Lab42

Essay Challenge

ARC Solution Concept

Outline of a Detailed Approach to Solving the Abstraction and Reasoning Corpus

The Hitchhiker's Guide to the ARC Challenge

Simon Ouellette

27. May 2023

Lab42, Mindfire Foundation | Switzerland | www.lab42.global

1 Introduction

To solve the Abstraction & Reasoning Corpus (ARC), the learner must be able to "reverse engineer", in a sense, the underlying process (or algorithm) that generated the examples. This is an impossible task for most machine learning approaches, which are typically geared towards multidimensional curve fitting, rather than algorithm learning.

This is a key distinction, and it is a commonly held belief that deep learning or end-to-end differentiable methods in general can only "fit a curve", rather than learn the underlying process or algorithm.

This is undeniably true for non-recurrent learners, such as feed-forward neural networks: their architecture makes it impossible for them to express the necessary recursive or iterative processing mechanisms required to learn an algorithm. Instead, they can only, at best, learn a direct mapping from the domain space to the range space. This is nothing more than a form of multidimensional curve fitting, and the learned model is thus, by construction, limited to the domain of training data. No ability to extrapolate can ever arise from such an approach.

However, for certain types of machine learning algorithms, these claims are called into question by recent experimental results [23, 8, 9, 11, 13, 25, 22, 39, 21, 37, 14, 40, 38, 29, 34, 17, 19].

Humans are able to solve novel, complex reasoning problems by the appropriate composition of, and iteration over, simple logical primitives. As long as the same "first principles", or fundamental laws, still hold on the test environment, it is possible to extrapolate successfully in this manner. In this essay, the very limited set of algorithms that offer such a capability will be discussed, and a plausible methodology will be proposed to tackle the Abstraction & Reasoning Corpus.

2 Literature Review

2.1 Deep Learning-based Algorithm Learning

2.1.1 The Neural GPU

In 2015, Kaiser & Sutskever[8] proposed the Neural GPU: a neural network architecture which, they claim, is able to learn algorithms. It essentially consists of a recurrent neural network where the central component is a Convolutional Gated Recurrent Unit (rather than a LSTM unit, for example). Unlike Long Short-Term Memory (LSTM) networks, the Neural GPU does not process a new input sequence element at each iteration, instead it iterates over (convolutional) operations on the whole input.

They performed 6 experiments: long binary addition, long binary multiplication, copying sequences, reversing sequences, duplicating sequences and counting by sorting bits. For each of these experiments, they trained on sequences of up to a predetermined length, and evaluated on much longer sequences than

those seen during training, in order to demonstrate the capacity for algorithmic extrapolation.

For binary addition and multiplication, they reported an accuracy of 100% on sequences of up to 2000 bits, using models they trained on only on 20 bits. In contrast, Stack-RNNs[4] and attentional LSTMs failed to generalize. On the 4 other algorithmic tasks, they trained their model using sequences of up to 41, and tested on sequences of up to 4001. They also reported perfect generalization on these experiments.

They visualized the operations that occur inside the Neural GPU and presented the output as a video. From this visualization it is obvious, for example, that for the duplication task it correctly learned to move a part of the embedding downwards in each step. That is, it learned the solution process itself, not a memorized mapping or a shortcut solution.

However, they noted that these results are taken from only the few best models out of 729 models trained via grid search. In other words, training a Neural GPU that generalizes well seems to be very difficult, and highly dependent on hyperparameter choices.

Two subsequent papers, however, addressed this unreliability of the Neural GPU. In 2016, Price et al.[9] experimented with Neural GPUs, showing that with bigger model size and a carefully designed curriculum, they obtained more reliably accurate models. In 2018, Freivalds & Liepins[11] applied some modifications to the original Neural GPU architecture. All their trained models generalized to 100 times longer inputs with less than 1% error, which represents a significant improvement in training stability.

2.1.2 Universal Transformers

The canonical (or "vanilla") transformer consists of a stack of 6 encoder blocks and 6 decoder blocks. This means that, by construction, it is unable to perform more than 6 sequential attention operations while encoding (and the same goes for decoding). One attention operation here refers to the parallel process of attending to each token in the input sequence. This means, in other words, that **the total number of operations a canonical Transformer can perform scales linearly with the input sequence length, and the total number of sequential operations is constant**.

As a result, the Transformer cannot be said, in practice, to be Turing complete (or computationally universal). That is, it cannot learn to solve problems that require any arbitrary, dynamic number of iterations[13].

These limitations of the original Transformer architecture are the motivation for the development of the Universal Transformer[13] (UT). This algorithm consists of a Transformer architecture augmented with an arbitrary recurrence mechanism over the encoding and decoding blocks. This means that the UT can loop indefinitely over a certain set of operations, and decide when the halting condition has been met. It should be noted that, unlike Recurrent Neural Networks (RNN), the UT iterates over encoding block (or decoding) operations (i.e. through processing "depth"), rather than over time steps in the input se-

quence. Hence, the number of operations that can be learned by the UT scales super-linearly with respect to the input sequence length.

The authors of the UT architecture validated their approach on bAbI question answering, subject-verb agreement, LAMBADA language modeling, learning to execute (LTE), machine translation and algorithmic tasks similar to the ones used in the Neural GPU. Not all of these are relevant to the question of algorithmic learning that is central to this essay, so the focus will be on the LTE and algorithmic tasks.

LTE tasks[4] consist of mapping the character-level representation of a program to their correct output. In other words, the algorithm must learn to execute the instructions (or "code") presented to it. This program evaluation mode consists of 3 different sub-tasks: "Program", "Control", "Addition". In addition to "program evaluation", they investigated the task of memorizing an input sequence: the learner receives a number sequence such as "1212123", and after going through each of the input elements, it must output the identical sequence ("Copy"). Two other variants of the problem are presented as well: one where the input sequence is presented twice in a row ("Double"), and one where it is presented in reverse order ("Reverse").

In both the memorization and program evaluation variants of the LTE task, the Universal Transformer reached an accuracy of 100%, with the exception of the "Program" sub-task, where it obtained a sequence accuracy of 63%. In comparison, the vanilla transformer reached 63% ("Copy"), 55%("Double"), and 26%("Reverse") sequence accuracies on the memorization variants, and 29%("Program"), 66%("Control") and 100%("Addition") on the program evaluation variants. While these are higher accuracies than with the LSTM, these results provide empirical support to the notion that vanilla Transformers are not Turing-complete in practice (due to the static number of possible iterations).

The algorithmic tasks consist of learning to copy, reverse or add strings of decimals symbols (0 to 9). They trained the models on sequences of length 40, and evaluated them on sequences of length 400.

On the "Copy" algorithmic task, the UT reached a sequence of accuracy of 35%, compared to 3% for the vanilla transformer and 9% for the LSTM. On the "Reverse" task, the UT performed at 46%, while the vanilla transformer only had 6% and the LSTM 11%. Finally, on the "Addition" task, the UT only obtained 2% accuracy, while the vanilla transformer had 0% accuracy like the LSTM.

This further supports the superiority of UTs to vanilla transformers for algorithm learning, although it also brings up the question: why did the UT not obtain a 100% accuracy? It is further worth noting that the Neural GPU obtained 100% sequence accuracy on all of those algorithmic tasks, although the authors noted that curriculum learning was necessary to reach that level of performance. Curriculum learning was not used to generate the Universal Transformer results.

In 2022, researchers[25] added an external grid-like memory to the UT, and showed that it was able to learn multi-operand, multi-digit addition. The model was trained on sequences of up to 10 digits and up to 4 operands. The results

indicated near-perfect (above 99%) accuracy on sequences of up to 2000 digits for 2-terms, which is a longer sequence than with the Neural GPU.

2.1.3 Deep Thinking Systems

In 2021, researchers examined whether recurrent neural networks trained on easy problems can extrapolate the concepts they learned to harder instances of these problems[22]. They used three different classes of problems for their experimentation: computing prefix sums, solving mazes, and solving chess puzzles. For each of these, they trained recurrent neural networks, specifically a recurrence over ResNets[6], on an "easy" subset, and they evaluated it on a much harder set of problems that cannot possibly have been "memorized" (or "curve-fitted") during training. The goal was to demonstrate that the model has learned iterative application of fundamental principles, i.e. an algorithm, rather than a direct mapping from training domain to training range.

In the prefix sums problem, the training samples are binary strings. The goal is to output a binary string of equal length in which each bit represents the cumulative sum of input bits modulo 2. For example, for the input: [1, 0, 1, 0, 1], we would expect to see: [1, 1, 0, 0, 1].

On the prefix sums problems, they reported that the recurrent ResNet architecture trained on 32-bit strings can generalize with accuracy greater than 90% to strings of up to 44 bits. On the maze task, they trained on "small" (9x9 grids) mazes and reported performance slightly upward of 70% on "large" (13x13) mazes. On the chess puzzles, they trained on problem instances with an Elo rating below 1385, and tested on problem instances with an Elo rating greater than 1385. The results indicated a success rate on the hard puzzles of around 74.7%.

Overall, they concluded that their proposed solution does indeed learn algorithmic extrapolation, but only in a modest, approximate sense that degenerates as the complexity of the test problems scale up.

These authors, in later research[23], discovered a fundamental reason for the inability of their recurrent neural networks to scale up beyond a certain level of algorithmic complexity, which they call "overthinking". In particular, they discovered that as the algorithm iterates over a hidden state in an attempt to solve a problem, the gradual intermediate transformations tend to lose track of the original problem (information gets lost).

When humans think deeply about a problem, we often stop to re-read the problem statement or to re-evaluate the task in question. Similarly, in order to solve the "overthinking" problem, they proposed a "recall architecture", which essentially re-concatenates at every processing step the original input (i.e. "skip connections").

It is a simple strategy that yields remarkable improvements. On the task of computing prefix sums, they trained a recurrent ResNet (as in their previous work) on 32-bit inputs and extrapolated to 512-bit data. Without the recall architecture, the accuracy is 0%. With the recall architecture, the accuracy improves to 97%. On the task of solving mazes, they trained on 9x9 grids

and extrapolated to 59x59 grids with a 97% accuracy (note: they must also be trained with the special "progressive loss" that was also introduced in that paper, otherwise the accuracy drops to 83%), while the baseline "Deep Thinking" architecture failed completely at a 0% accuracy. The gains on the chess puzzles were, however, more modest at 4%, possibly indicating that the main difficulty in this class of problems is of a different nature than algorithmic extrapolation.

2.1.4 Language Models and reasoning

Large language models (LLM) have demonstrated an emergent ability to perform some forms of verbal reasoning, given the right conditions. In-context learning describes the use case where LLMs are provided with an input prompt that contains a few input-output examples of a task, and it is expected to learn how to solve the task for future examples only from these few input examples. This is a form of "few-shot learning".

For example, one might provide the following prompt:

> Input: [0, 0, 0, 3], Output: [0, 0, 3, 0]
> Input: [0, 2, 0, 0], Output: [2, 0, 0, 0]
> Input: [0, 0, 9, 0], Output: [0, 9, 0, 0]
> Input: [0, 5, 0, 0], What is the output?

And it would be expected to learn the task solution (shifting the sequence to the left by 1 position) from the 3 examples and to give the answer: [5, 0, 0, 0].

Typically, in-context learning refers to an emergent ability to do this in spite of the fact that they have not been trained with this task structure. Instead, LLMs are trained in a self-supervised manner to simply predict the next words in sentences of various documents. It is therefore surprising that this capability emerges from an entirely different training approach.

However, research has shown that also using such an in-context learning setup at training time[26, 24, 33, 27] helps with the downstream few-shot learning capabilities. It is, therefore, also possible to "meta-train" a Transformer to directly use this few-shot input-output example structure.

Chain-of-thought prompting is a technique that consists of describing intermediate steps within these in-context input-output examples. This encourages the model to learn the reasoning process itself and to explain its own reasoning when making the inferences.

On the ScienceQA benchmark[30], which consists of multi-modal multiple choice questions on science topics that require an explanation for each answer, "Multimodal Chain-of-thought" approaches[42] led to super-human performance at 91.68% (also outperforming GPT-3.5 by 16.51%).

An example of a ScienceQA question:

> "**Question**: Which type of force from the baby's hand opens the cabinet door?
> **Options**: A) pull B) push

> **Context**: A baby wants to know what is inside of a cabinet. Her hand applies a force to the door, and the door opens. (An image is provided of a baby in the process of opening a cabinet door)
> **Answer**: The answer is A.
> **Because**: The baby's hand applies a force to the cabinet door. This force causes the door to open. The direction of this force is toward the baby's hand. This force is a pull."

The question of whether LLMs rely only on superficial correlations or use a deeper kind of reasoning was analyzed in a recent paper[28]. The authors trained GPT on a trace of Othello moves, with no visual or spatial information about the state of the board itself. Using probing, an established technique in NLP, they explored the internal representation learned by the neural network.

They showed that for each step in a game, if they probed the internal state of the model, they were able to predict the ground truth state of the board (i.e. for each square on the board, whether it contains white, black or blank) with an error rate of 1.7%. In comparison, a randomly initialized model gave an error rate of 26.2%. From this they conclude that, from the list of moves alone, the model maintains an internal representation of the state of the board as the game evolves. As a result, it is argued, predicting the next move is more than just a matter of learning surface correlations and probabilities.

2.1.5 Issues with deep learning

Gülçehre & Bengio [5] showed that gradient descent struggles on problems that are a composition of two or more highly non-linear tasks. They likened the search of a solution, in these cases, to searching for a needle in a haystack. This is a problem for ARC tasks, since they can be themselves seen as complex and abstract tasks composed of simpler tasks.

In their experiment, they generated 3 Pentomino shapes per example. These shapes were randomly positioned in the image, and they were randomly scaled and rotated. The goal was to return 1 if the three shapes in the image are the same, 0 otherwise. Conceptually, this task consists of first identifying the shapes themselves, and then reasoning logically about whether they are same or different. They were unable to find a neural network architecture that solves this task "as is" (i.e. without the help of intermediate hints). They pointed out that if either of the two sub-tasks is considered separately, they can easily be learned. The problem only truly arises with their composition.

In 2017, researchers[12] analyzed two examples of problems on which deep learning struggles. The first experiment, known as the parity learning problem, works as follows: binary vectors are generated with random numbers of 1s, and the classifier must learn to distinguish between odd and even numbers of 1s. However, the task considers only a specific subset of the bits, not the whole vector, and this subset is unknown at training time (it must be learned). The learner must therefore figure out that the 1s must be counted, the notion of parity must be learned, and the subset of bits to consider must be learned. As

the dimensionality of these vectors increases, the problem becomes increasingly difficult to learn: until a dimensionality of around 30, where the classifier was no longer able to perform beyond random chance.

In another experiment, they generated images of straight lines at random positions, lengths and angles. The intermediate task was to determine whether the line slopes "upwards" or "downwards". The end goal of this task was, like the previous one, a parity problem. Each sample was a k-tuple of these lines, and based on their "upwardness" or "downwardness", the solver had to determine whether there was an odd or even number of lines of a given slope orientation. This is also unlearnable "end-to-end" (for k greater than 3), according to their mathematical proof: the gradients are too noisy. More formally, they show mathematically that the signal present in the gradients for this task decreases exponentially as k increases.

They conclude that end-to-end attempts at solving these tasks cannot succeed, but if we decompose them into sub-tasks, and first learn these separately, then it becomes possible to solve the composite problem.

In more recent research[40], it was demonstrated that these learning difficulties in solving "multi-hop" reasoning problems persist even in LLMs and Transformers. For example, in the StrategyQA benchmark, which requires implicit decomposition into reasoning steps, one of the largest available language models (Gopher) achieves an accuracy of 61%, whereas human performance is around 87%.

The authors[40] **proved mathematically that with intermediate subtask supervision, a sequence-to-sequence model can learn any task that follows the structure of an efficient decomposition into simpler sub-tasks**. In other words, they proved a theorem guaranteeing that, when intermediate supervision is available, efficient neural network learning is possible.

In a similar line of reasoning, other researchers[38] **demonstrated that curriculum learning is an efficient way to provide this intermediate supervision**. They showed their results on the parity learning problem.

2.2 DSL-based Algorithm Learning

2.2.1 Brief review of the state of the art

Inductive program synthesis using Domain Specific Languages (DSLs) involves providing a few examples of the task's inputs and outputs to an algorithm. The algorithm then generates a program to solve this task, in the form of a tree of tokens selected from a grammar or DSL. These tokens represent function primitives that operate on the provided task inputs in order to generate the expected output.

In 2017, the algorithm RobustFill was proposed[10] for neural program synthesis. It consists of an attentional LSTM-based sequence-to-sequence model that outputs a sequence of tokens from a DSL. This sequence forms a program, which is then executed in order to solve the test examples from the task.

They used it to solve tasks from the FlashFill dataset, which consists of

learning to automatically fill in string examples, as in a spreadsheet. This is inspired by the Microsoft Excel functionality that allows users to fill in new rows or columns in a spreadsheet by first providing it with a few examples of the transformations to make. It is effectively a way of generating macros from examples, rather than from programming.

They achieved 92% accuracy on the FlashFill dataset, which is the same accuracy as the corresponding Microsoft Excel functionality at the time of writing. Moreover, when they injected noise in the dataset, they found that their approach remained highly accurate, while the Microsoft Excel version quickly approached a performance of 0% as the number of noisy characters was increased.

The state of the art in DSL-based inductive program synthesis is currently DreamCoder[20]. It is a hybrid approach between neural networks and discrete enumerative search over trees of programs (represented by elements of a DSL). The neural network's purpose is to output the probabilities of using each token from a DSL in the program to generate, while the discrete search component enumerates in non-increasing order of probability the possible programs.

What makes DreamCoder particularly interesting compared to its predecessors is its ability to learn new function primitives and add them to the DSL, which effectively grows over time. This capability is made possible through an advanced automatic refactoring algorithm that makes sure that semantically equivalent code segments across various tasks are correctly identified as "shared" and re-usable. Without this refactoring functionality, slightly different wordings of the same functionality would prevent the identification of a reusable subroutine.

The authors[20] showed results from a variety of tasks, including:

- *List processing*: consists of examples of number sequences as inputs and outputs, where different transformations are applied such as sorting the list, removing duplicates, summing up the numbers, etc.
- *Text editing*: similar to list processing, but it is character strings as inputs and outputs, and operations such as generating the initials (first letter of each word) or dropping the last 3 characters.
- *LOGO graphics*: a graphic is presented, and the LOGO Turtle instructions required to reproduce the graphic must be provided. So the generated program is essentially a list of instructions such as "forward 50, left 60, forward 100, etc.".
- *Blocks towers*: similar to LOGO graphics, a block tower pattern is presented as input, and the generated program must contain the correct block placement instructions to reproduce the input.
- *Symbolic regression*: the algorithm is presented with a set of input/output examples from a mathematical function, such as a law of physics.

They compared their results to RobustFill, as well as an earlier version of DreamCoder, called EC2. On text editing, DreamCoder achieved an accuracy around 70% (RobustFill: 0%). On LOGO graphics, DreamCoder achieved around 90% (RobustFill: 0%). On list processing, it reached around 90% (RobustFill: 15%). On symbolic regression, it attained approximately 85% (RobustFill: 70%). On tower building (Blocks towers) it reached a performance around 95% (RobustFill: 10%).

A Master's student from MIT attempted to use DreamCoder to solve the ARC challenge for his thesis[18]. However, his experimental results were inconclusive, with only a few ARC tasks being successfully solved. They concluded that DreamCoder's search algorithm is too weak to scale to challenging ARC tasks.

They further stated, in their conclusion:

> "Another direction involves addressing the vast amounts of prior knowledge humans bring to a challenge like ARC. Large generative pretrained models offer a potential way to incorporate prior knowledge in this manner. It remains to be seen the extent to which such models can generalize to the idiosyncratic ARC environment of discrete-colored grids, but the impressive improvements shown over the past few years indicate that betting against such models is not safe."

The state of the art on the ARC challenge is a DSL solution that implements a variety of grid transformation primitives. A discrete search over the appropriate DSL components is made, until the best fitting program is generated. This is then used to solve the given task. The solution does not contain any learning, all building blocks are hard-coded, and a heuristic search is used to solve the tasks.

2.2.2 Combinatorial explosion

It is well known that DSL-based inductive program synthesis suffers from a combinatorial explosion problem[20, 41]. This term refers to the fact that the search space of candidate programs increases exponentially as we add new elements to the grammar. As a result, as we try to increase the flexibility and expressivity of the DSL, so does the complexity of the search increase. It also refers to the fact that search space increases exponentially with the depth (complexity) of the target solution. This severely limits the usefulness of this DSL-based search methodology for non-trivial problems.

DreamCoder attempts to tame this combinatorial explosion in at least two ways. First, it does this by learning an increasingly growing DSL from reusable subroutines that it discovers. Indeed, the fact that the DSL becomes increasingly specific to the problem domain as it trains on it means that the target solution for each task can be expressed by a smaller amount of primitives. This accelerates the search for a solution. Second, there is a neural network module

that assigns probabilities to the DSL elements, which means that the search can focus on a narrower subset.

Some researchers[41] proposed a divide-align-conquer strategy to address this issue. They first decomposed a visual scene (an input grid in ARC) into its constituents by performing object segmentation. They then "aligned" corresponding objects between input and output grid, and they solved each of these transformations individually. The idea is that the complexity of the larger task can be mitigated by dividing it into simpler sub-tasks. However, due to the problem-specific, hard-coded nature of the segmentation and alignment steps (which do not apply generally to all cases), their approach only solves 25% of ARC tasks.

2.3 Key takeaways

- Recurrence is a fundamental requirement for the ability to learn algorithms.
- Improved versions of the Neural GPU, the Universal Transformer, and Deep Thinking Systems have all demonstrated the capacity to solve algorithmic tasks in a way that generalizes beyond the training set.
- Without intermediate supervision, highly composite tasks are difficult or impossible to solve for (deep) learning approaches.
- Curriculum learning is a good way to provide the necessary intermediate supervision.
- DSL-based approaches can also solve simple algorithmic tasks.
- DSL-based approaches suffer from a combinatorial explosion problem that prevents them from being practical for more sophisticated tasks.
- DreamCoder, in principle, mitigates this combinatorial explosion because it progressively learns new components in its DSL.

3 Neural Networks vs DSL Approaches

Coming back to ARC, one might ask: why have DSL-based approaches outperformed neural networks so far? **The answer is simple: DSL-based approaches are high in bias, low in variance, while neural networks are high in variance, low in bias. As a result, due to the very small size of the ARC training set, strong inductive biases and priors are necessary**. Thus, attempting to train any neural network approach, even a Turing-complete one, from scratch on the ARC dataset is almost certain to fail. More generally, any low bias, high variance learner trained from scratch on the ARC dataset will fail.

In the following subsections we explore various criticisms and difficulties associated with both approaches.

3.1 Multitasking

In [15], the following argument is made:

> "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".

At the end of that sentence they cite as supportive evidence a 2019 paper on quantifying generalization in reinforcement learning. This argument probably no longer holds much weight in 2023, considering that we have seen various successful generalist models such as:

- A generalist reinforcement learning agent by DeepMind [35]
- ChatGPT and other LLMs capable of few-shot learning new tasks via in-context learning

It should be clear at this stage that a sufficiently large neural network is indeed able to learn to multitask, and even to adapt on-the-fly to novel tasks. In the proposed solution, we will train a model using supervised in-context training. This will unlock the required few-shot learning capability that makes it possible to train a model that will perform multiple different tasks.

It should be noted that neither the Neural GPU nor Deep Thinking Systems have demonstrated a capacity for in-context learning, therefore they will be omitted from the proposed list of algorithm candidates.

3.2 Cumulative approximation error

In [15], the following argument is made:

> "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"

This argument is partially true. That is, it is false that it fails to generalize as soon as the range of the training set is exceeded (as shown in the Literature Review). However, it is true that it fails to generalize consistently to infinity. That being said, one should be careful in interpreting the implications of this. There are 2 possible explanations for this phenomenon. Either:

1. A failure of systematicity: The neural networks don't correctly learn the underlying algorithmic process, and instead only learn a direct mapping from inputs to outputs (i.e. "compressed memorization").
2. A failure of productivity: The neural networks correctly learn the underlying algorithmic process, but the fact that the weights are continuous-valued means that there is a small residual error that accumulates as we keep iterating in the reasoning process.

All evidence presented in the previous sections point to the 2nd one being the case, not the first one. Indeed, if the 1st explanation were the case, the experiments in algorithmic extrapolation would fail immediately as we step beyond the training data's domain. The model could not have memorized what it has never seen. Instead, the performance gradually decreases as the cumulative error increases.

The author of this essay has performed a few experiments in algorithmic extrapolation that confirm and explain this problem. They are easy to reproduce and can be attempted by the skeptical reader.

In the experiment, a RNN was trained to count the number of non-zero pixels in an input sequence (one-hot-encoded for a total possibility of 10 different colors). The goal of the RNN is to output, for each of the 9 possible non-zero colors, the pixel count from the grid. If the rounded scalar value for each predicted count corresponds exactly to the ground truth, the prediction is considered a success. If even only 1 value is off, it is considered a failure.

The RNN was trained on grid samples of dimensionality from 1x1 to 7x7 inclusively. It was then tested on grids of dimensionality 18x18, and the generalization accuracy was 99.7%. However, as the grid dimensionality was increased beyond that, the generalization accuracy decreased. For example, at 20x20, the accuracy was around 75%.

Thanks to the simplicity of this experiment, it was easy to inspect the weights to understand what the RNN learned and why it failed to generalize "infinitely" beyond the training set. Indeed, the recurrent neural network's core unit is simply a linear layer that maps a vector of length 19 to a vector of length 9. The input to this linear layer consists of 10 elements for the input (one-hot-encoded color of the pixel x_t), concatenated with the hidden state h_t of dimension 9. The output is the new hidden state h_{t+1}, which has dimensionality 9 as well. This is presumably the simplest RNN structure that is able to solve this problem.

For each index i in the output h_{t+1}, the weights that were learned corresponded to a value of 1 for the connection from $x_{i+1,t}$ to $h_{i,t+1}$, as well as a value of 1 for the connection from $h_{i,t}$ to $h_{i,t+1}$. The other weights all had a value of zero.

In summary, it learned to add the one-hot-encoded color value in the current grid position to the previous hidden state's value at that position. This means that the hidden state acts, as one would expect, as an internal counter for each of the 9 non-zero colors. The fact that $x_{0,t}$ is not connected to anything means that it has learned to ignore the 0-valued pixels as intended. From this we can conclude that it has indeed learned the correct algorithm: it is not merely "memorizing" a mapping from a grid to a number. It is, instead, truly keeping an internal count for each color and incrementing the appropriate counter by 1 each time it sees a certain color in the input sequence.

However, there is a catch. These values of 1 and 0 for the weights are not exact. Instead, we have weight values such as 1.00002, or 0.00031, for example. As we iteratively multiply and sum these weight values, the hidden state's internal counter starts accumulating counting error. When the sequence is long enough, eventually that error rounds up to 1 more (or less) than the true

answer. This is the only reason the model is unable to generalize "infinitely" beyond the scale of the training set.

If that is not proof enough, you can convince yourself by simply applying the rounding operation to the learned weights after training, but before test inference time. With exact weights, you will find that your learned model will indeed generalize to the limits of your computational power.

This simple experiment explains why recurrent neural networks are successful at learning the iterative application of fundamental principles in a way that generalizes – or, rather, continuous-valued approximations thereof. This explains (at least in part) the gradual deterioration with increased "iterative depth" that we see in all papers that report algorithmic extrapolation results from deep learning.

In principle, this is indeed a problem that ought to be addressed for the topic of algorithmic extrapolation from continuous-valued neural networks. It does truly mean that we can never have such a neural network that will be able to count, or perform arithmetic operations, that generalize to infinity (regardless of computational resource limits).

To be able to do this, it seems that we need some sort of mechanism that "snaps" the hidden state back to the nearest discrete concept. In the aforementioned counting experiment, the rounding operation serves that purpose. For problems where we deal with discrete numbers, perhaps rounding up the weights is enough. However, this is not a generalizable solution.

Solving this problem is beyond the scope of this essay, although what comes to mind is a prototypical network-like approach where vectors are projected into a metric space, and the nearest cluster center (prototype) is kept and used for the next recurrence operation, rather than the previous result directly. This way, the intermediate results of each iteration are forced to "snap back" to a core prototype concept, rather than allowing it to gradually drift away. The trick is, as always, how to do this in a differentiable manner?

Pragmatically speaking, though, it is not obvious that this cumulative error problem truly is an issue for the approach that will be proposed here. In ARC, we will train on problems of similar or even greater reasoning depth than on the test set (because, as will be seen, we will generate that training data ourselves). Therefore, extrapolating to significantly longer or more complex problems should not be necessary. That is, if the ARC challenge consisted of training on grids of up to 30x30, and testing on grids of 500x500 – then yes, this would be a problem that needs to be tackled. Until then, approximation is enough.

3.3 Tabula rasa

DSL-based solutions come with very explicit, hand-crafted domain knowledge priors: the function primitives. On the other hand, neural networks, if not pre-trained on anything, are almost entirely tabula rasa. The only exception to this are its inductive biases that come from its architecture (for example, the convolutional bias in CNNs).

This implies that DSL-based approaches are well suited to the small training dataset that comes with the ARC challenge. Neural networks, and indeed any low-bias/high-variance machine learning algorithm, are incapable of learning such complex concepts from so few examples. **The idea that humans are able to solve ARC challenges from only a few examples, therefore a machine learning algorithm should be able to do the same, is a false comparison**. Humans benefit from billions of years of learning through the process of evolution. That is a lot of data to learn from.

In order to address this, in the case of the ARC challenge, a data simulation framework will have to be built. More than that, it's not just about the quantity of data examples, but also how they are presented to the learner.

3.4 The needle in the haystack

As discussed in section 2.2.2, DSL-based methods suffer from a combinatorial explosion problem. This is a serious problem for these methods, although, in theory, DreamCoder should be able to mitigate this issue because of its ability to gradually simplify the search space as it learns new subroutines.

Though less obvious, gradient descent also suffers from this problem for cases where the gradients are uninformative for a vast area of the loss landscape. This has been discussed in length in section 2.1.5. The conclusion from this research is that, through the use of intermediate concept supervision, such as curriculum learning, it has been proven that all tasks are learnable.

More details on how this will be accomplished will be provided in the Solution Proposal section.

3.5 Abstract visual reasoning

So far, this essay has mostly focused on relatively simple algorithmic tasks in order to showcase deep learning's much underestimated ability to learn algorithms. However, one should not forget that the complexity of ARC tasks goes much beyond these. In a sense, ARC is closer to abstract visual reasoning benchmarks such as the Synthetic Visual Reasoning Test (SVRT)[2] or Raven's Progressive Matrices (RPM)[1].

These benchmarks are not complex from a visual perception standpoint, their difficulty lies in the spatial abstraction notions that the learner needs to represent in order to solve them.

On these types of benchmarks, even though different approaches other than deep learning ones have been attempted, the state-of-the-art[36] is formed largely by the latter. On SVRT, it is a recurrent vision transformer architecture[32](RViT) that currently achieves the best performance. It is interesting to note that the RViT architecture is essentially a universal transformer augmented with an CNN module.

RPM is one of the most popular visual reasoning benchmarks. Initially, researchers[7] attempted to solve it via rules and heuristic-based algorithms.

Since then, however, deep learning approaches have emerged as the superior ones[31].

3.6 Summary & Discussion

In both approaches there are task-specific routines that need to be coded. In the DSL case, it is to provide the building blocks to the solutions. In the Deep Learning case, it is to generate training data. In both cases, too, the intermediate building blocks have to be coded.

The differences between both approaches are as follows:

- In the DSL case, the programmer must figure out how to solve a task. That is, the primitive functions that are coded must all be elements of a solution. And, even though the search algorithm takes care of figuring out how to combine these components, the programmer must somehow make sure that all of the elements of a solution are there to begin with. This means, in some sense, that they must already have a good idea of how the end problem is to be solved.

- In the deep learning case, the programmer must only define the task, for the purpose of training data generation. The code written is agnostic to the method of solution, it only matters that the needed input-output examples are generated.

In the DreamCoder case, it would appear that both types of coding are necessary. If one uses a very high-level, task-specific DSL, the function primitive coding is more effortful, but less data generation is required. If instead the DSL is very low-level, we end up with an approach very similar to deep learning cases where the learning component does most of the work. Consequently, DreamCoder is a sort of hybrid solution that is difficult to place strictly into DSL or deep learning categories. It depends on how it is used.

Building a DSL, in particular making sure that all the function signatures compose correctly, is more difficult than just coding a data generator for the same task. Making sure that your DSL contains all the necessary components to build a solution to the problem becomes quickly intractable as complexity increases. Additionally, as a task increases in level of abstraction, it becomes more difficult to hard-code a solution for it.

It is, after all, for this very reason that machine learning was invented. Some notions are too complex or ambiguous to define via strict "if/else" type of coding. Instead, it is easier to use optimization methods to automatically learn a latent representation of an object or process from examples.

As a consequence of that, **while the performance of Deep Learning and DSL approaches on simple algorithmic tasks is comparable, when it comes to more complex and abstract visual reasoning benchmarks, Deep Learning establishes the state-of-the-art**. The notable exception to this is the ARC challenge, for reasons that have already been explained.

There is, also, a deeper case against methods that require too much of a high-level or task-specific DSL. Even if such an approach were to successfully complete the ARC challenge, it would require a high degree of re-engineering when changing the problem domain.

It is questionable whether one can be said to have implemented general intelligence, even if it somehow solves ARC, by merely running a search on hard-coded blocks of code. The brittleness and specificity of such an approach is self-evident (for one thing, there is no actual learning).

Because so far the state of the art on the ARC challenge has been achieved by heavily DSL-based approaches, it is tempting to keep pursuing that direction. However, we are in danger of falling prey to the First Step Fallacy: the idea that because preliminary steps have brought us closer to a goal, it automatically implies that we can keep going in that direction until we reach it. A more poetic analogy would be that of a man building a ladder to the moon. From the first version of the ladder that he built, he sees that he is now closer to the moon. So he proceeds to constantly add steps to his ladder.

Keeping this analogy in mind, the fundamental argument made in this proposal is as follows: DSL-based approaches are the ladder. They progressively brought us closer to a solution to the ARC challenge. But pursuing in that direction would require effort equivalent to building a ladder to the moon. Instead, we must build a rocket. That is, we must build a much more powerful and automated approach: a Turing-complete learning algorithm deployed on a scale equivalent to that of LLMs. Also, much like building a rocket, a lot of initial engineering effort will be spent without much visible progress.

3.7 Key takeaways

- DSL-based approaches currently achieve the best performance on the ARC challenge, because they are high-bias, low-variance techniques that don't require a lot of data.
- Recent achievements of deep learning show that it is capable of learning to solve a variety of tasks with one model - such as with in-context learning.
- Performance of deep learning methods gradually deteriorates as the number of iterations required to solve a problem go beyond the training set.
- This cumulative approximation error is due to inexact weights, causing the minuscule error to grow over inference iterations while solving a deep problem.
- There is no *a priori* reason to believe that this is a problem for the ARC challenge, due to predetermined grid dimensions and arbitrary computational depth for the generated training data.
- Generating large quantities of data, over a vast distribution of tasks, will be necessary to train our solution.

- Curriculum learning is necessary to allow the model to learn intermediate concepts.
- Abstraction visual reasoning is currently dominated by deep learning (especially transformer) methods.
- Due to the combinatorial explosion and the inability of DSL-reliant methods to solve complex visual reasoning tasks, the choice of deep learning solutions will be preferred.
- One exception to this is DreamCoder, which addresses the combinatorial explosion problem.

4 The Solution Proposal

4.1 Requirements

The previous sections can be used to deduce the requirements for a solution:

1. It must be Turing-complete: it must be possible to learn an algorithm (rather than only a mapping), that involves a dynamic, arbitrary number of loops.
2. It must be able to multitask, and operate in a meta-learning setting, such as via in-context learning.
3. It must use a form of intermediate supervision, such as curriculum learning.
4. Data must be generated/simulated, since there is not enough data in the ARC training set.
5. It must be able to scale to complex visual reasoning problems.
6. It will require a lot of computing resources, possibly at the same level as LLMs.

From the literature review of learning algorithms, there are only two options that can be structured to fit this these requirements:

1. Universal Transformers[13], possibly augmented with external memory[25]
2. DreamCoder [20]

In addition to architectural decisions, it has been determined that one cannot simply use these algorithms "as is" on the ARC training set: the training regime matters as much, if not more so, than the choice of algorithm itself. This part is, in fact, the most elaborate and difficult aspect of building a successful ARC solution.

The approach proposed in this work uses supervised learning using a custom-made data generator. This is in stark constrast with LLMs, which are trained mainly by self-supervised learning on publicly available datasets. Applying the self-supervised paradigm to the ARC challenge is highly uncertain, given the radical difference between the ARC domain and real-world images and video. It is unlikely that such an approach would eliminate the need for a task generator.

In addition, it is also potentially "overkill", in a way, because the real world is a vast superset of the ARC. Presumably, over 99.99% of the observed data samples would be irrelevant to ARC.

4.2 Curriculum learning

When it comes to curriculum learning, there are different ways of implementing it, and they are not equal. In the Learning to Execute[4] paper, they compare 3 different methods.

The "naive" strategy consists of first training on a dataset of minimal complexity/scale/difficulty, until convergence. Then, the complexity, scale or difficulty is increased by a small amount, and model training is resumed on that new dataset. This process is repeated an indefinite amount of times.

In the "mixed" strategy, instead of starting with only a small level of complexity, and increasing gradually, training samples are drawn from random levels of complexity. As a result, a training batch may consist of small, intermediate examples as well as full-blown task examples. This approach is arguably not much different than the usual random sampling of batches, but what still makes it curriculum learning is the fact that examples smaller or less complex than the intended task are available in the training set.

Finally, in the "combined" strategy, every training sample is either drawn (randomly) from the "naive" strategy or the "mixed" strategy. This hybrid approach was the one that performed best in their experiments. It always outperformed the "naive" approach, and it generally (but not always) outperformed the "mixed" strategy.

To be more specific to our problem, curriculum learning must be adapted to multitask learning. That is, we are not merely fine-tuning a few continuous parameters such as sequence length: we are trying to learn multiple distinct tasks of varying levels of complexity.

This implies a definition of the level of difficulty of a task, so that we can arrange the training regime according to it. It can be set manually when the task generation code is written, or it can be calculated dynamically at training time, as in some automated curriculum learning strategies. Because writing code that generates the training data will be necessary in our case, it will be simpler and more appropriate to set these difficulty levels manually at development time. Using automated curriculum learning strategies would only add an additional potential point of failure, and complexify the solution beyond what is necessary.

Setting these curriculum levels is not an exact science, but a good rule of thumb is to estimate how many directly learnable sub-tasks are required to solve the given task. A level of 0 means that the sub-task is so simple that it

can be learned directly. An example of such a task is the one described in the pixel count experiment from section 3.2 (cumulative approximation error). A task that would require counting pixels and returning the color that has the most instances (or the least instances) would be a level 1, because it combines two "primitive" tasks. More examples of tasks and corresponding curriculum settings will be given in section 4.4 (data generation).

In short, using manually defined task complexity settings, the proposed solution will present training task samples using the "combined" curriculum strategy.

4.3 Multitasking and meta-learning

ARC uses a few-shot learning problem setting, where a few support examples are presented, the algorithm must quickly learn how to solve the task from these examples, and the query set is presented, which the learner must solve. It is, therefore, a meta-learning dataset.

If opting for the Transformer-based models, the input sequence will contain a concatenated description of the input-output examples, and the output sequence will start with a description of the input test grid. The model will then be expected to fill in the output grid by predicting the next tokens in the output sequence.This is what is referred to as in-context learning.

For the DreamCoder option, separate input-output examples (in the form of sequences for consumption by the neural network component) are provided in a task batch. This is the default method of use explained in the original paper.

4.4 Data generation

Deciding what tasks need to be generated is probably the most important aspect of this solution. Doing this properly is crucial to the success of this approach.

The ARC challenge is intended to capture human-like (two-dimensional) visual reasoning in general: the held-out test set can be made up of essentially any task, as long as it makes sense to a person. Therefore, generating relevant training data means summarizing all of the human core knowledge elements that are necessary for such visual reasoning. This is, of course, no easy task.

The underlying assumption is that there are core "skills", like building blocks of intelligence, that have been perfected via evolution (as well as learned in infancy) to solve the reasoning problems encountered in human life. These core skills are finite in quantity, and composable in order to allow efficient adaptation to novel instances of problems. In a sense, there is no such thing as a truly novel problem: there are only novel permutations of already known principles. A useful training data generator, then, will need to be designed with these core principles in mind: we are not interested in solving random, arbitrary tasks (by virtue of the "No free lunch" theorem, such a thing is not possible anyway).

When an athlete is training for a sport, they do more diverse and specific exercises than merely playing the game itself. There are exercises designed to improve cardio, specific types of flexibility, specific muscles, specific skills, etc.

Together, these are indirectly meant to improve success at the final result, the sport itself.

In a similar way, the training regime for the ARC solution should not only focus on solving complete ARC tasks. More basic exercises, corresponding to core skills and simpler sub-tasks, must also be included. Otherwise, as has been explained in the previous sections, learning will fail due to the absence of intermediate supervision.

With regards to which core skills should be focused on, cognitive science offers some clues[3]. Visuo-spatial reasoning involves groups of notions such as cardinality, objectness, geometry, relations (such as sameness/difference), positioning, transformations, symmetries, patterns, repetition. This list is probably not exhaustive. Unfortunately, it would seem that trial and error is the only way to arrive at a truly complete list (the process of discovery will be further described in the Roadmap section). The currently proposed list of core skill groups will be defined as follows:

1. **Cardinality**: the ability to count, to perform cardinality related comparisons such as equal, greater than, less than, maximum, minimum. The ability to perform arithmetic operations such as addition, subtraction, division, multiplication, modulo. Notions of odd and even (parity).

   Example of a task: random pixels drawn at random locations with random colors on the grid. The output is a 1x1 grid that contains the color that has the most pixel instances.

   Proposed curriculum level: 1. (must count pixels, then must learn to apply the "maximum" concept)

2. **Objectness**: the ability to "detect" objects, and separate them from the background (in a way that is roughly equivalent to how humans would do it). The notion of collision between objects. Distinguishing different objects.

   Example of a task: a square of random dimensions, position and color is drawn on the input grid. The output grid is a clipped grid that contains only this square (with the same color and dimensions).

   Proposed curriculum level: 0.

3. **Geometry**: the notion of "Platonic" shapes. Pure straight lines, triangles, squares, rectangles, circles.

   Example of a task: 4 red pixels are positioned randomly on the input grid. The output must draw a filled yellow rectangle that is contained within these 4 pixels.

   Proposed curriculum level: 2. (need positioning, because the drawn rectangle must be placed relative to the 4 existing pixels, and objectness to identify these 4 anchor points as distinct objects)

4. **Relations**: this is the most abstract group of core skills, and involves (visual) relational reasoning such as same vs different, bigger than and smaller than, "odd one out", is part of (is a component of), is a type of, is inside of, etc.

   Example of a task: a few shapes are drawn in the input grid. Only one of them is surrounded by a red square. The output grid must be a copy of what is inside the red square.

   Proposed curriculum level: 1. (need objectness notions to identify the distinct shapes, especially the red square)

5. **Positioning**: the notion of relative positions - to the right of, to the left of, below, above, top-right, etc. Also involves more absolute positions such as the top-left quadrant of the grid, the left half, the right half, the bottom-right quadrant of the grid, etc.

   Example of a task: random shapes are drawn on the input grid. The output grid is the content of the top-left quadrant with all shapes rotated clockwise by 90 degrees.

   Proposed curriculum level: 2. (need objectness to identify the objects, and transformations to apply the rotations)

6. **Transformations**: geometric transformations such as scaling, clipping, translation, rotation, horizontal and vertical flipping. Also includes color-based transformation, i.e. changing color and filling in shapes with a certain color.

   Example of a task: a random shape of random scale, color and position is drawn. The output grid is simply the input grid with the shape flipped horizontally.

   Proposed curriculum level: 1. (requires objectness)

7. **Symmetries**: this is the notion in the geometric sense, i.e. the idea that objects or grids can be composed of reducible components applied as a "mirror image".

   Example of a task: various vertically symmetrical shapes are drawn on the input grid. The output consists of a copy of the input grid where the top half of each object is removed.

   Proposed curriculum level: 1. (requires objectness)

8. **Patterns**: the ability to detect repeating, predictable groups of pixels or objects.

   Example of a task: A square pattern of pixels is tiled on the input grid. In the output grid, only 1 instance of this pattern must be drawn. The output grid is therefore of the same dimensions as the repeating pattern.

Proposed curriculum level: 0.

9. **Repetition**: the ability to iterate and repeat patterns while generating the output grid. This is one of the core knowledge groups that most directly expresses the need for a learner that is able to execute dynamic loops.

   Example of a task: a pixel is drawn with a random color and position, in a grid of random dimension. In the output grid, this pixel must be duplicated iteratively to the right until the grid edge is reached (effectively turning it into a straight line).

   Proposed curriculum level: 0.

In summary, one must build a data generator in the sequence-to-sequence format that represents tasks from the most atomic, basic core skills to the most complex composite tasks possible. With each of these, an associated curriculum difficulty level must be carefully selected in accordance to the ideas presented in section 4.2. This is the aspect of the proposed solution that will require the most engineering effort, as well as trial and error. Simply crowdsourcing data generation will not work: people will input complete ARC tasks. This dataset will be unlearnable, although it can serve as a test set, for example.

With regards to the actual core principles of the tasks to generate, they have to be designed manually (as this list was). By definition, if we had an automatic method that was able to design all of the required tasks, we would have an algorithm that already understands all of the necessary core principles (and thus, with very little training, could presumably solve ARC).

That being said, it is possible to develop algorithmic methods of variation on the core principles that will "augment" the tasks in a similar way to data augmentation in machine learning. At the very minimum, aspects such as pixel color and position randomization, or random shape generation, can be automated. One might even, to some extent, randomize variations on goals of the task, by automatically generating the possible combinations of required sub-tasks. For example, if we have a sub-task whose goal is to clip the content of a red square and use it as output, and a sub-task whose goal is to rotate shapes by 90 degrees clockwise, then the sequential composition of these two sub-tasks yields a composite task where the output grid must contain the rotated content of the red square.

4.5 Roadmap and scaling

It is unlikely that a medium-scale architecture trained on a home PC will be able to learn to solve the ARC challenge. Given all of the necessary core knowledge that must be learned to solve it, and the implication that this model is essentially an "Artificial General Intelligence"(AGI) model, a computational scale equivalent to today's largest LLMs will possibly be needed.

That being said, it is desirable to first aim for a proof-of-concept version that demonstrates the validity of the approach, before investing additional resources

into the full-scale solution. This is why this section will propose developmental steps towards the final version that can solve ARC.

Stage 1: validating algorithmic extrapolation

This initial stage consists of validating the idea that the specified algorithms are able to learn core skills from the training curriculum, and to properly combine them when solving novel composite tasks.

Step 1: develop a reasonable number of tasks (100?) for a particular core skill group, such as cardinality. These tasks cannot rely on other core skills, since those will not have been trained on. These first tasks can potentially be implemented using relatively small grids (10x10?) to reduce computational effort, and to delay the need to address the long-range sequence problem (see section 6.3 in the Appendix).

Step 2: train one (or both, for comparison) of the algorithms that fit the aforementioned requirements on the task generator developed in Step 1. This training must use the curriculum training regime explained in previous sections. The training must be done such that there is a held out subset of tasks, to be used only to evaluate the generalization ability of the trained models. In other words, we are interested in testing cross-task generalization, rather than generalization from some samples of all tasks to new samples of these same tasks.

Step 3: evaluate performance on the held-out set. Here, two results might occur: either the performance is near perfect, or the performance has degraded relative to the training set. If it is the former, this stage is complete and we can proceed to stage 2. For good measure, however, it would be better to repeat this experiment with a new group of core skills, and a larger set of tasks - to reliably confirm that the idea works.

If, however, the performance deteriorates significantly from the training set, **don't panic**. Here are some plausible reasons for failure:

1. The test set may require intermediate concepts not seen in the training set. This is possible at this stage, because the number of tasks is relatively small. Review the generated tasks, and make sure all necessary intermediate concepts are there. It is then possible to re-arrange the training/test split, or to add new tasks, to solve this deficiency.

2. There simply might not be enough distinct training tasks to learn in a way that generalizes. By adding more training tasks and verifying if the test performance has increased, one can easily test this hypothesis.

3. The assessment of curriculum levels could be faulty. Does it need more granularity? Is the algorithm able to learn the intermediate concepts correctly before trying to learn the composite tasks?

4. Perhaps it is a weakness in the learning algorithm itself. Consider trying the other one (DreamCoder or Universal Transformers). If both have been

tried, potential implementation improvements in the appendix could help as well.

If all of the above fails, it would be reasonable to conclude that some assumptions underlying this suggested approach are incorrect. More research would need to be done to achieve successful algorithmic extrapolation for the implemented group of tasks.

Stage 2: validating the reasoning power of core skills

The purpose of this stage is to test the dual hypothesis that:

1. there is a finite and reasonable number of core skills that need to be learned
2. the apparently infinite number of possible ARC tasks merely stems from the countless possible variations or combinations of these "few" core skills

If that hypothesis is true, then learning a new core skill group will unlock success on a superlinear, rather than linear or sublinear, number of tasks. To verify this, then, one must test performance on ARC, gradually add new (sub)tasks to the data generator, re-train and re-evaluate. The new number of ARC tasks solved should be significantly greater than the number of distinct tasks that were added for the new skill group.

Indeed, if it is not the case, it implies that one would need to implement a quasi-infinite number of new tasks in the data generator to be able to solve the correspondingly quasi-infinite number of possible ARC tasks. Such a scenario would imply either a failure of the hypothesis behind this approach, or a failure of the algorithm to learn these core skills in a composable manner (although, in principle, stage 1 should have confirmed the latter). In both cases, the approach proposed in this essay would need to be revisited.

Step 4: test on the ARC training set whether the trained model is able to solve a non-zero amount of tasks.

Step 5: implement a new group of skills and re-train the model on the new full dataset (including the new core group). The objectness core skill group is probably a good choice at this stage, if it wasn't used in stage 1, because it unlocks a lot of diversity in the possible tasks to implement. Additionally, most ARC tasks require at least some notion of objectness.

Step 6: re-test on the same ARC training set: did the number of successfully solved tasks go up? By how much? This ideally should be repeated with 2 or 3 new core skills, to confirm preliminary results. Estimate the number of new ARC tasks that are successfully solved per new distinct task implemented in the data generator. If this ratio is significantly greater than 1, the approach has been essentially validated. It is then recommended to move to stage 3.

Stage 3: full-scale deployment

At this stage, the assumptions behind the current approach have been largely validated, and one can proceed to the final step of scaling up the solution.

Step 7: finish developing the data generator (that is, implementing the core skill groups that have been previously enumerated)

Step 8: increase the maximum grid size for the generated problems. Implement the necessary modifications to the Universal Transformer to support long sequences (see Appendix for pointers).

Step 9: using large-scale deep learning computing resources, train variants of the selected models with an increased hyperparameter scale. Note the test performance on the ARC validation set after each training phase. Repeat until convergence.

Step 10: if the performance plateau reached by scaling up the model is unsatisfactory, the first step is to troubleshoot which types of ARC tasks it fails on, and ask: is there a core skill that is missing in the training data?

4.6 Key takeaways

- Key requirements of algorithm candidates include: computational universality, meta-learning, ability to scale in complexity (e.g. for abstract visual reasoning).
- A data generator will be written, that incorporates curriculum learning and covers the core skill groups that were identified.
- A smaller scale proof-of-concept will first be built to validate the underlying hypotheses of this approach.
- The central validation principle to be to test whether the number of new ARC tasks solved scales superlinearly with respect to the number of new distinct tasks implemented in the data generator.

5 Conclusion

In summary, we started by analyzing the ability of deep learning approaches to learn algorithms. By surveying the literature, we showed that deep learning approaches using the appropriate inductive biases and trained using intermediate concept supervision are able to learn algorithms. Their capabilities on relatively simple algorithmic extrapolation tasks are comparable to that of DSL-based methods.

One exception to this fact is the cumulative approximation error problem, which is found in deep learning methods due to the use of continuous-valued weights. Gradient descent yields slight approximation errors that accumulate at inference time when iterating on a scale that goes well beyond the training set. While this is admittedly a challenge for some problem domains, it does not appear to be an issue for the ARC challenge, due to the predetermined maximum grid sizes and the arbitrary complexity that we can implement in our own generated task data.

We then explored more complex and abstract visual reasoning benchmarks, solved by larger scale algorithms. We showed that deep learning-based (especially Transformed-based) solutions dominate in those areas, while reports of DSL successes are missing. The notable exception to this is the ARC challenge, where DSL is still the state-of-the-art, however the reason is due to a lack of training data. Indeed, ARC is the only benchmark on which DSLs still dominate. It is also the only benchmark that lacks a sufficient training dataset. Finding the training set is, arguably, the heart of the ARC challenge.

In section 4, the solution proposal was made. The known algorithms that satisfy the requirements are DreamCoder and Universal Transformers (and its variants). The importance of a training regime designed around core knowledge principles, and that uses curriculum learning, was emphasized. The astute reader will have noticed a subjective preference for the Transformer-based approach, even though no theoretical argument was provided against using DreamCoder.

This is simply the result of the author's personal experience with both approaches: DreamCoder is extraordinarily difficult to use and to customize. Transformer-based methods, on the other hand, are much easier to experiment with. Both approaches are treated as theoretically equivalent, but there are several practical advantages (if only usability) to using Transformers over DreamCoder.

Finally, a step-by-step roadmap was presented. It will allow the researchers who follow it to first validate the approach on limited resources, and then to scale it up for a final evaluation. It should also be reiterated that the proposed methodology is just that: a methodology. Realistically, as the steps in the roadmap are attempted, unexpected issues will arise. The appendix is an attempt at anticipating these problems and providing pointers to the necessary literature.

References

[1] John C Raven and John Hugh Court. *Raven's progressive matrices and vocabulary scales*. Oxford Psychologists Press Oxford, 1998.

[2] François Fleuret et al. "Comparing machines and humans on a visual categorization test". In: *Proceedings of the National Academy of Sciences* 108.43 (2011), pp. 17621–17625.

[3] Giorgio Vallortigara. "Core knowledge of object, number, and geometry: A comparative and neural approach". In: *Cognitive Neuropsychology* 29.1-2 (2012). PMID: 22292801, pp. 213–236. DOI: 10.1080/02643294.2012.654772. eprint: https://doi.org/10.1080/02643294.2012.654772. URL: https://doi.org/10.1080/02643294.2012.654772.

[4] Wojciech Zaremba and Ilya Sutskever. "Learning to execute". In: *arXiv preprint arXiv:1410.4615* (2014).

[5] Çağlar Gülçehre and Yoshua Bengio. "Knowledge matters: Importance of prior information for optimization". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 226–257.

[6] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[7] José Hernández-Orallo et al. "Computer models solving intelligence test problems: Progress and implications". In: *Artificial Intelligence* 230 (2016), pp. 74–107.

[8] Lukasz Kaiser and Ilya Sutskever. "Neural GPUs Learn Algorithms". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1511.08228.

[9] Eric Price, Wojciech Zaremba, and Ilya Sutskever. "Extensions and limitations of the neural gpu". In: *arXiv preprint arXiv:1611.00736* (2016).

[10] Jacob Devlin et al. "Robustfill: Neural program learning under noisy i/o". In: *International conference on machine learning*. PMLR. 2017, pp. 990–998.

[11] Karlis Freivalds and Renars Liepins. "Improving the neural GPU architecture for algorithm learning". In: *arXiv preprint arXiv:1702.08727* (2017).

[12] Shai Shalev-Shwartz, Ohad Shamir, and Shaked Shammah. "Failures of gradient-based deep learning". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 3067–3075.

[13] Mostafa Dehghani et al. "Universal Transformers". In: *Proceedings of International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=HyzdRiR9Y7.

[14] Mirac Suzgun et al. "LSTM Networks Can Perform Dynamic Counting". In: *ACL 2019* (2019), p. 44.

[15] Andrzej Banburski et al. *Dreaming with ARC*. Tech. rep. Center for Brains, Minds and Machines (CBMM), 2020.

[16] Emilio Parisotto et al. "Stabilizing transformers for reinforcement learning". In: *International conference on machine learning*. PMLR. 2020, pp. 7487–7498.

[17] Andreas Robinson. "Progress Extrapolating Algorithmic Learning to Arbitrary Sequence Lengths". In: *arXiv preprint arXiv:2003.08494* (2020).

[18] Simon Alford. "A Neurosymbolic Approach to Abstraction and Reasoning". PhD thesis. Massachusetts Institute of Technology, 2021.

[19] Rohan Deshpande, Jerry Chen, and Isabelle Lee. "RecT: A Recursive Transformer Architecture for Generalizable Mathematical Reasoning." In: *NeSy*. 2021, pp. 165–175.

[20] Kevin Ellis et al. "DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 835–850. ISBN: 9781450383912. DOI: 10.1145/3453483.3454080. URL: https://doi.org/10.1145/3453483.3454080.

[21] Maxwell Nye et al. "Show your work: Scratchpads for intermediate computation with language models". In: *arXiv preprint arXiv:2112.00114* (2021).

[22] Avi Schwarzschild et al. "Can you learn an algorithm? generalizing from easy to hard problems with recurrent networks". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 6695–6706.

[23] Arpit Bansal et al. "End-to-end Algorithm Synthesis with Recurrent Networks: Extrapolation without Overthinking". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 20232–20242.

[24] Mingda Chen et al. "Improving In-Context Few-Shot Learning via Self-Supervised Training". In: *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2022, pp. 3558–3573.

[25] Samuel Cognolato and Alberto Testolin. "Transformers discover an elementary calculation system exploiting local attention and grid-like problem representation". In: *2022 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2022, pp. 1–8.

[26] Qingxiu Dong et al. "A Survey for In-context Learning". In: *arXiv preprint arXiv:2301.00234* (2022).

[27] Louis Kirsch et al. "General-Purpose In-Context Learning by Meta-Learning Transformers". In: *Sixth Workshop on Meta-Learning at the Conference on Neural Information Processing Systems*. 2022. URL: https://openreview.net/forum?id=t6tA-KB4d0.

[28] Kenneth Li et al. "Emergent world representations: Exploring a sequence model trained on a synthetic task". In: *arXiv preprint arXiv:2210.13382* (2022).

[29] Yuxuan Li and James L McClelland. "Systematic Generalization and Emergent Structures in Transformers Trained on Structured Tasks". In: *arXiv preprint arXiv:2210.00400* (2022).

[30] Pan Lu et al. "Learn to explain: Multimodal reasoning via thought chains for science question answering". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 2507–2521.

[31] Mikołaj Małkiński and Jacek Mańdziuk. "Deep Learning Methods for Abstract Visual Reasoning: A Survey on Raven's Progressive Matrices". In: *arXiv preprint arXiv:2201.12382* (2022).

[32] Nicola Messina et al. “Recurrent vision transformer for solving visual reasoning problems”. In: *Image Analysis and Processing–ICIAP 2022: 21st International Conference, Lecce, Italy, May 23–27, 2022, Proceedings, Part III*. Springer. 2022, pp. 50–61.

[33] Sewon Min et al. “MetaICL: Learning to Learn In Context”. In: *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2022, pp. 2791–2809.

[34] Andrew J Nam et al. “Achieving and Understanding Out-of-Distribution Generaliza-tion in Systematic Reasoning in Small-Scale Transformers”. In: *arXiv preprint arXiv:2210.03275* (2022).

[35] Scott Reed et al. “A Generalist Agent”. In: *Transactions on Machine Learning Research* (2022). Featured Certification. ISSN: 2835-8856. URL: https://openreview.net/forum?id=1ikK0kHjvj.

[36] Rufai Yusuf Zakari et al. “VQA and Visual Reasoning: An Overview of Recent Datasets, Methods and Challenges”. In: *arXiv preprint arXiv:2212.13296* (2022).

[37] Denny Zhou et al. “Least-to-most prompting enables complex reasoning in large language models”. In: *arXiv preprint arXiv:2205.10625* (2022).

[38] Elisabetta Cornacchia and Elchanan Mossel. “A Mathematical Model for Curriculum Learning”. In: *arXiv preprint arXiv:2301.13833* (2023).

[39] Adaptive Agent Team et al. “Human-Timescale Adaptation in an Open-Ended Task Space”. In: *ArXiv* abs/2301.07608 (2023).

[40] Noam Wies, Yoav Levine, and Amnon Shashua. “Sub-Task Decomposition Enables Learning in Sequence to Sequence Tasks”. In: *Proceedings of 11th International Conference on Learning Representations (ICLR)*. 2023. URL: https://openreview.net/pdf?id=BrJATVZDWEH.

[41] Jonas Witt et al. “A Divide-Align-Conquer Strategy for Program Synthesis”. In: *arXiv preprint arXiv:2301.03094* (2023).

[42] Zhuosheng Zhang et al. “Multimodal chain-of-thought reasoning in language models”. In: *arXiv preprint arXiv:2302.00923* (2023).

6 Appendix

6.1 Gated Transformer XL

Canonical transformers are difficult to train. It has been proposed that a gated transformer[16] improves and stabilizes training. These gated transformer blocks are essentially a TransformerXL-I block with an added Gating Layer after the multi-head attention module, as well as after the MLP module.

It is possible that using this as the unit over which recurrence occurs in the UT could yield improved results, or at least converge faster during training.

6.2 Grid-based positional encoding

In the ARC challenge, there is no complex visual perception *per se*, however we are fundamentally dealing with grids, rather than sequences. It could be useful to have an inductive bias that builds in the understanding that pixels that are either vertically or horizontally adjacent are closer to each other (this concept gets lost in the traditional sequence-based positional encoding).

To do this, the positional encoding should be calculated from the 2D matrix, before flattening it to a sequence for the model. A simple mechanism could represent the position as a 2-dimensional vector, with one dimension representing the X position, and another being the Y position. This could simply be concatenated to the token (or embedding thereof).

6.3 Long sequences

Large grids (30x30) would flatten to sequences of 900 tokens. For in-context learning, typically at least 6 such grids would have to be contained in the sequence. As a result, the input sequences would largely exceed the usual recommended length for a standard transformer block. This is likely to be a problem for the standard Universal Transformer approach.

There is, fortunately, a great deal of literature on long-range transformers. The transformer XL, and in particular the Gated Transformer XL mentioned in section 6.1, are good candidates. Augmenting the Universal Transformers to support long-range sequences will require experimentation and research, but it is not *a propri* a problem.