., Omar, C., Chugh, R.: Program sketching with live bidirectional evaluation. In: Proceedings of the ACM on Programming Languages, vol. 4, no. ICFP, pp. 1–29 (2020). https://doi.org/10.1145/3408991

15. McAleer, S., Agostinelli, F., Shmakov, A., Baldi, P.: Solving the Rubik's cube with approximate policy iteration. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6–9, 2019. OpenReview.net (2019). https://openreview.net/forum?id=Hyfn2jCcKm
16. Mnih, V., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
17. Nye, M., Hewitt, L., Tenenbaum, J., Solar-Lezama, A.: Learning to infer program sketches (2019)
18. Polozov, O., Gulwani, S.: FlashMeta: a framework for inductive program synthesis. In: Aldrich, J., Eugster, P. (eds.) OOPSLA, pp. 107–126. ACM (2015). http://dblp.uni-trier.de/db/conf/oopsla/oopsla2015.html#PolozovG15
19. Silver, D., et al.: Mastering the game of go without human knowledge. Nature **550**(7676), 354–359 (2017)
20. Srivastava, S., Gulwani, S., Chaudhuri, S., Foster, J.S.: Path-based inductive synthesis for program inversion. SIGPLAN Not. **46**(6), 492–503 (2011). https://doi.org/10.1145/1993316.1993557
21. Valkov, L., Chaudhari, D., Srivastava, A., Sutton, C., Chaudhuri, S.: HOUDINI: lifelong learning as program synthesis (2018)
22. Vinyals, O., et al.: Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature **575**(7782), 350–354 (2019)
23. Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks (2017)
24. Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R., Smola, A.: Deep sets (2018)
25. Zhou, C., Li, C.L., Poczos, B.: Unsupervised program synthesis for images using tree-structured LSTM (2020)
26. Zohar, A., Wolf, L.: Automatic program synthesis of long programs with a learned garbage collector (2019)