AUTOMATICALLY COMPOSING REPRESENTATION TRANSFORMATIONS AS A MEANS FOR GENERALIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

How can we build a learner that can capture the essence of what makes a hard problem more complex than a simple one, break the hard problem along characteristic lines into smaller problems it knows how to solve, and sequentially solve the smaller problems until the larger one is solved? To work towards this goal, we focus on learning to generalize in a particular family of problems that exhibit compositional and recursive structure: their solutions can be found by composing in sequence a set of reusable partial solutions. Our key idea is to recast the problem of generalization as a problem of learning algorithmic procedures: we can formulate a solution to this family as a sequential decision-making process over transformations between representations. Our formulation enables the learner to learn the structure and parameters of its own computation graph with sparse supervision, make analogies between problems by transforming one problem representation to another, and exploit modularity and reuse to scale to problems of varying complexity. Experiments on solving a variety of multilingual arithmetic problems demonstrate that our method discovers the hierarchical decomposition of a problem into its subproblems, generalizes out of distribution to unseen problem classes, and extrapolates to harder versions of the same problem, yielding a 10-fold reduction in sample complexity compared to a monolithic recurrent neural network. We further show that our method can compose learned spatial transformations to recover canonical MNIST digits from transformed ones.

1 INTRODUCTION

Learning to generalize to harder problems from solving simpler ones is a difficult challenge in machine learning because the learner must discover regularities that cover a combinatorially large space of problems from only limited data. To make progress towards this goal, this paper focuses on a particular class of problems that exhibit compositional and recursive structure: their solutions can be found by composing in sequence a small set of reusable partial solutions. The key perspective of this paper is to recast the problem of generalization to a problem of learning algorithmic procedures over representation transformations: discovering the structure of a family of problems amounts to learning a set of reusable primitive transformations and their means of composition.

Approaches to this challenge generally have either assumed pre-specified transformations but learned the structure, often with dense supervision, or learned the transformations but pre-specified the structure (see Sec. 5). Pre-specifying one or the other is a major limitation to learning programs that can adapt to different types of data. The contribution of this paper is to propose a direction towards learning both the structure and transformations together with sparse supervision. We propose the compositional recursive learner (CRL), which learns both the structure and transformations of a modular recursive program that iteratively re-represents the input representation into more familiar representations it knows how to compute with.

Our approach is based on the following core ideas. The first core idea is that training on a distribution of composite problems can encourage the spontaneous specialization of reusable computational units that perform transformations between representations. Rather than learning to solve a new problem from scratch, a learner can make an *analogy* between the new problem and previously en-

countered problems and re-represent the new problem into one it knows how to solve. The second core idea is to treat the construction of a program as a sequential decision-making problem over these transformations. This formulation not only implements loops, recursion, composition, and selective attention, but enables *metareasoning*, where the learner dynamically customizes its program complexity to the problem instance. The third core idea is to operationalize the learner in differentiable programs by using reinforcement learning algorithms to solve the discrete optimization problem of selecting which transformation to apply, while simultaneously learning the transformations with backpropagation.

Multilingual arithmetic problems are a challenging but minimal family of tasks that exhibit compositional and recursive properties (see Sec. 2). Understanding how to learn to decompose and exploit the structure of arithmetic is a first step towards solving any more complex domain, which would at least have the structure we study here in arithmetic. Our experiments on generalization to a completely unseen distribution of harder problems from only observing less than 3×10^{-4} of the training distribution show that CRL outperforms baselines without recursive or compositional reusability in their structure, even when the baseline is pretrained on all of the primitive transformations of the dataset. We further show that our method can compose learned spatial transformations to recover canonical MNIST digits from transformed ones.

Summary of contributions: (1) To consider a learner whose primitive transformations and means of their composition match the structure of the problem distribution, we narrow the scope of problems to those that exhibit compositional and recursive structure, which we formalize in Sec. 2. (2) In Sec. 3 we propose a hypothesis that the emergence of primitive transformations can be encouraged by training on a distribution of composite problems. (3) We frame the problem of learning the structure of a program as a sequential decision-making problem over these transformations. These two pieces comprise the CRL framework. (4) To implement CRL in differentiable programs, we present a formalism for expressing the construction of a neural network in terms of an Markov decision process that generates computation graphs. (5) Our experimental findings in Sec. 4 highlight the limitations of monolithic static network architectures in generalizing to structurally similar but more complex problems than they have been trained on, providing evidence in favor of learners that dynamically re-program themselves, from a repertoire of learned reusable computational units, to the current problem they face.

2 General Setup

It would be impossible to generalize if the universe exhibited no regularities. Discovering and exploiting the structure in their world helps humans generalize make predictions and solve problems in new situations. We formalize the notion of compositional structure in terms of *problems*, describe the characteristics of these problems, and define the challenge of how to learn programs that reflect the structure of problems.

Problems: To solve a *problem* P_i is to transform its input representation x_i into its output representation y_i. x_i and y_i are associated with their respective *types* t_x and t_y. In a *translational problem*, the input representation and the output representation differ (i.e. $t_x \neq t_y$). In a *recursive problem*, they are the same (i.e. $t_x = t_y$). Two problems P_i and P_j are of the same *class* if the types of their input representations and output representations match (i.e., $x_i = x_j$ and $y_i = y_j$). For clarity we focus on functions with one input and output; one can use currying to extend our analysis to functions with multiple inputs.

Composite problems: Problems can be compositions of other problems, which themselves can be composite. A *composite problem* $P_a = P_b \circ P_c$ means that the solution of P_a can be found by first solving P_c, then solving P_b with the output of P_c as input. In this case, P_b and P_c are *subproblems* with respect to P_a. This imposes constraints on the types of the input and output representations, so $t_{x_c} = t_{x_a}$, $t_{x_b} = t_{y_c}$, and $t_{y_a} = t_{y_b}$ must all hold, but t_{x_c} need not be equal t_{x_b}. Note that translational problems can have both translational or recursive subproblems, and vice-versa. Therefore, all composite problems can be expressed as a composition of translational and recursive subproblems.

Examples: Towers of Hanoi is a recursive problem (where both t_x and t_y are disk configurations), as is arithmetic (where both t_x and t_y are expressions). Robot sensorimotor control is a trans-

lational problem with translational and recursive subproblems: sensorimotor control can be broken down into a recognition model (a translation problem between $t_x =$`pixels` and $t_y =$`scene description`) , which itself is a composition of recursive problems (various image filters), and a control policy translating intermediate percepts to motor torques. Multimodal problems have different input and output types, but unimodal problems need not have the same type: translating French to English is a different class from translating French to Spanish.

The goal: Let $\mathcal{P}$ be a set of composite problems that share a set of regularities. We want the learner to generalize to solve translational problems of unseen classes (e.g., the learner was trained to translate French to English and English to Spanish but not French to Spanish). We also want the learner to extrapolate to solve recursive problems that have higher complexity than it has seen (e.g., the learner was trained to solve up to 5-length arithmetic expressions but is tested on 10-length expressions).

The challenge: In general the internal compositional structure of problems is unknown. For a particular problem P_i, the learner only is given an input and the output type (x_i, t_{y_i}) and is expected to produce output y_i. The only supervision signal it receives is its prediction error. The learner is trained on a set of problems $P_{train} \in \mathcal{P}$ and tested on a disjoint set of problems $P_{test} \in \mathcal{P}$. When data is limited, naïve approaches quickly overfit to P_{train} and do not generalize to P_{test}. To solve translational problems the learner must learn to make an analogy between the new problem and previously encountered problems by re-representing the new problem into a class it is familiar with. To solve recursive problems, the learner must learn an algorithm to iteratively reduce the harder problem into simpler problems of the same class. To do so, one powerful design strategy in constructing programs is to base the structure of the program on the structure of the system being modeled (Abelson et al., 1996); how does a learner automatically program itself to reflect the structure of each problem?

3 A Learner That Programs Itself

Learning both the structure and transformations of CRL from sparse superivision is challenging for several reasons. First, we need to encourage the "crystallization" of primitive transformations from data without any explicit feedback on what the transformations should be (Sec 3.1). Second, we need to learn how to compose these transformations to reflect the subproblem structure of the problems $\mathcal{P}$ without execution traces (Sec 3.2). Third, we need to learn the transformations themselves, but simultaneously learning both the primitives and the means of their composition is a difficult chicken-and-egg optimization problem because learning good transformations would make sense only if they are composed correctly, and vice versa (Sec 3.3).

3.1 Pattern-Matching over Problems via Analogy

If CRL is trained on a diverse distribution of composite problems that share enough structure, the hope is that it can distill out the individual structural components into specialized, modular *computational units*: atomic function operators that perform transformations between representations. Indeed, often it is easier to solve a problem P_a by making an analogy to a problem P_b that one already knows how to solve, rather than retraining to solve P_b from scratch (Schmidhuber, 2015; Simon, 1969). Transforming the type of input representation from t_{x_a} to t_{x_b} is one way of making such analogies. Therefore, if a learner hasn't ever translated French to Spanish but knows how to translate French to English and English to Spanish, it can reduce the French to Spanish problem to a English to Spanish problem by transforming the French input to English input, which it knows how to compute with. If a learner hasn't ever seen a 10-length arithmetic expression before, but it knows how to reduce an n-length expression to an $(n-1)$-length one, then it can iteratively reduce the 10-length expression down to a length it knows. However, spontaneous specialization does not come for free. If we constrain the representational vocabulary and draw the boundaries of modularity via transformations between representations, we can encourage specialization to naturally emerge when the computational units do not know the global problem they are solving but can make local progress to iteratively re-represent the problem into a more familiar form. However, when there are shared subproblems across the composite family, we would hope CRL would learn reusable computational units that capture that subproblem structure in the family.

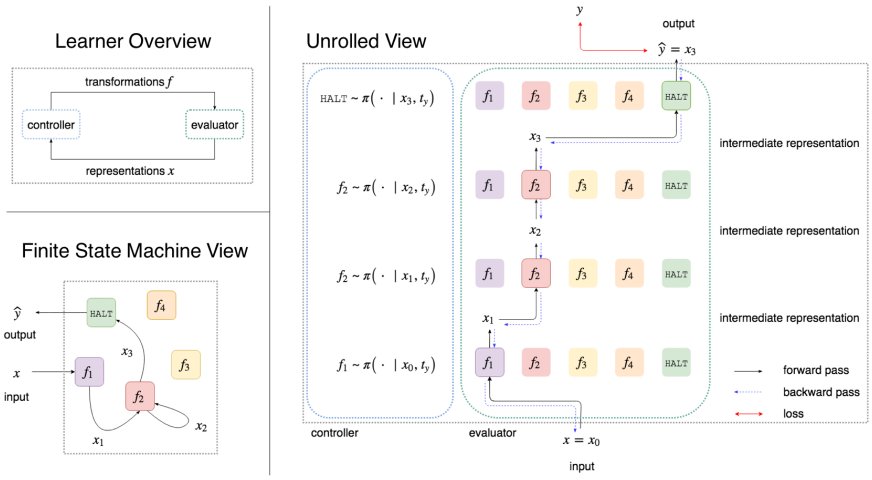


Figure 1: **Compositional recursive learner (CRL):** *top-left*: CRL is a cycle between a controller and evaluator: the controller selects a function f given an intermediate representation x and the evaluator applies f on x to create a new representation. *bottom-left*: CRL learns dynamically learns the structure of a program customized for its problem, and this program can be viewed as a finite state machine. *right*: A series of computations in the program is equivalent to a traversal through the Meta-MDP, where functions can be reused across different stages of computation, allowing for recursive computation.

3.2 Learning the Structure of a Computation as a Sequential Decision-Making Process

A transformation between representations can be generalized as any computation which changes the state of a finite state machine to another. Learning how to apply computational units in sequence can then be formulated as a sequential decision-making problem, where the state space is the intermediate results produced by a program and the action space is the set of computations.

This kind of sequential decision problem can be formalized as a meta-level Markov decision process (MDP) (Hay et al., 2014), defined by a tuple $(\mathcal{X}, \mathcal{F}, \mathcal{P}_{meta}, r, \gamma)$. $\mathcal{X}$ is the set of information states (intermediate results of computation), $\mathcal{C}$ is the set of computations, $\mathcal{P}_{meta}(x_j, f_j, x_{j+1})$ is the transition model that expresses the probability that at step j the computation f_j will change the information state from x_j to x_{j+1}, γ is a discount factor. The goal of CRL is to select a series of computations f to iteratively transform the input x into its predicted output $\hat{y}$. When CRL selects a special computation, the HALT signal, the current result is produced as output. The learner incurs a cost for every computation it executes and receives a terminal reward that reflects how its output $\hat{y}$ matches the desired output y. The computation cost and the HALT signal differentiate the meta-level MDP from a generic MDP, requiring CRL to balance program complexity and performance. The result is a program composed of a sequence of computations that customizes its complexity to the problem.

3.3 Learning the Transformations and their Means of Composition from Sparse Rewards

The meta-level MDP describes how to choose transformations, but not how to learn the form of the transformations themselves. We solve this problem by learning differentiable programs composed of neural networks, in which both the transformations and their composition are learned.

The implementation consists of a controller $\pi(f|x, t_y)$, a set of functions $f_k \in \mathcal{F}$, and an evaluator (Fig. 1). At step j, the controller observes the intermediate state of computation x_j and the target output type t_y. It selects a function f_k and a portion of x_j to operate on. The evaluator applies f_k to transform x_j into a new state of computation x_{j+1}. When π selects HALT, a loss is computed by comparing the current state of computation with the desired output. The loss is backpropagated through the functions, which are trained with Adam (Kingma & Ba, 2014). The controller receives a sparse reward derived from the loss, incurs a cost for each computation, and is trained with proximal policy optimization (Schulman et al., 2017).

The state space (excluding the initial state, which is CRL's input) consists of the outputs of the functions, the action space consists of the functions themselves, and the transition model is the evaluator. Unlike standard reinforcement learning problems, here the state space and action space can vary in dimensionality, as we show in our experiments. Moreover, as CRL trains, the function parameters change, so transition dynamics are non-stationary. To overcome these optimization difficulties, we draw from (Dechter et al., 2013) and gradually grow the complexity of the problems during training such that learning more complex compositions are bootstrapped from transformations learned in the context of simple compositions earlier on (see Appx. C for intuition). Because the outputs of a function are the internal representations of the larger neural network, CRL simultaneously designs its own internal representation language and the transformations that convert between them. CRL's states of computation are its internal representations, which may or may not correspond with the external representation given by the problem.

3.4 Discussion of Design Choices

Modularity and reuse of computational units are natural consequences of this framework. Such weight-sharing enables CRL to maximize knowledge it learns within and across problems under low data regimes. The computation graph that CRL constructs by its choices over functions is flexible enough to learn to reflect the true underlying compositional structure within a problem (although it need not to), thereby allowing CRL to make analogies transforming a larger problem into subproblems it knows how to solve. We need only a single controller that can learn to compose functions for a wide variety of problems; choosing when to terminate allows for metareasoning, extrapolation, and learning in an online setting where problems increasingly grow in complexity.

Our framework provides a language for describing a learner that designs itself: making the connection between programs, neural networks, and MDPs establishes the equivalence between computational units, functions, and representation transformations and opens opportunities to borrow tools from reinforcement learning and deep learning as a practical means for solving both the continuous and discrete optimization problems of respectively learning the transformations and their means composition. By explicitly attempting to discover and exploit the compositional structure of composite problems, we transform the generalization problem into a problem of learning algorithmic procedures over transformations between representations.

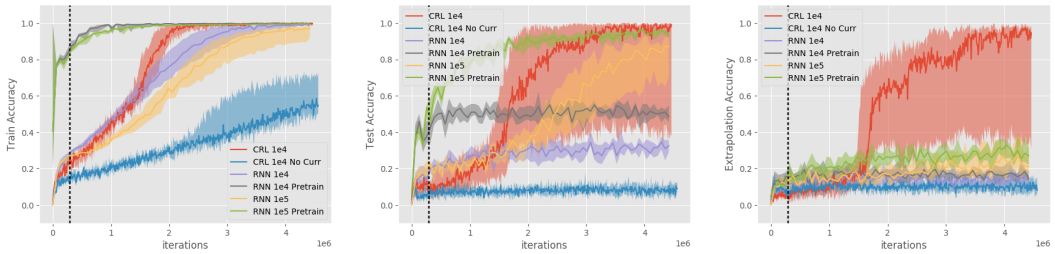
4 Experiments

The following experiments explore the compositional learning capabilities of CRL. In the highly structured domain of multilingual arithmetic, CRL discovers primitive modules that capture primitive operations and composes these modules to extrapolate to a combinatorially larger space of problems it has never seen. In underconstrained domains such as compositional image transformations, CRL learns to compose imperfect knowledge of primitive transformations to iteratively re-represent transformed MNIST digits into canonical ones.

4.1 Multilingual Arithmetic

Multilingual arithmetic problems is a domain that exactly tests a learner's capability to exploit compositionality to solve problems beyond its training distribuition. Arithmetic expressions come in m different *types*, each corresponding to a language that expresses it. An example input is `three plus four times seven` (the *source* language is English) and an example corresponding output is `uno` (the *target* language is Spanish). We arbitrarily chose $m = 5$ languages: English, Numerals, PigLatin, Reversed-English, Spanish. During training, each source language is seen with four target languages (and one held out for testing) and each target language is seen with four source languages (and one held out for testing). If a learner successfuly captures the compositional structure of these problems, it should automatically (1) solve problems of signficantly longer length by recursively reducing these problems to smaller instances(2) solve problems with unseen different language pairs by translating between languages.

The input is a tuple $(x^{(s)}, t_y)$, where x is the arithmetic expression expressed in source language s, and t_y is the language with which the learner should output the answer. The learner trains on a curriculum of a limited set of 2, 3, 4, 5-length expressions ($2.76 \cdot 10^4$ of the training distribution),



(a) Training Accuracy (b) Test Accuracy (c) Extrapolation Accuracy

Figure 2: **Multilingual math task.** CRL generalizes and extrapolates out of distribution significantly better than the RNN, which, even with ten times more data, does not extrapolate to 10-length multilingual arithmetic expressions. Even pretraining the RNN on domain-specific auxiliary tasks does not help it generalize or extrapolate, highlighting a limitation of using monolithic architectures for compositional problems. By comparing CRL with a version trained without a curriculum ("No Curr": blue), we see the benefit of slowly growing the complexity of problems throughout training, although this benefit does not transfer to the RNN. The vertical black dashed line indicates at which point all the training data has been added when CRL is trained with a curriculum (red). Note that generalization becomes apparent only after a million iterations after all the training data has been added. For (b, c) we only show accuracy on the expressions with the maximum length of those added so far to the curriculum. "1e4" and "1e5" correspond to the order of magnitude of the number of samples in the dataset, of which 70% are used for training. The dataset size is an order of magnitude larger than that of the numerical arithmetic experiment because it includes 25 language pairs. 10, 50, and 90 percentiles are shown over 6 runs.

and is asked to generalize to 5-length expressions (test set) extrapolate to 10-length expressions (extrapolation set) with unseen language pairs (problem space: $4.92 \cdot 10^{15}$ problems; see Appx. A).

CRL begins with several randomly initialized *reducers* and *translators*, subject to the constraints that a reducer transforms a window of three terms (selected by the conroller) into a softmax distribution over the vocabulary, and a translator applies the same function to every element of a length-k sequence to produce a length-k sequence of softmax distributions. The controller has access to t_y at every step of computation, but the functions do not, forcing the functions to learn behaviors agnostic of the language pair and putting the burden of choosing how to generalizing to different input-output types on the controller. CRL is compared with an RNN architecture (Chung et al., 2014) directly trained to map input to output, where t_y is concatenated with every element of $x^{(s)}$.

Results: With 10 times more data the RNN (yellow) generalizes to the 5-terms test set but *fails to extrapolate to the 10-terms dataset.* This is an intriguing result that suggests that with enough data the RNN's distributed representation captures some compositionality in the language pairs but cannot extrapolate to a larger problem space than which it has been trained. In contrast, CRL (red in Fig. 2) can extrapolate consistently better because, by design, it can solve the longer expressions by composing its learned primitives more times.

It is not just domain-specific knowledge: Although they are all learned, the existence of CRL's attention mechanism and the topology of its reducers and translators may give it a domain-specific advantage over the RNN. However, an RNN that was pre-trained to 100% accuracy on domain-specific auxiliary tasks (translating between numerical math to all other languages and of solving 3-term numerical math problems) still cannot extrapolate with 10 times more data, even though these auxiliary tasks give the RNN a nontrivial boost on the test set. This experiment highlights a fundamental gap in knowledge representation between a learner endowed with an inductive bias for compositionality and traditional monolithic function approximators, echoed in Fodor & Pylyshyn (1988); Lake & Baroni (2017).

Qualitative Analysis

Though the randomly initialized functions initially have no semantic meaning, as training progresses the functions gradually specialize such that after convergence the controller chooses a particular ordering of the functions given a new input expression. The controller gradually learned windows centered around operators (e.g. $2 + 3$ rather than $\times 4-$), suggesting that it has discovered semantic role of these primitive two-term expressions by pattern-matching common structure across arithmetic expressions of different lengths. Fig. 3 elucidates how CRL makes analogies to re-represent

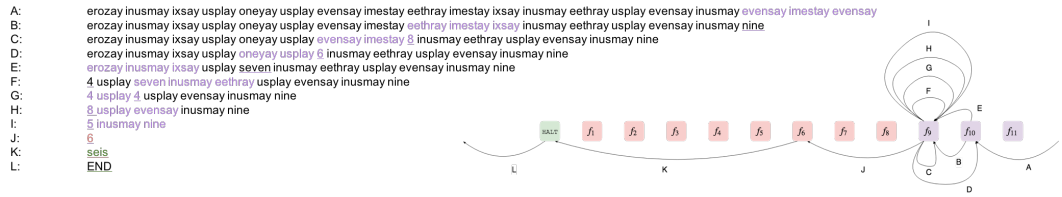


Figure 3: **Qualitative Analysis:** A visualization of a randomly selected execution trace from the extrapolation set. The input is the math expression $0 - 6 + 1 + 7 \times 3 \times 6 - 3 + 7 - 7 \times 7$ expressed in Pig Latin. The desired output is *seis*, which is the value of the expression, 6, expressed in Spanish. Each step of computation is labeled A - L, which is reflected in the finite state machine view on the right. The purple functions are reducers and the pink functions are translators. The input to a function is highlighted and the output of the function is underlined. The controller learns order of operations. The reducer f_9 always reduces to numerals and reducer f_{10} always reduces to English terms. This is interesting: because the functions do not know what the target language is, they learn to specialize for a specific purpose, resulting in hybrid arithmetic expressions made of different languages, the learners own *internal* language. Note also that the learner exhibits *metareasoning*, or reasoning about its own computation: it has never seen (Pig Latin, Spanish) language pair before during training, but it has seen (Pig Latin, Numerals) and (Numerals, Spanish), so it first reduces the Pig Latin expression to a numerical evaluation, and then translates that to its Spanish representation using the translator f_6. Note that all of this computation is happening internally to the learner, which computes on softmax distributions over the vocabulary; for visualization we show the token of the distribution with maximum probability.

an unseen problem into one that is more unfamiliar. Because the language pair is one it has not seen, the controller learns to route through expressions it does know how to work with to solve a new problem that it has never seen before. CRL learns reducers that reduce to a particular language from any language, which result in intermediate hybrid-language expressions. Notably, after it has reduced the entire expression, it takes an *additional* step to translate the final term to a term in the target language, even though the term before translation is not in either the source or the target language. Indeed, this shows that the controller reasons about its own computation, taking additional steps that it deems necessary to compose the primitives (the translators and reducers) it had learned to solve unseen problems (although it sometimes tries to `HALT` unnecessarily: see Appx. D.5). Rather than learn translators and reducers that are specific to single input and single output language pair as we had expected, the functions tended to learn operations specific to the output language only, possibly due to their nonlinear nature.

4.2 Image Transformations

This section explores how CRL may be applied to significantly less structured domains. The task is to classify an MNIST digit, where the MNIST digit has been randomly translated, rotated, and resized. Suppose CRL has knowledge of what untransformed MNIST digits look like; is it possible that CRL can learn to compose appropriate spatial affine transformations in sequence to convert the transformed MNIST digit into a "canonical" one, such that it can use a pre-trained classifier to classify it? To reformulate a scenario to one that is more familar is common in the real world: humans view an object at different angles yet understand it is the same object; they may have an accustomed route to work, but can adapt to a detour if the route is blocked. To be able to re-represent unfamiliar situations into familiar ones by composing primitive operations that capture the axes of variation in a domain is a benefit of a learner endowed with compositionality.

In Fig. 4, CRL is trained on a distribution of MNIST digits that were first modified by either one transformation or two transformations. CRL is initalized with four modules: a Spatial Transformer Network (STN) (Jaderberg et al., 2015) parametrized to only rotate (action 0), an STN that only scales (action 1), an STN that only translates (action 2), and an identity function (action 3).

The bottom row of Fig. 4 is on an unseen dataset of MNIST images generated by composing three transformations (one more than CRL has ever seen). In this transfer setting, we tested whether it is possible to freeze the modules learned from seeing only one or two transformations, and only re-learning the controller on the dataset with three transformations. Quantitatively, CRL averages about 70% accuracy, oscillating between 60% and 80% accuracy (as opposed to about 85%, oscillating between 80% and 90% in the one-two transformation set). One possible reason for a poorer performance on the transfer set can be because freezing the modules limits the ability of CRL to im-

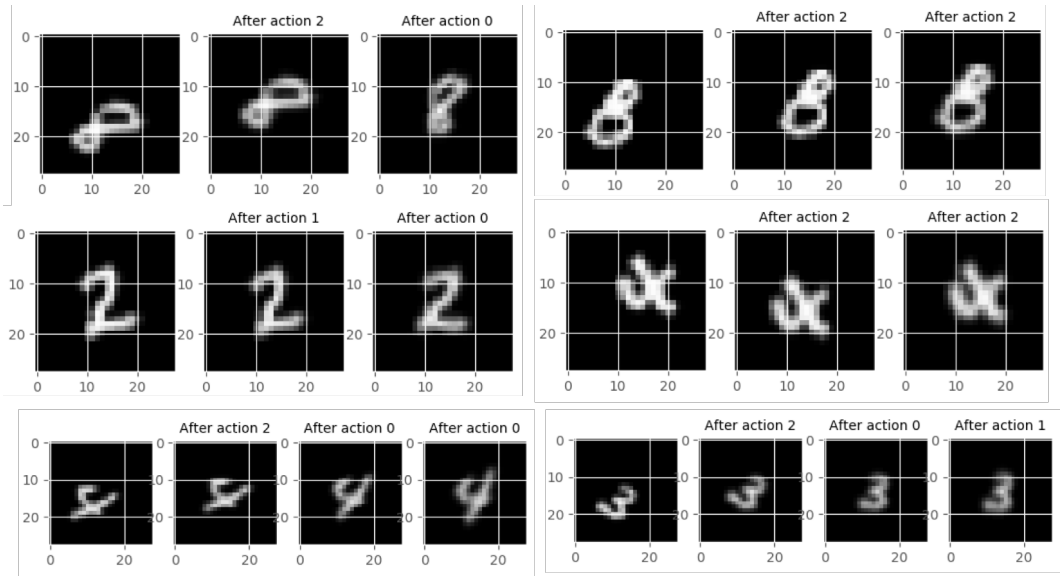


Figure 4: **Image Transformations:** The leftmost images in each quadrant is the input. CRL was trained on images modified by one or two transformations. All images are on held-out digits; the top two rows show composition of transformations from the same distribution as those seen during training, and the bottom row was generated with three transformations, unseen during training. For simplicity CRL was constrained to take exactly two steps in the top two rows and exactly three steps in the bottom row. *top-left*: CRL first translates, then rotates. *middle-left*: CRL first shrinks, then rotates. *top-right:* CRL translates, then translates again. Note that this sequence of transformations did not occur in the data generation process. *middle-right:* A suboptimal case where CRL did not rotate as one would expect it to, but instead composes two translations. *bottom-left:* CRL translates, then rotates twice. *bottom-right:* CRL translates, then rotates, then enlarges.

prove: as a result, if the modules are imperfect and do not capture precisely the transformations in the underlying generative process of the data, the controller is limited in what it can do. Furthermore, this domain is signficantly more challenging than the mulilingual arithmetic domain because the underlying compositional structure of the data is latent: the learner only receives pixels as input, so to discover that there was a sequence of transformations that generated its input is nontrivial. Lastly, this domain is high dimensional, which poses several challenges for preventing degeneracies. Unless more constraints are applied to the funcitons and the intermediate representations, CRL could easily fall into a trap of only choosing one module (since all modules take in pixels and produce pixels), and get stuck in suboptimal local optima. We plan to investigate better ways of stabilizing and regularizating the training for future work.

5 Related Work

Reformulation, or re-representing a problem to make it easier to solve, dates back at least to the 1960s (Holte & Choueiry, 2003; Simon, 1969; Anderson, 1990); CRL is an initial step towards bridging this idea with deep learning. Indeed for generalizing to multiple different tasks, CRL departs from the metalearning framework (Thrun & Pratt, 2012; Schmidhuber, 1987), which seeks to learn a learner that can learn quickly on the new problem, and takes a metareasoning approach (Russell & Wefald, 1991; Hay et al., 2014), where the learner reasons about its own knowledge and computations to solve the new problem. Instead of learning an algorithm to learn, the learner can learn an algorithm to reason: casting the generalization problem as an algorithm design problem is a hypothesis for the mechanism with which humans can generalize to new scenarios (Griffiths et al., 2015; Callaway et al., 2017; Lieder et al., 2017). Our work is one of a growing community (Hamrick et al., 2017; Graves, 2016; Oh et al., 2017) linking metareasoning with deep learning.

Compositionality is a major motivation for neural program induction, whose work generally falls into two categories. The first assumes pre-specified transformations and learns the structure (from dense supervision on execution traces to sparse-rewards) (Reed & De Freitas, 2015; Cai et al., 2017; Xu et al., 2017; Chen et al., 2017; Ganin et al., 2018; Bunel et al., 2018; Feser et al., 2016; Džeroski

et al., 2001; Zaremba et al., 2016). The second learns the transformations but pre-specifies the structure (Devin et al., 2017; Andreas et al., 2016; Riedel et al., 2016). The closest work in this respect to ours is (Gaunt et al., 2016) which assumes a differentiable interpreter grounded in source code; we take a different perspective of learning algorithms through re-representations, which enables our method to adapt on-line to new problem classes. Work that has explored compositionality in the input space include (Schlag & Schmidhuber, 2017; Battaglia et al., 2018).

Research on self-organizing learners (Hinton et al., 2018; Rosenbaum et al., 2018; Fernando et al., 2017) have begun to examine how to *route* through an fixed architecture to encapsulate computation, but the hard limitation that prevents fixed architectures from scaling to problems with complexity beyond that which they were trained is their limited number of computation steps. Parascandolo et al. (2017) focuses on a complementary direction to ours; whereas the focus on learning causal mechanisms for a single step, we focus on learning how to compose the modules.

6 Discussion

We have considered learning to generalize and extrapolate with limited data to harder compositional problems than a learner has previously seen. We have taken steps toward this challenge by presenting a characterization, algorithm, and implementation of a learner that programs itself automatically to reflect the structure of the problem it faces. Our key ideas are (1) transforming representations with modular units of computation is a potential solution for decomposing problems in a way that reflects their subproblem structure; (2) learning the structure of a computation can be formulated as a sequential decision-making problem; (3) tools from reinforcement learning and deep learning can serve as a practical means for solving both the continuous and discrete optimization problems of respectively learning the transformations and their means of composition with sparse rewards. Our experiments support the core perspective of this paper: for compositional problems under low data regimes, casting the generalization problem as a problem of learning algorithmic procedures over representation transformations offers significant advantages over monolithic architectures that only rely on the diversity in the data to shape a shared representation. This work brings together ideas from various areas of research – reformulation, metareasoning, program induction, modularity, self-organizing neural networks – to provide a proof-of-concept for learning compositional and recursive learners that design themselves and reason about their own computations.

References

Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2 edition, 1996. ISBN 0-262-51087-1.

John Robert Anderson. The adaptive character of thought. chapter 5, pp. 191–230. Psychology Press, 1990.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 39–48, 2016.

Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. *arXiv preprint arXiv:1704.06611*, 2017.

Frederick Callaway, Falk Lieder, Paul Krueger, and Thomas L Griffiths. Mouselab-mdp: A new paradigm for tracing how people plan. 2017.

Xinyun Chen, Chang Liu, and Dawn Song. Learning neural programs to parse programs. *arXiv preprint arXiv:1706.01284*, 2017.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Eyal Dechter, Jonathan Malmaud, Ryan P Adams, and Joshua B Tenenbaum. Bootstrap learning via modular concept discovery. In *IJCAI*, pp. 1302–1309, 2013.

Coline Devin, Abhishek Gupta, Trevor Darrell, Pieter Abbeel, and Sergey Levine. Learning modular neural network policies for multi-task and multi-robot transfer. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pp. 2169–2176. IEEE, 2017.

Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine learning*, 43(1-2):7–52, 2001.

SM Ali Eslami, Nicolas Heess, Theophane Weber, Yuval Tassa, David Szepesvari, Geoffrey E Hinton, et al. Attend, infer, repeat: Fast scene understanding with generative models. In *Advances in Neural Information Processing Systems*, pp. 3225–3233, 2016.

Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

John K Feser, Marc Brockschmidt, Alexander L Gaunt, and Daniel Tarlow. Differentiable functional program interpreters. *arXiv preprint arXiv:1611.01988*, 2016.

Jerry A Fodor and Zenon W Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1-2):3–71, 1988.

Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S.M. Ali Eslami, and Orial Vinyals. Synthesizing programs for images using reinforced adversarial learning. *arXiv preprint arXiv:1804.01118*, 2018.

Alexander L Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. *arXiv preprint arXiv:1611.02109*, 2016.

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Thomas L Griffiths, Falk Lieder, and Noah D Goodman. Rational use of cognitive resources: Levels of analysis between the computational and the algorithmic. *Topics in cognitive science*, 7(2): 217–229, 2015.

Jessica B Hamrick, Andrew J Ballard, Razvan Pascanu, Oriol Vinyals, Nicolas Heess, and Peter W Battaglia. Metacontrol for adaptive imagination-based optimization. *arXiv preprint arXiv:1705.02670*, 2017.

Nicholas Hay, Stuart Russell, David Tolpin, and Solomon Eyal Shimony. Selecting computations: Theory and applications. *arXiv preprint arXiv:1408.2048*, 2014.

Geoffrey Hinton, Nicholas Frosst, and Sara Sabour. Matrix capsules with em routing. 2018.

Robert C Holte and Berthe Y Choueiry. Abstraction and reformulation in artificial intelligence. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, 358(1435): 1197–1204, 2003.

Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. In *Advances in neural information processing systems*, pp. 2017–2025, 2015.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

BM Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. *arXiv preprint arXiv:1711.00350*, 2017.

Falk Lieder, Frederick Callaway, Sayan Gul, Paul M Krueger, and Thomas L Griffiths. Learning to select computations. *arXiv preprint arXiv:1711.06892*, 2017.

Volodymyr Mnih, Nicolas Heess, Alex Graves, et al. Recurrent models of visual attention. In *Advances in neural information processing systems*, pp. 2204–2212, 2014.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

Junhyuk Oh, Satinder Singh, Honglak Lee, and Pushmeet Kohli. Zero-shot task generalization with multi-task deep reinforcement learning. *arXiv preprint arXiv:1706.05064*, 2017.

Giambattista Parascandolo, Mateo Rojas-Carulla, Niki Kilbertus, and Bernhard Schölkopf. Learning independent causal mechanisms. *arXiv preprint arXiv:1712.00961*, 2017.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Sebastian Riedel, Matko Bosnjak, and Tim Rocktäschel. Programming with a differentiable forth interpreter. *CoRR, abs/1605.06640*, 2016.

Clemens Rosenbaum, Tim Klinger, and Matthew Riemer. Routing networks: Adaptive selection of non-linear functions for multi-task learning. *International Conference on Learning Representations*, 2018.

Stuart Russell and Eric Wefald. Principles of metareasoning. *Artificial intelligence*, 49(1-3):361–395, 1991.

Imanol Schlag and Jürgen Schmidhuber. Gated fast weights for on-the-fly neural program generation. In *NIPS Metalearning Workshop*, 2017.

Jürgen Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987.

Jürgen Schmidhuber. On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models. *arXiv preprint arXiv:1511.09249*, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Herbert A Simon. The sciences of the artificial. *Cambridge, MA*, pp. 132, 1969.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pp. 3104–3112, 2014.

Sebastian Thrun and Lorien Pratt. *Learning to learn*. Springer Science & Business Media, 2012.

Danfei Xu, Suraj Nair, Yuke Zhu, Julian Gao, Animesh Garg, Li Fei-Fei, and Silvio Savarese. Neural task programming: Learning to generalize across hierarchical tasks. *arXiv preprint arXiv:1710.01813*, 2017.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pp. 2048–2057, 2015.

Wojciech Zaremba, Tomas Mikolov, Armand Joulin, and Rob Fergus. Learning simple algorithms from examples. In *International Conference on Machine Learning*, pp. 421–429, 2016.

A DATA

Numerical math: The dataset contains arithmetic expressions of k terms where the terms are integers $\in [0, 9]$ and the operators are $\in \{+, \times, -\}$. The number of possible problems is $(10^k)(3^{k-1})$. The learner sees $5810/(2.04 \cdot 10^{14}) = 2.85 \cdot 10^{-11}$ of the training distribution. The number of possible problems in the extrapolation set is $(10^{20})(3^{19}) = 1.16 \cdot 10^{29}$. An input expression is a sequence of one-hot vectors of size 13.

# Terms	Prob. Space	# Train Samples	Frac. of Prob. Space
2	$(10^2)(3^1) = 3 \cdot 10^2$	210	$7 \cdot 10^{-1}$
3	$(10^3)(3^2) = 9 \cdot 10^3$	700	$7.78 \cdot 10^{-2}$
4	$(10^4)(3^3) = 2.7 \cdot 10^5$	700	$2.6 \cdot 10^{-3}$
5	$(10^5)(3^4) = 8.1 \cdot 10^6$	700	$8.64 \cdot 10^{-5}$
6	$(10^6)(3^5) = 2.43 \cdot 10^8$	700	$2.88 \cdot 10^{-6}$
7	$(10^7)(3^6) = 7.29 \cdot 10^9$	700	$9.60 \cdot 10^{-8}$
8	$(10^8)(3^7) = 2.19 \cdot 10^{11}$	700	$3.20 \cdot 10^{-9}$
9	$(10^9)(3^8) = 6.56 \cdot 10^{12}$	700	$1.07 \cdot 10^{-10}$
10	$(10^{10})(3^9) = 1.97 \cdot 10^{14}$	700	$3.56 \cdot 10^{-12}$
Total	$2.04 \cdot 10^{14}$	5810	$2.85 \cdot 10^{-11}$

Table 1: Numerical Arithmetic Dataset

Multilingual math: The dataset contains arithmetic expressions of k terms where the terms are integers $\in [0, 9]$ and the operators are $\in \{+, \cdot, -\}$, expressed in five different languages. With 5 choices for the source language and target language, the number of possible problems is $(10^k)(3^{k-1})(5^2)$. In training, each source language is seen with 4 target languages and each target language is seen with 4 source languages: 20 pairs are seen in training and 5 pairs are held out for testing. The learner sees $46200/(1.68 \cdot 10^8) = 2.76 \cdot 10^{-4}$ of the training distribution. The entire space of possible problems in the extrapolation set is $(10^{10})(3^9)(5^2) = 4.92 \cdot 10^{15}$ out of which we draw samples from the 5 held-out language pairs $\big((10^{10})(3^9)(5) = 9.84 \cdot 10^{14}$ possible$\big)$. An input expression is a sequence of one-hot vectors of size $13 \times 5 + 1 = 66$ where the single additional element is a `STOP` token (for training the RNN).

# Terms	Prob. Space	Train Prob. Space	# Train Samples	Frac. of Train Dist.	Frac. of Prob. Space
2	$(10^2)(3^1)(25) = 7.5 \cdot 10^3$	$(10^2)(3^1)(20) = 6 \cdot 10^3$	$210 \cdot 20 = 4.2 \cdot 10^3$	$7 \cdot 10^{-1}$	$5.6 \cdot 10^{-1}$
3	$(10^3)(3^2)(25) = 2.25 \cdot 10^5$	$(10^3)(3^2)(20) = 1.8 \cdot 10^5$	$700 \cdot 20 = 1.4 \cdot 10^4$	$7.78 \cdot 10^{-2}$	$6.22 \cdot 10^{-2}$
4	$(10^4)(3^3)(25) = 6.75 \cdot 10^6$	$(10^4)(3^3)(20) = 5.4 \cdot 10^6$	$700 \cdot 20 = 1.4 \cdot 10^4$	$2.6 \cdot 10^{-3}$	$2.07 \cdot 10^{-3}$
5	$(10^5)(3^4)(25) = 2.02 \cdot 10^8$	$(10^5)(3^4)(20) = 1.62 \cdot 10^8$	$700 \cdot 20 = 1.4 \cdot 10^4$	$8.64 \cdot 10^{-5}$	$6.91 \cdot 10^{-5}$
Total	$2.09 \cdot 10^8$	$1.68 \cdot 10^8$	46200	$2.76 \cdot 10^{-4}$	$2.21 \cdot 10^{-4}$

Table 2: Multilingual Arithmetic Dataset

B LEARNER DETAILS

All models are implemented in PyTorch (Paszke et al., 2017) and we will make the code and data available online.

B.1 BASELINES

The RNN is implemented as a sequence-to-sequence (Sutskever et al., 2014) GRU. The routing networks (experimental comparison in Appx. D.2) implementation used the same hyperparameters as those for the best CRL model.

B.2 CRL

Controller: The controller consists of a policy network and a value function, each implemented as GRUs that read in the input expression. The value function outputs a value estimate for the current expression. For the numerical arithmetic task, the policy network first selects a reducer and then conditioned on that choice selects the location in the input expression to apply the reducer. For the

multilingual arithmetic task, the policy first samples whether to halt, reduce, or translate, and then conditioned on that choice (if it doesn't halt) it samples the reducer (along with an index to apply it) or translator. **Functions:** The reducers are initialized as a two-layer feedforward network with ReLU non-linearities (Nair & Hinton, 2010). The translators are a linear weight matrices.

C EXPERIMENT DETAILS

Training procedure: The training procedure for the controller follows the standard Proximal Policy Optimization training procedure, where the learner samples a set of episodes, pushes them to a replay buffer, and every k episodes updates the controller based on the episodes collected. Independently, every k' episodes we consolidate those k' episodes into a batch and use it to train the functions. We found via a grid search $k = 1024$ and $k' = 256$. Through an informal search whose heuristic was performance on the training set, we settled on updating the curriculum of CRL every 10^5 iterations and updating the curriculum of the RNN every $5 \cdot 10^4$ iterations.

Domain-specific nuances: In the case that `HALT` is called to early, CRL treats it as a no-op. Similarly, if a reduction operator is called when there is only one token in the expression, the learner also treats it as a no-op. There are other ways around this domain-specific nuance, such as to always halt whenever `HALT` is called but only do backpropagation from the loss if the expression has been fully reduced (otherwise it wouldn't make sense to compute a loss on an expression that has not been fully reduced). The way we interpret these "invalid actions" is analogous to a standard practice in reinforcement learning of keeping an agent in the same state if it walks into a wall of a maze.

Symmetry breaking: We believe that the random initialization of the functions and the controller breaks the symmetry between the functions. For episodes 0 through k the controller still has the same random initial weights, and for episodes 0 through k' the functions still have the same random initial weights. Because of the initial randomness, the initial controller will select certain functions more than others for certain inputs; similarly initially certain functions will perform better than others for certain inputs. Therefore, after k iterations, the controller's parameters will update in a direction that will make choosing the functions that luckily performed better for certain inputs more likely; similarly, after k' iterations, the functions' parameters will update in a direction that will make them better for the inputs they have been given. So gradually, functions that initially were slightly better at certain inputs will become more specialized towards those inputs and they will also get selected more for those inputs.

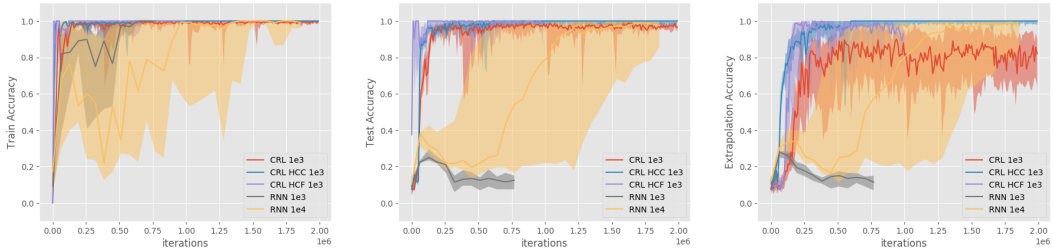
Training objective: The objective of the composition of functions is to minimize the negative log likelihood of the correct answer to the arithmetic problem. The objective of the controller is to maximize reward. It receives a reward of 1 if the token with maximum log likelihood is that of the correct answer, 0 if not, and -0.01 for every computation step it takes. The step penalty was found by a scale search over $\{-1, -0.1, -0.01, -0.001\}$ and -0.01 was a penalty that we found balanced accuracy and computation time to a reasonable degree during training. There is no explicit feedback on what the transformations should be and on how they are composed.

D ADDITIONAL EXPERIMENTS

D.1 NUMERICAL MATH

The input is a numerical arithmetic expression (e.g. $3 + 4 \times 7$) and the desired output (e.g. 1) is the evaluation of the expression modulo 10. The input and output have the same type. In our experiments we train on a curriculum of length-2 expressions to length-10 expressions, adding new expressions to an expanding dataset over the course of training in a lifelong learning fashion. The first challenge is to learn from this limited data (only 6510 training expressions) to generalize well to unseen length-10 expressions in the test set ($\approx 2^{14}$ possible). The second challenge is to extrapolate from this limited data to length-20 expressions ($\approx 10^{29}$ possible).

Expressions are represented as sequences of one-hot vectors. The functions ("reducers") used in CRL reduce a length k expression to a length $k - 3$ one: the controller chooses a window of 3 terms in the expression, and the reducer transforms that into a softmax distribution over a single term in the vocabulary. This process is repeated until the expression is reduced to one term, at which point



(a) Training Accuracy (b) Test Accuracy (c) Extrapolation Accuracy

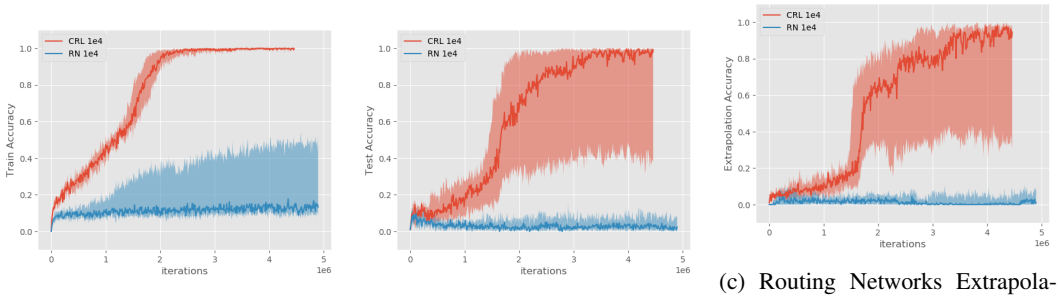
Figure 5: **Numerical math task.** We compare our learner with the RNN baseline. As a sanity check, we also compare with a version of our learner which has a hardcoded controller (HCC) and a learner which has hardcoded functions (HCF) (in which case the controller is restricted to select windows of 3 with an operator in the middle, but this is not the case in general). All models perform well on the training set. Only our method and its HCC, HCF modifications generalize to the testing and extrapolation set. The RNN requires 10 times more data to generalize to the testing and extrapolation set. For (b, c) we only show accuracy on the expressions with the maximum length of those added so far to the curriculum. "1e3" and "1e4" correspond to the order of magnitude of the number of samples in the dataset, of which 70% are used for training. 10, 50, and 90 percentiles are shown over 6 runs.

the expression persists until the controller `HALT`s. Choosing to attend to a subset of the input is not domain-specific; this strategy is applicable in many settings (Mnih et al., 2014; Eslami et al., 2016; Xu et al., 2015). We compare with an RNN architecture (Chung et al., 2014) directly trained to map input to output.

Results: Interestingly, though the RNN eventually generalizes to different 10-length expressions and extrapolates to 20-length expressions (yellow in Fig. 5) with 10 times more data as CRL, it completely overfits when given the same amount of data (gray). In contrast, CRL (red) does not overfit, generalizing significantly better to both the 10-length and 20-length test sets. We believe that the modular disentangled structure in CRL biases it to cleave the problem distribution at its joints, yielding this 10-fold reduction in sample complexity relative to the RNN.

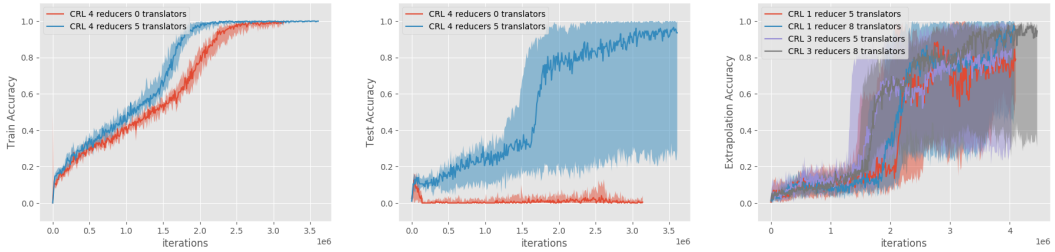
Indeed, we found that the controller naturally learned windows centered around operators (e.g. $2+3$ rather than $\times 4-$), suggesting that it has discovered semantic role of these primitive two-term expressions by pattern-matching common structure across arithmetic expressions of different lengths. Note that CRL's extrapolation accuracy here is not perfect compared to (Cai et al., 2017): but what we believe is noteworthy is that CRL achieves such high extrapolation accuracy with only sparse supervision, *without* the step-by-step supervision on execution traces, the stack-based model of execution, and hardcoded transformations.

D.2 Ablation Study



(a) Routing Networks Train (b) Routing Networks Test (c) Routing Networks Extrapolation

Figure 6: **Routing Networks:** Routing Networks take significantly more iterations to train, showing that routing alone may not necessarily be enough for generalizing to new problems: reusing computational units *within* a computation provides a significant benefit. 10, 50, and 90 percentiles are shown over 6 runs.



(a) Pathological Train (b) Pathological Test (c) Different numbers of functions

Figure 7: **Variations:** The minimum number of reducers and translators that can solve the multilingual math problems is 1 and m respectively, where m is the number of languages. This is on an extrapolation task, which has more terms *and* different language pairs. (a, b): Four reducers and zero translators (red) is a pathological choice of functions that causes CRL to overfit, but it does not when translators are provided. (c) In the non-pathological cases, regardless of the number of functions, the learner metareasons about the resources it has to customize its computation to the problem. 10, 50, and 90 percentiles are shown over 6 runs.

We compare our approach to Routing Networks (RN) (Rosenbaum et al., 2018), which we can implement as an ablation of our method by fixing the number of computation steps and using a different controller per target language and per computation step. Figure 6 compares CRL with RN in the multilingual math domain. We observe that the assumptions that Routing Networks make for multitask classification do not necessarily hold for general compositional problems. For example, because RN uses a different set of functions per computation step, it is unable to exploit compositional structure to reuse its learned functions for simpler versions of the same problem. The number of parameters also scales by a linear factor of the number of computation steps. Even with functional reuse across steps, a finite computation horizon limits the method from scaling to problems of variable length, or take extra steps to route through what it does know to figure out how to solve what it does not know, which our method does. Comparing with RN suggests that routing alone is not sufficient for finding the hierarchical decomposition of a problem into its subproblems: a crucial piece is the *reuse* of learned primitive transformations. We have also improved upon RN by showing that it is possible to train a single controller as a function approximator across multiple tasks.

D.3 Variations

Here we study the effect of varying the number of functions available to our learner. Fig. 7a, 7b highlights a particular pathological choice of functions that causes CRL to overfit. If CRL uses four reducers and zero translators (red), it is not surprising that it fails to generalize to the test set: recall that each source language is only seen with four target languages during training with one held out; each reducer can just learn to reduce to one of the four target languages. What is interesting though is that when we add five translators to the four reducers (blue), we see certain runs achieve 100% generalization, even though CRL need not use the translators at all in order to fit the training set. That the blue training curve is slightly faster than the red offers a possible explanation: it may be harder to find a program where each reducer can reduce any source language to their specialized target language, and easier to find programs that involve steps of re-representation (through these translators), where the solution to a new problem is found merely by re-representing that problem into a problem that learner is more familiar with. The four-reducers-five-translators could have overfitted completely like the four-reducers-zero-translators case, but it consistently does not.

Indeed, we find that when we vary the number of reducers in $\{1, 3\}$ and the number of translators in $\{5, 8\}$ in Fig. 7c, the extrapolation performance is consistent across the choices of different numbers of functions, suggesting that CRL is quite robust to the number of functions in non-pathological cases.

D.4 How far can we push extrapolation?

The extrapolation accuracy from 6 to 100 terms after training on a curriculum from 2 to 5 terms (46200 examples) on the multilingual arithmetic task. The number of possible 100-term problems

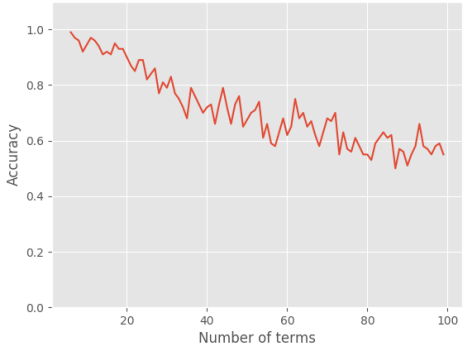


Figure 8: Multilingual Extrapolation

is $(10^{100})(3^{99})(5^2) = 4.29 \cdot 10^{148}$ and CRL achieves about 60% accuracy on these problems; a random guess would be 10%.

D.5 Execution Traces: Examples

Here are two randomly selected execution traces from the numerical arithmetic extrapolation task (train on 10 terms, extrapolate to 20 terms), where CRL's accuracy hovers around 80%. These expressions are derived from the internal representations of CRL, which are softmax distributions over the vocabulary (except for the first expression, which is one-hot because it is the input). The expressions here show the maximum value for each internal representation.

This is a successful execution. The input is `6*1*3-4+6*0*0+1-7-3+3+3*4+1+1+3+3+6+2+7` and the correct answer is `3`. Notice that the order in which controller applies its functions does not strictly follow the order of operations but respects the rules of order of operations: for example, it may decide to perform addition (A) before multiplication (B) if it doesn't affect the final answer.

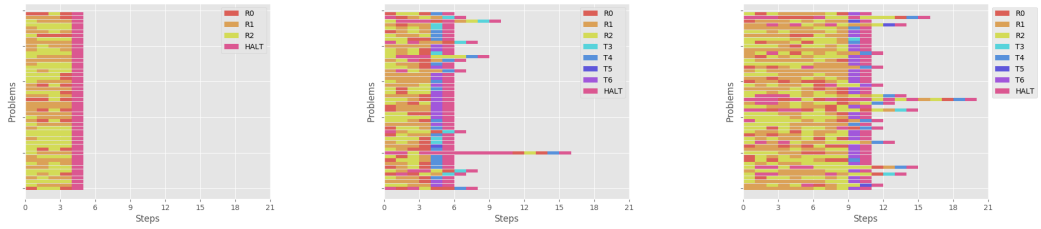
```
6*1*3-4+6*0*0+1-7-3+3+3*4+1+1+3+3+6+2+7        # 3 * 4 = 2
6*1*3-4+6*0*0+1-7-3+3+2+1+1+3+3+6+2+7          # 3 + 2 = 5
6*1*3-4+6*0*0+1-7-3+5+1+1+3+3+6+2+7            # 1 - 7 = 4
6*1*3-4+6*0*0+4-3+5+1+1+3+3+6+2+7              # 0 * 0 = 0
6*1*3-4+6*0+4-3+5+1+1+3+3+6+2+7                # 4 - 3 = 1
6*1*3-4+6*0+1+5+1+1+3+3+6+2+7                  # 1 + 3 = 4
6*1*3-4+6*0+1+5+1+4+3+6+2+7                    # 5 + 1 = 6
6*1*3-4+6*0+1+6+4+3+6+2+7                      # 1 + 6 = 7
6*1*3-4+6*0+7+4+3+6+2+7                        # 2 + 7 = 9
6*1*3-4+6*0+7+4+3+6+9                          # 3 + 6 = 9
6*1*3-4+6*0+7+4+9+9                            # 6 * 0 = 0                  ------------------------------------------
6*1*3-4+0+7+4+9+9                              # tried to HALT              everything above this line is extrapolation
6*1*3-4+0+7+4+9+9                              # 9 + 9 = 8                  (A)
6*1*3-4+0+7+4+8                                # 1 * 3 = 3                  (B)
6*3-4+0+7+4+8                                  # 0 + 7 = 7
6*3-4+7+4+8                                    # 6 * 3 = 8
8-4+7+4+8                                      # 8 - 4 = 4
4+7+4+8                                        # 4 + 7 = 1
1+4+8                                          # 1 + 4 = 5
5+8                                            # 5 + 8 = 3
3                                              # HALT
END
```

This is an unsuccessful execution trace. The input is `5+6-4+5*7*3*3*3*8*0*1-4+6-3*5*3+6-0+0-4-6` and the correct answer is `0`. Notice that it tends to follow of order of operations by doing multiplication first, although it does make mistakes (D), which in this case was the reason for its incorrect answer. Note that CRL never receives explicit feedback about its mistakes on what its functions learn to do or the order in which it applies them; it only receives a sparse reward signal at the very end. Although (C) was a calculation mistake, it turns out that it does not matter because the subexpression would be multiplied by 0 anyways.

```
5+6-4+5*7*3*3*8*0*1-4+6-3*5*3+6-0+0-4-6        # 3 * 8 = 4
5+6-4+5*7*3*4*0*1-4+6-3*5*3+6-0+0-4-6          # 0 - 4 = 6
```

```
5+6-4+5*7*3*4*0*1-4+6-3*5*3+6-0+6-6          # 5 * 7 = 5
5+6-4+5*3*4*0*1-4+6-3*5*3+6-0+6-6            # 3 * 4 = 4 (mistake)      (C)
5+6-4+5*4*0*1-4+6-3*5*3+6-0+6-6              # tried to HALT
5+6-4+5*4*0*1-4+6-3*5*3+6-0+6-6              # 5 * 4 = 0
5+6-4+0*0*1-4+6-3*5*3+6-0+6-6                # 6 - 6 = 0
5+6-4+0*0*1-4+6-3*5*3+6-0+0                  # 6 - 3 = 3                (D: order of operations mistake)
5+6-4+0*0*1-4+3*5*3+6-0+0                    # tried to HALT
5+6-4+0*0*1-4+3*5*3+6-0+0                    # tried to HALT
5+6-4+0*0*1-4+3*5*3+6-0+0                    # tried to HALT
5+6-4+0*0*1-4+3*5*3+6+0                      # 3 * 5 = 5
5+6-4+0*0*1-4+5*3+6+0                        # 0 * 1 = 0
5+6-4+0*1-4+5*3+6+0                          # 5 * 3 = 5                ------------------------------------------
5+6-4+0*1-4+5+6+0                            # 0 * 1 = 0                everything above this line is extrapolation
5+6-4+0-4+5+6+0                              # tried to HALT
5+6-4+0-4+5+6+0                              # tried to HALT
5+6-4+0-4+5+6+0                              # tried to HALT
5+6-4+0-4+5+6+0                              # tried to HALT
5+6-4+0-4+5+6+0                              # tried to HALT
5+6-4+0-4+5+6+0                              # tried to HALT
5+6-4+0-4+5+6+0                              # tried to HALT
5+6-4+0-4+5+6+0                              # 6 + 0 = 0
5+6-4+0-4+5+6                                # 5 + 6 = 1
5+6-4+0-4+1                                  # 0 - 4 = 6
5+6-4+6+1                                    # 5 + 6 = 1
1-4+6+1                                      # 1 - 4 = 7
7+6+1                                        # 7 + 6 = 3
3+1                                          # 3 + 1 = 4
4                                            # HALT
END
```

D.6 EXECUTION TRACES: FUNCTION SELECTION



(a) Validation Execution Traces (b) Test Execution Traces (c) Extrapolation Execution Traces

Figure 9: **Multilingual Arithmetic Execution Traces**

Fig. 9 compares the execution traces of CRL on the validation, test, and extrapolation sets after training. The language pairs in the test and extrapolation sets are the same, and the language pairs among training, validation, and test are disjoint. The functions beginning with R are reducers, and the functions beginning with T are translators. CRL takes an additional step to translate during the test and extrapolation execution traces. The traces that are significantly longer are because CRL tries to HALT too early and wastes computation steps.