Synthesis of Differentiable Functional Programs for Lifelong Learning

Lazar Valkov [1] **Dipak Chaudhari** [2] **Akash Srivastava** [1] **Charles Sutton** [1] **Swarat Chaudhuri** [2]

Abstract

We present a *neurosymbolic approach* to the lifelong learning of algorithmic tasks that mix perception and procedural reasoning. Reusing high-level concepts across domains and learning complex procedures are two key challenges in lifelong learning. We show that a combination of gradient-based learning and symbolic program synthesis can be a more effective response to these challenges than purely neural methods. Concretely, our approach, called HOUDINI, represents neural networks as strongly typed, end-to-end differentiable functional programs that use symbolic higher-order combinators to compose a library of neural functions. Our learning algorithm consists of: (1) a program synthesizer that performs a type-directed search over programs in this language, and decides on the library functions that should be reused and the architectures that should be used to combine them; and (2) a neural module that trains synthesized programs using stochastic gradient descent. We evaluate our approach on three algorithmic tasks. Our experiments show that our type-directed search technique is able to significantly prune the search space of programs, and that the overall approach transfers high-level concepts more effectively than monolithic neural networks as well as traditional transfer learning.

1. Introduction

Representation learning for real-world tasks requires not only the ability to solve perceptual tasks such as object recognition, but also *procedural knowledge*, knowledge about how to do things, which can be complex yet difficult to articulate explicitly. The goal of building systems that learn procedural knowledge has motivated a recent body of work on neural methods for learning representations of algorithms (Graves et al., 2014; Kurach et al., 2015; Reed & de Freitas, 2016; Devlin et al., 2017).

For the task of engineering deep learning systems, *differentiable programming languages* (Maclaurin et al., 2015; Paszke et al., 2017a; Bosnjak et al., 2017; Gaunt et al., 2016a; Bunel et al., 2016) have emerged as a powerful framework. In contrast to earlier libraries (Abadi et al., 2016; Theano Development Team, 2016) which require the programmer to explicitly construct a computation graph, differentiable languages are similar in syntax, or sometimes direct extensions, to traditional programming languages. However, programs here are differentiable in their parameters, permitting gradient-based parameter learning. This framework has proven especially powerful for representing architectures that have input-dependent structure, such as deep networks over trees (Socher et al., 2013; Allamanis et al., 2017) and graphs (Li et al., 2016; Kipf & Welling, 2017).

Differentiable programming can provide other benefits to learning, for example, that we can define an inductive bias in the space of programs, and that programs in a differentiable language are potentially interpretable by people. A recent paper by Gaunt et al. (2017) points to another key appeal: high-level languages facilitate *transfer* across learning tasks. The paper gives a language called NEURAL TERPRET (NTPT) in which programs can invoke a library of trainable neural networks as subroutines. The parameters of these "library functions" are learned along with the code that calls them. The modularity of the language allows knowledge transfer, as a library function trained on a task can be reused in later tasks. In contrast to standard methods for transfer learning in deep networks, which re-use the first few layers of the network, neural libraries have the potential to enable reuse of higher, more abstract levels of the network, in what could be called *high-level transfer*.

In spite of its promise, though, inferring the structure of differentiable programs is a fundamentally hard problem. Indeed, the algorithms that current methods have so far been shown to learn are fairly simple. For example, while the NTPT approach successfully learns to evaluate a handwritten expression left to right, it falters on the more complex task of finding a path through a grid of images.

In this paper, we show that *functional programming abstractions* and *symbolic program synthesis* can be useful tools in overcoming this difficulty. The goal in program synthesis (Alur et al., 2013; Solar-Lezama, 2013; Feser et al.,

[1]University of Edinburgh [2]Rice University.

2015) is to discover desirable programs (defined appropriately), out of all programs expressible in a language. While combinatorially hard, the problem has seen substantial recent progress. Indeed, in a recent head-to-head comparison, symbolic program synthesis was reported to outperform gradient-based program induction (Gaunt et al., 2016a).

Functional programming is particularly appealing for synthesis of differentiable programs because functional combinators can compactly describe many common neural architectures (Olah, 2015). For example, `fold` combinators can be used to describe recurrent neural networks over sequences, or bottom-up recursive networks over trees. Convolution over matrices, lists, trees, and graphs can also be naturally expressed as a functional combinator. Because the same combinators can be used to express rich algorithmic operations ranging from list reversal to shortest path computation, they form a uniform, parsimonious idiom for describing tasks that combine perception and reasoning. Also, symbolic techniques are appealing for learning neural libraries because their strengths complement those of stochastic gradient descent (SGD): while SGD has proven remarkably effective for learning network parameters, each step of a symbolic search can explore large changes to the network structure.

Concretely, we present a framework for differentiable programming, called HOUDINI, in which a program synthesizer is used to search over models that are then trained using neural techniques. Essentially, HOUDINI can be seen as a program synthesis method for a functional subset of a language like Pytorch (Paszke et al., 2017a). HOUDINI is fundamentally a *neurosymbolic method*, in which symbolic techniques from program synthesis are used to select the structure of the network, described by a compact functional program, while gradient descent is used to tune parameters end-to-end. An interesting aspect of this approach is that the functional program can specify the architecture of the network, by using functional combinators to express the network connectivity, and can also specify learning transfer, by allowing the search procedure to choose among library functions. Thus, neurosymbolic synthesis can potentially enable new families of methods for transfer learning and few-shot learning.

Like Gaunt et al. (2017), we evaluate HOUDINI in the setting of *lifelong learning* (Thrun & Mitchell, 1995). Here, a model is trained on a series of tasks, and each training round is expected to improve performance in all tasks encountered so far. This setting raises the difficult issues of *catastrophic forgetting*, in which later tasks overwrite what has been learned from earlier tasks, and *negative transfer*, in which attempting to use information from earlier tasks hurts performance on later tasks. Our use of a neural library avoids both of these problems, because the library provides a natural mechanism to freeze and selectively re-use portions of networks that have been successful.

Our flagship task, a generalization of the grid navigation problem in Gaunt et al. (2017), is to compute least-cost paths in a grid whose nodes are labeled by images that contain numbers (specifically, speed limits). On this task, HOUDINI is able to discover an algorithm that has the structure of the Bellman-Ford shortest path algorithm (Bellman, 1958), but uses a learned neural function that approximates the algorithm's "relaxation" step. Our experiments suggest that our approach leads to greater transfer than traditional "low-level" transfer learning, in which weights learned in one round of training are reused on new tasks. Indeed, we show that the knowledge learned here can be transferred across different classes of images — specifically, from grids labeled with MNIST digits to grids labeled with speed limits.

2. The HOUDINI Programming Language

HOUDINI consists of two components. The first is a typed functional language of differentiable programs. The second is a learning procedure split into a symbolic module and a neural module. Given a task, the symbolic module enumerates parameterized programs in the HOUDINI language. The neural module uses gradient descent to find optimal parameters for synthesized programs; it also assesses the quality of solutions and decides whether an adequately performant solution has been discovered. In this section, we present our language. The next section presents our synthesizer.

As is standard in program synthesis, we introduce a specialized programming language that defines the set of all programs that can be produced by the synthesis procedure. Because our goal is to synthesize deep models, every program in the language defines a function that is differentiable with respect to a vector of parameters. Importantly, the language is also functional. To a first approximation, a *functional programming language* is one in which procedures are mathematical functions which do not have persistent state, and in which functions are so-called *first-class objects*, which means that they can be passed as inputs and potentially returned by other functions. Functional programming is important for HOUDINI, both because functional languages have important benefits for program synthesis, but also because, as we will see, the abstractions provided by functional languages describe the most popular deep architectures in a natural and compact way.

Our language design is based on three ideas:

- The ubiquitous use of *function composition* to glue together different networks.
- The use of *higher-order combinators* such as **map** and **fold** to uniformly represent neural architectures as well as patterns of recursion in procedural tasks.

Types τ:

$$\begin{array}{lll} \tau & ::= & Atom \mid ADT \mid F \\ Atom & ::= & \texttt{bool} \mid \texttt{real} \\ TT & ::= & Atom \mid \texttt{Tensor}\langle Atom\rangle[m_1][m_2]\dots[m_k] \\ ADT & ::= & TT \mid \alpha\langle TT\rangle \\ F & ::= & ADT \mid F_1 \to F_2. \end{array}$$

Programs $\mathbf{e}$:

$$e \quad ::= \quad \langle\!\langle \tau \rangle\!\rangle \mid \oplus_w \mid e_0 \circ e_1 \mid \mathbf{map}_\alpha\ e \mid \mathbf{fold}_\alpha\ e \mid \mathbf{conv}_\alpha\ e.$$

Figure 1. Syntax of types τ and partial programs e in HOUDINI. Here, α is the name of an ADT, for example `list` or `graph`; $m_1, \dots, m_k \geq 1$ define the shape of a tensor; $F_1 \to F_2$ is the type of a function from F_1 to F_2. In the definition of e, $\langle\!\langle \tau \rangle\!\rangle$ is a *hole*, $\oplus_w$ is a neural library function parameterized by weights w; $\circ$ is the composition operator; and **map**, **fold**, and **conv** denote map, fold, and convolution over a data type α.

- The use of a strong *type discipline* to distinguish between neural computations over different forms of data, and to avoid generating provably incorrect programs during symbolic exploration.

Figure 1 gives shows the grammar for the HOUDINI language. Here, τ denotes types and e denotes *partial* programs, i.e., programs with "holes" that represent missing code. The concept of partial programs is important for us as we synthesize programs through top-down refinement, starting with a program with a missing body, and then iteratively adding code until there are no holes to fill. Now we explain the language in more detail.

Types. The "atomic" data types in HOUDINI are booleans (`bool`) and reals. For us, `bool` is relaxed to mean a real value in $[0 \dots 1]$, which for example, allows the type system to track whether a vector has been passed through a sigmoid. *Tensors* over these types are also permitted. We have a distinct type $\texttt{Tensor}\langle\tau\rangle[m_1][m_2]\dots[m_k]$ for tensors of shape $m_1 \times \dots \times m_k$ whose elements have atomic type τ. (The dimensions $m_1, \dots, m_k$, as well as k itself, are bounded to keep the number of types finite.)

The language allows abstract data types (ADTs) $\alpha\langle\tau\rangle$ parameterized by a tensor or atomic type τ. The current implementation of HOUDINI supports two families of such types: $\texttt{list}\langle\tau\rangle$, lists with elements of type τ, and $\texttt{graph}\langle\tau\rangle$, graphs whose nodes contain values of type τ. However, this set can be augmented in principle. Finally, the language allows functions whose inputs and outputs can be of the aforementioned types, or other functions.

Programs. The fundamental operation in HOUDINI is *function composition*. Such composition can involve two classes of functions: differentiable library functions $\oplus_w$, that are parameterized by weights w and implemented by neural networks, and symbolic higher-order combinators that implement the control flow into which these neural functions are embedded.

We allow the following three families of combinators. The first two are standard constructs for functional languages, whereas the third is introduced specifically for deep models.

- Map combinators $\mathbf{map}_{\alpha\langle\tau\rangle}$, for ADTs of the form $\alpha\langle\tau\rangle$. Suppose e is a function. The expression $\mathbf{map}_{\texttt{list}\langle\tau\rangle}\ e$ is a function that, given a list $[a_1, \dots, a_k]$, returns the list $[e(a_1), \dots, e(a_k)]$. The expression $\mathbf{map}_{\texttt{graph}_\tau}\ e$ is a function that, given a graph G whose i-th node is labeled with a value a_i, returns a graph that is identical to G, but whose i-th node is labeled by $e(a_i)$.
- Left-fold combinators $\mathbf{fold}_{\alpha\langle\tau\rangle}$. For a function e, $\mathbf{fold}_{\texttt{list}\langle\tau\rangle}\ e$ is the function that, given a list $[a_1, \dots, a_k]$, returns the value $(e\ (e\ \dots (e\ (e\ a_1\ a_2)\ a_3) \dots)\ a_k)$. To define fold over a graph, we assume a linear order on the graph's nodes. Given G, the function $\mathbf{fold}_{\texttt{graph}\langle\tau\rangle}\ e$ returns the fold over the list $[a_1, \dots, a_k]$, where a_i is the value at the i-th node in this order.
- Convolution combinators $\mathbf{conv}_{\alpha\langle\tau\rangle}$. Let $p > 0$ be a fixed constant. For a function e, $\mathbf{conv}_{\texttt{list}\langle\tau\rangle}\ e$ is the function that, given a list $[a_1, \dots, a_k]$, returns the list $[a'_1, \dots, a'_k]$, where $a'_i = \mathbf{fold}_{\texttt{list}\langle\tau\rangle}\ e\ [a_{i-p}, \dots, a_i, \dots, a_{i+p}]$. (We define $a_j = a_1$ if $j < 1$, and $a_j = a_k$ if $j > k$.) Given G, the function $\mathbf{conv}_{\texttt{graph}\langle\tau\rangle}\ e$ returns the graph G' whose node u contains the value $\mathbf{fold}_{\texttt{list}\langle\tau\rangle}\ e\ [a_{i_1}, \dots, a_{i_m}]$, where a_{i_j} is the value stored in the j-th neighbor of u (according to the linear order on graph nodes).

The notation $\langle\!\langle \tau \rangle\!\rangle$ represents a *hole* of type τ. A program with holes (a partial program) has no operational meaning, and is only an intermediate structure used during synthesis.

Every neural library function is assumed to be annotated with a type. HOUDINI checks whether a partial program uses types in a consistent way, using the following rules:

- The type of a hole $\langle\!\langle \tau \rangle\!\rangle$ is, axiomatically, τ. The type of a library function $\oplus_w$ is, axiomatically, the type that it is annotated with.
- $e = e' \circ e''$ is assigned a type iff e' has type $\tau \to \tau'$ and e'' has type $\tau' \to \tau''$. In this case, e has type $\tau \to \tau''$.
- $e = \mathbf{map}_{\alpha\langle\tau\rangle}\ e'$ is assigned a type iff e' has the type $\tau \to \tau'$. In this case, the type of e is $\alpha\langle\tau\rangle \to \alpha\langle\tau'\rangle$.
- $e = \mathbf{fold}_{\alpha\langle\tau\rangle}\ e'$ is assigned a type iff e' has the type $\tau' \to (\tau \to \tau')$. In this case, e has type $\alpha\langle\tau\rangle \to \tau'$.
- $e = \mathbf{conv}_{\alpha\langle\tau\rangle}\ e'$ is assigned a type iff e' has the type $\tau' \to (\tau \to \tau')$. In this case, e has type $\alpha\langle\tau\rangle \to \alpha\langle\tau'\rangle$.

If it is not possible to assign a type to the partial program e, then it is invalid.

Note that complete HOUDINI programs do not have variables. Programs are based almost entirely on function com-

Figure 2. A grid of 32x32x3 images selected from the German Traffic Sign Recognition Benchmark dataset (Stallkamp et al., 2012). The least-cost path from the top left to the bottom right node is marked.

position; only higher-order combinators are ever directly applied. Thus, the language follows the *point-free* style of functional programming (Backus, 1978), which has its roots in combinatory logic (Curry et al., 1958). This stylized representation of programs is highly expressive, but is also succinct and reduces the amount of enumeration needed during synthesis.

HOUDINI for deep learning. The language has several properties that are useful for specifying deep models. First, one can inductively show that HOUDINI programs are differentiable, that is, any complete program e in our language is differentiable in the parameters w of the neural library functions used in e.

Second, common deep architectures can be compactly represented in our language. For example, deep feedforward networks can be represented by $\oplus_1 \circ \ldots \oplus_k$, where each $\oplus_i$ is a neural function. Following Olah (2015), recurrent neural nets can be naturally expressed as $\mathbf{fold}_{\texttt{list}\langle\alpha\rangle}\ \oplus$, where $\oplus$ is a neural function. Graph convolutional networks can be expressed using terms of form $\mathbf{conv}_{\texttt{graph}\langle\alpha\rangle}\ \oplus$. Going even further, straightforward extensions of our language are possible to handle bidirectional recurrent networks, attention mechanisms, and so on, which we leave to future work.

Example: Shortest path in a grid of images. Now we show how HOUDINI can model tasks that mix perception and procedural reasoning, using an example that generalizes the navigation task of Gaunt et al. (2017). Suppose we are given a grid of images (for example, Figure 2), where elements represent speed limits on various streets and are connected horizontally and vertically, but not diagonally. Passing through each node induces a penalty, which depends on the node's speed limit, with lower (stricter) speed limit having a higher penalty. The task is to predict the minimum cost $d(u)$ incurred while traveling from a fixed starting point $init$ (the top left element) to each of the remaining nodes u.

One way to compute these costs is to use the Bellman-Ford shortest-path algorithm (Bellman, 1958). This algorithm is an iterative, dynamic-programming computation whose i-th iteration computes an estimated minimum cost $d_i(u)$ of travel to each node u in the graph. The cost estimates for the $(i+1)$-th iteration are given by applying a so-called *relaxation* operation:

$$d_{i+1}(u) := \min(d_i(u), \min_{v \in Adj(u)} d_i(v) + w(v))$$

where $w(u)$ is the penalty and $Adj(u)$ the neighbors of u.

Because the update to $d_i(u)$ only depends on values stored at u and its neighbors, the relaxation step can be represented as a convolution. This convolution applies to a graph whose nodes contain costs, and this graph is obtained by applying a *map* to the original graph of images. The Bellman-Ford algorithm amounts to repeated application, until convergence, of this convolution step.

For any function e, let e^i represent the composition of e with itself, i times. The HOUDINI program that models the shortest-path task is

$$(\mathbf{conv}^i_{\texttt{graph}\langle\texttt{Tensor[2]}\rangle}\ nn\_relax) \circ (\mathbf{map}_{\texttt{graph}\langle\texttt{Tensor[32][32][3]}\rangle}\ perceive).$$

Here, the function $perceive$ processes the images of speed limits and generates a tensor of type $\texttt{Tensor}[2]$ (the elements of the tensor store the penalty for the node and the current estimate of the cost of travel to the node). The nn_relax is a neural network that models that *relaxation* operation. As described in Section 4, HOUDINI is able to discover an approximation of this program purely from data.

3. Program Synthesis

Our program synthesis problem is as follows. For a (complete) HOUDINI program e_w parameterized by a vector w, let $e[w \mapsto v]$ be the function for the specific parameter vector v, i.e. by substituting w by v in e. Suppose we are given a library $\mathcal{L}$ of neural functions, as well as a training set $\mathcal{D}$ that describes the function that we want to learn. As usual, we assume that $\mathcal{D}$ consists of i.i.d. samples from a distribution p_{data}. We assume that $\mathcal{D}$ is properly typed, i.e., every training instance $(x_i, y_i) \in \mathcal{D}$ has the same type, which is known. We note that this means that we also know the type τ of our target function.

The learning problem can then be expressed as a problem in program synthesis. The goal is to discover a complete program e^*_w of type τ, and values v for w such that

$$e^*_w[w \mapsto v] = \text{argmin}_{e \in Progs(\mathcal{L}), w \in \mathbb{R}^n}(\mathbf{E}_{x \sim p_{data}}[l(e, \mathcal{D}, x)]),$$

where $Progs(\mathcal{L})$ is the universe of all complete programs over the library $\mathcal{L}$, and l is an appropriate loss function.

Our algorithm for this task (called SYNTH in this section) consists of a higher-level, symbolic program enumeration module called GENERATE and a lower-level gradient learning module called LEARN. The former module repeatedly generates complete, parameterized programs that are sent to the latter module. For each such program, LEARN discovers optimal parameter values, and updates an estimate of the globally minimal loss. This process continues until a function with sufficiently low loss is identified.

We skip a detailed discussion of LEARN as it is uses standard gradient descent. The symbolic module GENERATE follows a strategy of *heuristically-guided top-down iterative refinement* similar to the λ^2 program synthesizer (Feser et al., 2015). We now describe this module in more detail.

The initial input to GENERATE is the type τ of the function we want to learn. The procedure proceeds iteratively, maintaining a priority queue Q of *synthesis subtasks* of the form (e, f), where e is a type-safe partial or complete program of type τ, and f is either a hole of type τ' in e, or a special symbol $\perp$ indicating that e is complete. The interpretation of such a task is to find a replacement e' of type τ' for the hole f such that the program e'' obtained by substituting f by e' is complete. (Because e is type-safe by construction, e'' is of type τ.) The queue is sorted according to a heuristic cost function that prioritizes simpler programs.

The procedure iteratively processes subtasks in the queue Q, selecting a task (e, f) in the beginning of each iteration. If the program e is complete, it is sent to the neural module for parameter learning. Otherwise, GENERATE expands the program e by proposing a partial program that fills the hole f. To do this, GENERATE selects a production rule for partial programs from the language grammar (Figure 1). Suppose the right hand side of this rule is α. The algorithm constructs an expression e' from α by replacing each nonterminal in α by a hole with the same type as the nonterminal. If e' is not of the same type as f, it is automatically rejected. Otherwise, the algorithm constructs the program $e'' = e[f \mapsto e']$. For each hole f' in e'', the algorithm adds to Q a new task (e'', f'). If e'' has no hole, it adds to Q a task $(e'', \perp)$. Finally, at the end of GENERATE, we return the complete program e that has the best accuracy, as computed during LEARN, on a validation set.

While this enumeration procedure faces an inevitable combinatorial explosion, our use of functional abstractions and types help limit the amount of enumeration that is needed. For example, suppose the end goal of GENERATE is to synthesize a program of type $\texttt{list}\langle\texttt{Tensor}\langle\texttt{real}\rangle\rangle[3][28][28] \rightarrow \texttt{Tensor}\langle\texttt{real}\rangle[1]$. Our type system can immediately determine that the target program cannot have either of the forms $\mathbf{map}_\tau\ e$ or $\mathbf{conv}_\tau\ e$ for *any* e. Indeed, given the combinators in our language, the only possibility is that the target is of form $\mathbf{fold}_\tau\ e$, where $\tau = \texttt{list}\langle\texttt{Tensor}\langle\texttt{real}\rangle\rangle[3][28][28]$ and e can only have the type $\tau' = \texttt{Tensor}\langle\texttt{real}\rangle[1] \rightarrow (\texttt{Tensor}\langle\texttt{real}\rangle[3][28][28] \rightarrow \texttt{Tensor}\langle\texttt{real}\rangle)[1]$. Thus, in its first iteration, GENERATE can only add a single subtask $(\mathbf{fold}_\tau\ \langle\!\langle \tau' \rangle\!\rangle)$ to the task queue. We present empirical evidence on the effectiveness of types in pruning the search space in Section 4.

3.1. Application to Lifelong Learning

Because the synthesizer takes as input a library of neural modules, it can be naturally applied to lifelong learning. For us, a lifelong learning setting is a sequence of related tasks $\mathcal{D}_1, \mathcal{D}_2, \ldots$, where each task $\mathcal{D}_i$ has its own training and validation set. In this setting, the function SYNTH is called repeatedly, once for each task $\mathcal{D}_i$ using a neural library $\mathcal{L}_i$, returning a best-performing program e_i^* with parameters v_i^*.

We implement transfer learning simply by adding new modules to the neural library after each call to SYNTH. We add all of the neural functions from e_i^* back into the library, freezing their parameters. More formally, let $\oplus_{i1} \ldots \oplus_{iK}$ be the neural library functions which are called anywhere in e_i^*. Each library function $\oplus_{ik}$ has parameters w_{ik}, which have been set to the value v_{iK}^* by LEARN. Then the library for the next task is then $\mathcal{L}_{i+1} = \mathcal{L}_i \cup \{\oplus_{ik}[w_{ik} \mapsto v_{ik}^*]\}$. By substituting the parameters in this way, the parameter vectors of $\oplus_{ik}$ are frozen and can no longer be updated by subsequent tasks. Thus, we prevent catastrophic forgetting by design. Importantly, it is always possible for the synthesizer to introduce "fresh networks" whose parameters have not been pretrained. This is because the library always monotonically increases over time, so that the original neural library function with untrained parameters is still available.

This approach has the important implication that the set of neural library functions that the synthesizer uses is not fixed, but continually evolving. Because both trained and untrained versions of the library functions are available, this can be seen to permit *selective transfer*, meaning that depending on which version of the library function is chosen by the synthesizer, the learner has the option of using or not using previously learned knowledge in a new task. This way HOUDINI avoids negative transfer.

4. Evaluation

We evaluate HOUDINI on several lifelong learning settings that combine perceptual and algorithmic reasoning. Our aim is to show that differentiable functional program synthesis allows us to learn neural libraries that can transfer both perceptual and algorithmic knowledge across a series of learning tasks. We are particularly interested in evaluating several different types of transfer that are particularly

important in lifelong learning, namely, *low-level transfer* of perceptual concepts across domains, *high-level transfer* of algorithmic concepts that rely on different perceptual concepts as input, and *selective transfer* where the learning method itself makes the decision about which concepts from previous tasks to be reused. We demonstrate that all of these different modes of transfer learning occur as a natural byproduct of our neurosymbolic framework.

4.1. Tasks

Each lifelong learning setting that we propose is a sequence of individual learning tasks. The full list of tasks that we propose is shown in Figure 3. The individual tasks have an object recognition component operating on one of three different data sets of images: the MNIST data set of handwritten digits (LeCun et al., 1998), the NORB object recognition data set (LeCun et al., 2004), and finally the GTSRB data set of images of traffic signs (Stallkamp et al., 2012). All images are pre-processed to have zero mean and standard deviation of one. Additionally, NORB images were resized to 28x28 in order to match the MNIST dimensionality. Finally, the GTSRB images were all re-sized to 32x32x3.

We propose three higher-level tasks that build on the object recognition tasks. These are *counting* the number of instances of images of a certain class in a list of images; *summing* a list of images of digits; and finding the *shortest path* in a graph where evaluating the weight of an edge in the graph requires perceptual reasoning. We evaluate whether it is possible to transfer these high-level tasks across different perceptual settings.

We combine these low-level and high-level tasks into six task sequences (Figure 3). Three of the sequences (CS1, SS, GS1) involve low-level transfer, in which earlier tasks are perceptual tasks like recognizing digits, while later tasks introduce higher-level algorithmic problems that build on the earlier concepts. The other three task sequences (CS2, CS3, GS2) involve higher-level transfer, in which earlier tasks introduce a high-level concept, later tasks require a learner to re-use the high-level concept on different perceptual inputs. For example, in CS2, once the $count\_digit()$ is learned for counting digits of class d_1, the synthesizer can learn to reuse this counting network on a new class of digit d_2, even if the learning system has never before seen a digit of class d_2. Sequence CS3 displays the same principle, with the two image domains being different, namely, MNIST and NORB. Finally, the graph tasks sequence GS1 also demonstrates that the graph convolution combinator in HOUDINI allows learning of complex graph algorithms and GS2 tests if high-level transfer can be performed with this more complex task.

Individual tasks

recognize_digit(d): Binary classification of whether image contains a digit $d \in \{0 \dots 9\}$

classify_digit: Classify a digit into digit categories $(0 - 9)$

is_toy(t): Binary classification of whether an image contains a toy $t \in \{0 \dots 4\}$

regress_speed: Return the speed value from an image of speed limit sign.

regress_mnist: Return the value from a digit image from MNIST dataset.

count_digit(d): Given a list of images, count the number of images of digit d

count_toy(t): Given a list of images, count the number of images of toy t

sum_digits: Given a list of digit images, compute the sum of the digits.

shortest_path_street: Given a grid of images of speed limit signs, find the shortest distances to all other nodes

shortest_path_mnist: Given a grid a MNIST images, and a source node, find the shortest distances to all other nodes in the grid.

Task Sequences

Counting

CS1: Evaluate low-level transfer.
Task 1: recognize_digit($d1$); *Task 2*: recognize_digit($d2$); *Task 3*: count_digit($d1$); *Task 4*: count_digit($d2$)

CS2: Evaluate high-level transfer and whether perceptual tasks can be learned from supervision of higher-level tasks.
Task 1: recognize_digit($d1$); *Task 2*: count_digit($d1$); *Task 3*: count_digit($d2$); *Task 4*: recognize_digit($d2$)

CS3: Evaluate high-level transfer of counting across different image domains.
Task 1: recognize_digit(d); *Task 2*: count_digit(d); *Task 3*: count_toy(t); *Task 4*: is_toy(t)

Summing

SS: Demonstrate low-level transfer of a multi-class classifier as well as the advantage of functional methods like foldl in specific situations.
Task 1: classify_digit; *Task 2*: sum_digits

Single-Source Shortest Path

GS1: Learning of complex algorithms.
Task 1: regress_speed; *Task 2*: shortest_path_street

GS2: High-level transfer of complex algorithms.
Task 1: regress_mnist; *Task 2*: shortest_path_mnist; *Task 3:* shortest_path_street

Figure 3. Tasks and task sequences

4.2. Experimental Setup

In addition to the *higher-order combinators*, the symbolic module GENERATE also uses utility functions **repeat** and **zeros**. The function **repeat** takes in two arguments, an

integer n and a function f of type $\tau \to \tau$. The expression $\mathbf{repeat}\ n\ f$ is a syntactic sugar for a composition of function f with itself n times. The function $\mathbf{zeros}$ is used for generating zero tensors of appropriate dimensions.

For HOUDINI, the neural library functions are represented as one of the pre-determined neural modules: multi-layer perceptrons (MLPs), convolutional neural networks (CNNs) and recurrent neural networks (RNNs). MLP modules have one hidden layer of size 50. CNNs have two convolutional layers with 32 and 64 output channels respectively, each followed by max pooling, followed by a fully connected layer, a dropout and an output layer. RNN modules are long short-term memory (LSTM) networks with a hidden dimension of 100, followed by an output layer, which transforms the last hidden state. For a given task, we use the input and output types of the new function to decide between MLP, CNN, or RNN, and also deduce the output activation function.

We use two types of baselines: 1) *standalone networks*, which do not do transfer learning, but simply train a new network for each task in the sequence, starting from a random weights; and 2) a purely neural approach to *low-level transfer*, which transfers all weights learned in the previous task, except for the output layer that is kept task-specific. This is designed to reflect standard approaches to transfer learning, which re-use trained layers. The baselines use MLP, CNN and RNN modules as appropriate to the task, with the same architecture as the neural modules in HOUDINI. Specifically, if the input to the task is a single image, the baseline uses a CNN. If the input is a list of images, the baseline architecture processes individual images with the same CNN and finally applies an RNN. We use a similar combination for the shortest path tasks. The first task for sequences GS1 and GS2 is regression, which we train using a CNN, in which the last layer is linear. For the other tasks, we map a learned CNN regression module to each image in the grid. Afterwards, we linearize the grid row-wise, converting it into ofa list, and then we process it using an LSTM (RNN) with hidden state of size 100. The number was chosen so that both our implementation and the baseline have almost the same number of parameters.

For all digit tasks, the results shown are averaged over five randomly-chosen pairs of digit classes d_1 and d_2. For multi-class classification (Sequence SS - Task 1) and regression (GS1 - Task1, GS2 - Task 1), we used all training images available. For the rest of the tasks in GS1, GS2, GS3 and SS, we use 12000 data points for training, with 2100 for testing. The list lengths for training are [2, 3, 4, 5], and [6, 7, 8] for testing in order to evaluate the generalization to longer sequences. We train for 20 epochs on all list-related tasks and for 1000 epochs for the regression tasks. The training datasets for the graph shortest path tasks (GS1 - Task 2; GS2 - Task2, GS2 - Task3) consists of 70,000 3x3 grids and 1,000,000 4x4 grids, while the testing datasets consists of 10,000 5x5 grids. The number of epochs for these tasks is 5. In GS2 - Task3, the *low-level transfer* baseline reuses the regression function learned in GS2 - Task1, thus, the image dimensions from MNIST and the colored GTSRB need to match. Therefore, we expanded the MNIST digit images, used for the graph sequences GS1 and GS2, to 32x32x3 dimensionality.

Config.	Task	Number of programs		
		size = 4	size = 5	size = 6
Without types	Task 1	8182	110372	1318972
	Task 2	12333	179049	2278113
	Task 3	17834	278318	3727358
	Task 4	24182	422619	6474938
With types	Task 1	2	20	44
	Task 2	5	37	67
	Task 3	9	47	158
	Task 4	9	51	175

Table 1. Number of programs of different sizes (4, 5, and 6) generated by the symbolic module GENERATE without a type system and with a type system for the task sequence CS1.

We evaluate the competing approaches on various amounts of data, namely with 5%, 10%, 50% and 100% of the training data. For classification tasks, we report error, while for the regression tasks — counting, summing, regress_speed and shortest_path — we report root mean-squared error (RMSE). In all cases, we use early stopping, reporting the the test error at the epoch where the testing error was minimized.

A Note on Notation. In the rest of this paper, while presenting programs, we denote the map, fold, and convolution combinators over lists by $\mathbf{map\_l}$, $\mathbf{fold\_l}$, and $\mathbf{conv\_l}$, respectively. The corresponding combinators for graphs are respectively denoted $\mathbf{map\_g}$, $\mathbf{fold\_g}$, and $\mathbf{conv\_g}$. We omit mention of types when we present names of combinators. We write $\mathbf{compose}(e_1, e_2)$ for $e_1 \circ e_2$.

4.3. Results

First we evaluate the performance of the methods on the sequence based task sequences (Figures 4-6). In all cases where there is an opportunity to transfer from previous tasks, we see that HOUDINI has much lower error than either low-level transfer or standalone networks. We note that for the early tasks in each task sequence (e.g. CS1 tasks 1 and 2), there is little relevant information that can be transferred from early tasks, so as expected all methods perform similarly; e.g., the output of HOUDINI is a single library function. The actual programs generated by HOUDINI for all tasks are listed in Appendix A.

Task sequence CS1 (Figure 4) evaluates the methods' ability to selectively perform low-level transfer of a perceptual concept across higher level tasks. The first task that

provides a transfer opportunity is CS1 task 3, but there are two potential lower-level tasks that the methods could transfer from, namely, the tasks recognize_digit(d_1) and recognize_digit(d_2) respectively. HOUDINI first learns neural modules nn_cs1_1 and nn_cs1_2 for these two tasks, respectively. During the training for the count_digit($d1$) task, both the previously learned neural modules nn_cs1_1 and nn_cs1_2 are available in the library. The learner, however, picks the right module (nn_cs1_1) for reuse and learns the program

$$\mathbf{compose}(\mathbf{fold\_l}(\text{nn\_cs1\_5}, \mathbf{zeros}(1)), \mathbf{map\_l}(\text{lib.nn\_cs1\_1}))$$

where nn_cs1_5 is a fresh neural module. The baseline of low-level transfer cannot select which of the previous tasks to re-use, and so suffers worse performance.

In CS1 task 4 (Figure 4d), the functions required for this task are already learned, and as Figure 4d shows, HOUDINI is able to reuse the previously learned relevant digit recognizer and counter in order to achieve the same top performance, even in low-data settings. This is not possible for the baseline methods.

Task sequence CS2 examines the methods' ability to transfer higher-level concepts, in this case, the concept of counting, across different digit classification tasks. Here CS2 task 3 (Figure 5c) is the task that provides the first opportunity for transfer. We see that HOUDINI is able to learn much faster on this task because it is able to reuse a network which has learned from the previous counting task. The low-level transfer baseline, on the other hand exhibits negative transfer, performing worse than the standalone network without any transfer.

Finally, task sequence CS3 examines whether the methods can demonstrate high-level transfer when the input image domains are very different, from the MNIST domain to the NORB domain of toy images. We see in Figure 6c that the higher-level network still successfully transfers across tasks, learning a effective network for counting the number of toys of type t_1, even though the network has not previously seen any toy images at all. What is more, it can be seen that because of the high-level transfer, HOUDINI has learned a modular solution to this problem. From the subsequent performance on a standalone toy classification task (Figure 6d), we see that CS3 task 3 has already caused the network to induce a re-usable classifier on toys. Overall, it can be seen that HOUDINI outperforms all the baselines even under the limited data setting, confirming the successful selective transfer of both low-level and high-level perceptual information. Similar results can be seen on the summing task (see Appendix B).

Finally, for the graph-based tasks (Tables 2 and 3), we see that the graph convolutional program learned by HOUDINI on the graph tasks has significantly less error than a simple sequence model. For the shortest_path_street task in the graph sequence GS1, HOUDINI learns a program

$$\mathbf{compose}(\mathbf{repeat}(9, \mathbf{conv\_g}(\text{nn\_gs1\_2})), \mathbf{map\_g}(\text{lib.nn\_gs1\_1}))$$

where nn_gs1_2 is a learned relaxation function and lib.nn_gs1_1 is the neural module learned for the earlier task regress_speed. In the resulting program, we observe that a learned neural module is convolved over the graph, and the learned function is broadly similar to the relaxation operator for the Bellman-Ford algorithm. For the shortest_path_street task in the graph sequence GS2, HOUDINI learns a program

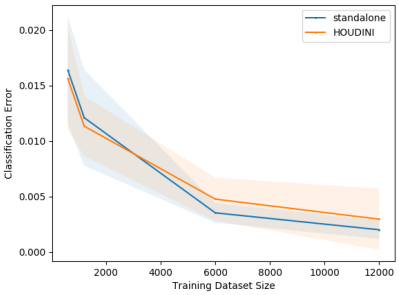
$$\mathbf{compose}(\mathbf{repeat}(9, \mathbf{conv\_g}(\text{lib.nn\_gs2\_2})), \mathbf{map\_g}(\text{nn\_gs2\_5}))$$

where nn_gs2_5 is the newly learned regress function for the street signs and lib.nn_gss_2 is the relaxation function already learned during the training for the earlier task shortest_path_mnist. In Table 3, we see that a graph program with the relaxation function learned on MNIST can be applied to a graph of street signs, suggesting that a domain-general operation is being learned.

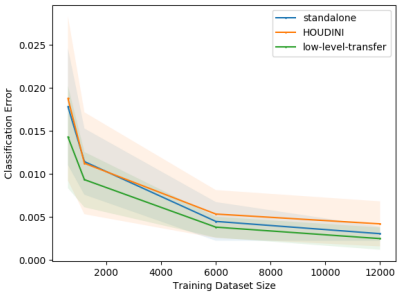
Finally, we demonstrate the extent to which incorporating ideas from program synthesis improves the scalability of the method, by assessing the impact of the type system on the symbolic search. We counted the number of programs generated by the symbolic module GENERATE with and without a type system. Let the *size* of a program be the number of occurrences (including repetitions) of library functions and higher-order combinators in the program's text. Table 1 shows the number of programs of different sizes generated for the tasks in the task sequence CS1. Since the typical program size for the task sequences we are evaluating is not more than 6, we varied the target program size from 4 to 6, When the type system is disabled, the only constraint the symbolic module has while composing the programs is the arity of the functions in the library. As can be seen from the large number of programs generated in this case, this constraint is clearly not sufficient to bring down the number of candidate programs to a manageable size. With the type system, however, the symbolic module generates only type consistent programs resulting in a much smaller number of candidate programs.

5. Related Work

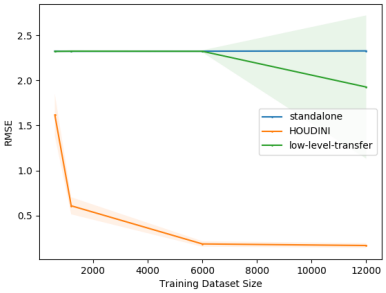
Differentiable programming refers to software frameworks that allow a programmer to define differentiable functions as programs in a Turing-complete language, and to automatically compute derivatives. Several frameworks



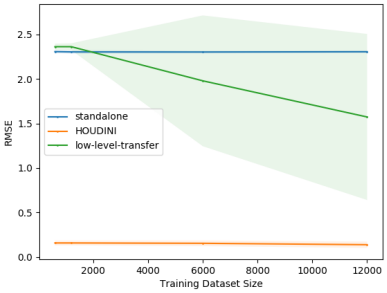
(a) Task 1: recognize_digit(d_1)



(b) Task 2: recognize_digit(d_2)

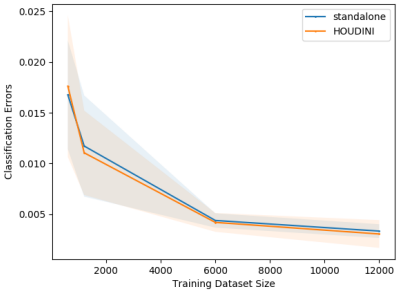


(c) Task 3: count_digit(d_1)

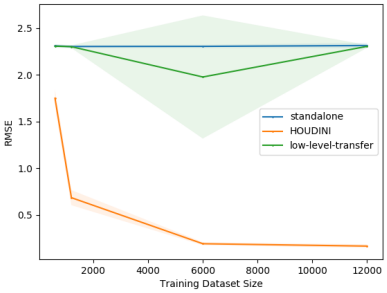


(d) Task 4: count_digit(d_2)

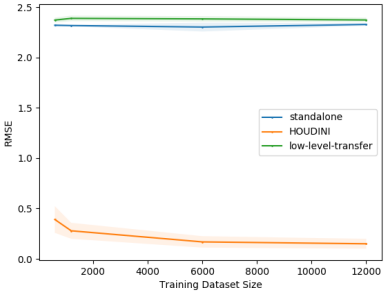
Figure 4. Lifelong learning for "learning to count" (Sequence CS1), demonstrating low-level transfer of perceptual recognizers.



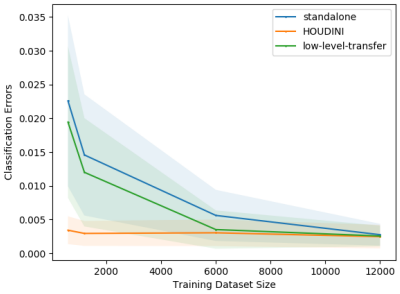
(a) Task 1: recognize_digit(d_1)



(b) Task 2: count_digit(d_1)

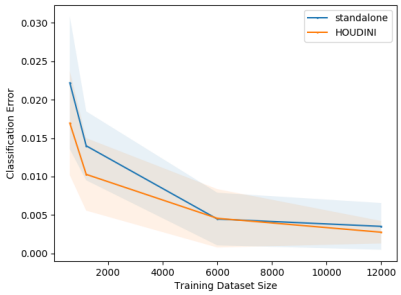


(c) Task 3: count_digit(d_2)

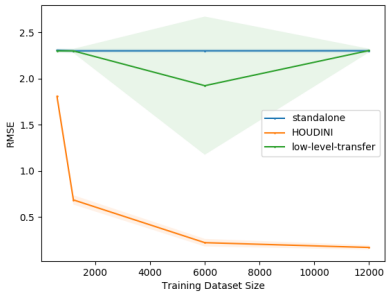


(d) Task 4: recognize_digit(d_2)

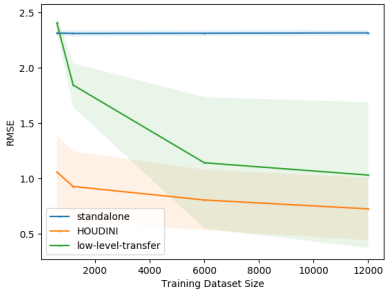
Figure 5. Lifelong learning for "learning to count" (Sequence CS2), demonstrating high-level transfer of a counting network across categories.



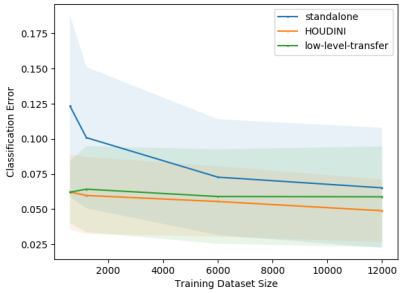
(a) Task 1: recognize_digit(d_1)



(b) Task 2: count_digit(d_1)



(c) Task 3: count_toy(t_1)



(d) Task 4: is_toy(t_1)

Figure 6. Lifelong learning for "learning to count" (Sequence CS3), demonstrating high-level transfer across different types of images. After learning to count MNIST digits, the same network can be used to count images of toys.

	Task 1	Task 2
RNN	0.37	5.96
HOUDINI	0.38	1.53

Table 2. Lifelong learning on graphs (task sequence GS1). Column 1: RMSE on speed/distance from image. Column 2: RMSE on shortest-path.

	Task 1	Task 2	Task 3
RNN	1.21	5.33	6.16
HOUDINI	1.32	1.64	3.44

Table 3. Lifelong learning on graphs, demonstrating high-level transfer (task sequence GS2). Column 1: RMSE on digits. Columns 2,3: RMSE on shortest-path.

have provided differentiable programming functionality in Python, including Autograd (Maclaurin et al., 2015), Pytorch (Paszke et al., 2017b), chainer (Tokui et al., 2015), and Tensorflow in eager mode (Shankar & Dobson, 2017). Other authors have explored differentiable versions of old languages like Forth (Riedel et al., 2016), and have introduced new differentiable languages like Terpret (Gaunt et al., 2016b).

One approach to learning representation of algorithms is *neural abstract machines* (Graves et al., 2014; Neelakantan et al., 2015; Reed & de Freitas, 2016; Cai et al., 2017), which aim to learn algorithms using recurrent architectures that incorporate additional memory sources such as stacks. Unlike HOUDINI, these methods do not produce an explicit symbolic representation of the algorithms that they learn.

As an alternative, *neural program synthesis* methods (Devlin et al., 2017; Balog et al., 2016; Parisotto et al., 2016; Kalyan et al., 2018) define deep architectures that output the code of programs that fit a given set of examples. Notably, these methods often combine deep networks with symbolic search. For example, some hybrid approaches (e.g. Balog et al., 2016; Ellis et al., 2017) use a deep model to predict the set of library functions that should be considered in a symbolic search. Unlike our work, however, neural program synthesis methods have not themselves synthesized differ-

entiable programs. The only exception, to our knowledge, is Gaunt et al. (2017), who consider synthesis of programs in the differentiable language NTPT. Their work has similar motivations to ours, but neither considers a functional language nor a neurosymbolic search. We demonstrate that both of these design principles lead to significant benefits.

A number of methods have been proposed for *neural architecture search*, including reinforcement learning, evolutionary computation, and best-first search (Zoph & Le, 2017; Liu et al., 2017; Real et al., 2017; Zhong et al., 2017). These search strategies could be incorporated into HOUDINI, and this is an intriguing avenue for future work. Indeed, this line of work is usefully orthogonal to ours: while HOUDINI provides a natural framework for formalizing the search problem, it does not commit to a search strategy.

In *lifelong learning* (Thrun & Mitchell, 1995; Carlson et al., 2010), the learning algorithm is given a series of tasks. The goal is to re-use information from earlier tasks in training of later tasks, while avoiding *catastrophic forgetting* (Kirkpatrick, 2017) of information from later tasks. Rusu et al. (2016) avoid catastrophic forgetting by freezing previously learned weight and introducing new ones, as well as a learnable lateral connections between the new and the old models. However, the topology of these connections is hard-wired, thus the architecture is unable to perform high-level transfer. This problem is addressed in Fernando et al. (2017), which optimizes the topology using a genetic algorithm. However, the experimental results show that the approach can experience negative transfer.

Neural module networks (NMNs) (Andreas et al., 2016), like our work, consider how to train networks with modular structure. Focusing on the task of visual question answering, they use a natural language parser to select and arrange relevant reusable neural modules, analogous to a program, for answering a question from an image. Hu et al. (2017) build on this work by replacing the natural language parser with a neural network, trained using a reinforcement learning algorithm. However, said network was observed to have difficulty while training, requiring initial expert-provided supervision. The major difference to our work is that NMNs require a natural language input to guide the method as to which modules to combine. The search procedure in HOUDINI is effective without this additional source of supervision on which program to select.

Symbolic program synthesis. In the traditional formulation of program synthesis, the goal is to find a program that satisfies a set of hard constraints. Many recent efforts target this problem using symbolic methods; see Gulwani et al. (2017) for a survey. Specifically, several of these papers (Feser et al., 2015; Osera & Zdancewic, 2015; Le & Gulwani, 2014) leverage compact functional representations of programs and type-based pruning to effectively search vast combinatorial spaces of programs. Our work repurposes these ideas to a setting where the objective is optimization, and where the synthesized programs are optimized using neural techniques.

6. Conclusion

We have presented HOUDINI, which is, so far as we know, the first symbolic program synthesis approach for differentiable functional programs. Deep networks can be naturally specified as differentiable programs, and functional programs can compactly represent popular deep architectures (Olah, 2015). Therefore, symbolic search through a space of differentiable functional programs is particularly appealing, because it can at the same time select both which pretrained neural library functions should be reused, and also what deep architecture should be used to combine them. On several lifelong learning tasks that combine perceptual and algorithmic reasoning, we showed that HOUDINI can accelerate learning by transferring high-level concepts. A key avenue for future work would be to explore search strategies by incorporating ideas from the literature on neurosymbolic synthesis of nondifferentiable programs (Balog et al., 2016; Kalyan et al., 2018; Parisotto et al., 2016; Ellis et al., 2017), and from neural architecture search (Zoph & Le, 2017; Liu et al., 2017; Real et al., 2017; Zhong et al., 2017).

References

Abadi, Martín, Barham, Paul, Chen, Jianmin, Chen, Zhifeng, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Irving, Geoffrey, Isard, Michael, Kudlur, Manjunath, Levenberg, Josh, Monga, Rajat, Moore, Sherry, Murray, Derek Gordon, Steiner, Benoit, Tucker, Paul A., Vasudevan, Vijay, Warden, Pete, Wicke, Martin, Yu, Yuan, and Zheng, Xiaoqiang. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pp. 265–283, 2016.

Allamanis, Miltiadis, Chanthirasegaran, Pankajan, Kohli, Pushmeet, and Sutton, Charles. Learning continuous semantic representations of symbolic expressions. In *International Conference on Machine Learning (ICML)*, 2017. URL `http://arxiv.org/abs/1611.01423`.

Alur, Rajeev, Bodík, Rastislav, Juniwal, Garvit, Martin, Milo M. K., Raghothaman, Mukund, Seshia, Sanjit A., Singh, Rishabh, Solar-Lezama, Armando, Torlak, Emina, and Udupa, Abhishek. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design*, FMCAD, pp. 1–17, 2013.

Andreas, Jacob, Rohrbach, Marcus, Darrell, Trevor, and

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

Backus, John. Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978. ISSN 0001-0782.

Balog, Matej, Gaunt, Alexander L, Brockschmidt, Marc, Nowozin, Sebastian, and Tarlow, Daniel. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Bellman, Richard. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

Bosnjak, Matko, Rocktäschel, Tim, Naradowsky, Jason, and Riedel, Sebastian. Programming with a differentiable forth interpreter. In *International Conference on Machine Learning, ICML*, pp. 547–556, 2017.

Bunel, Rudy R., Desmaison, Alban, Mudigonda, Pawan Kumar, Kohli, Pushmeet, and Torr, Philip H. S. Adaptive neural compilation. In *Advances in Neural Information Processing Systems 29*, pp. 1444–1452, 2016.

Cai, Jonathon, Shin, Richard, and Song, Dawn. Making neural programming architectures generalize via recursion. In *International Conference on Learning Representations (ICLR)*, 2017.

Carlson, Andrew, Betteridge, Justin, Kisiel, Bryan, Settles, Burr, Hruschka Jr, Estevam R, and Mitchell, Tom M. Toward an architecture for never-ending language learning. In *National Conference on Artificial Intelligence (AAAI)*, 2010.

Curry, Haskell Brooks, Feys, Robert, Craig, William, Hindley, J Roger, and Seldin, Jonathan P. *Combinatory logic*, volume 1. North-Holland Amsterdam, 1958.

Devlin, Jacob, Uesato, Jonathan, Bhupatiraju, Surya, Singh, Rishabh, Mohamed, Abdel-rahman, and Kohli, Pushmeet. Robustfill: Neural program learning under noisy I/O. In *International Conference on Machine Learning (ICML)*, 2017.

Ellis, Kevin, Ritchie, Daniel, Solar-Lezama, Armando, and Tenenbaum, Joshua B. Learning to infer graphics programs from hand-drawn images. *CoRR*, abs/1707.09627, 2017.

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

Feser, John K., Chaudhuri, Swarat, and Dillig, Isil. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 229–239, 2015.

Gaunt, Alexander L., Brockschmidt, Marc, Singh, Rishabh, Kushman, Nate, Kohli, Pushmeet, Taylor, Jonathan, and Tarlow, Daniel. Terpret: A probabilistic programming language for program induction. *CoRR*, abs/1608.04428, 2016a.

Gaunt, Alexander L, Brockschmidt, Marc, Singh, Rishabh, Kushman, Nate, Kohli, Pushmeet, Taylor, Jonathan, and Tarlow, Daniel. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016b.

Gaunt, Alexander L., Brockschmidt, Marc, Kushman, Nate, and Tarlow, Daniel. Differentiable programs with neural libraries. In *International Conference on Machine Learning (ICML)*, pp. 1213–1222, 2017.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Gulwani, Sumit, Polozov, Oleksandr, and Singh, Rishabh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

Hu, Ronghang, Andreas, Jacob, Rohrbach, Marcus, Darrell, Trevor, and Saenko, Kate. Learning to reason: End-to-end module networks for visual question answering. *CoRR, abs/1704.05526*, 3, 2017.

Kalyan, Ashwin, Mohta, Abhishek, Polozov, Oleksandr, Batra, Dhruv, Jain, Prateek, and Gulwani, Sumit. Neural-guided deductive search for real-time program synthesis from examples. In *International Conference on Learning Representations (ICLR)*, 2018.

Kipf, Thomas N. and Welling, Max. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

Kirkpatrick, James et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017. ISSN 0027-8424.

Kurach, Karol, Andrychowicz, Marcin, and Sutskever, Ilya. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

Le, Vu and Gulwani, Sumit. FlashExtract: A framework for data extraction by examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 542–553, 2014.

LeCun, Yann, Bottou, Léon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

LeCun, Yann, Huang, Fu Jie, and Bottou, Leon. Learning methods for generic object recognition with invariance to pose and lighting. In *Computer Vision and Pattern Recognition (CVPR)*, 2004.

Li, Yujia, Tarlow, Daniel, Brockschmidt, Marc, and Zemel, Richard. Gated graph sequence neural networks. In *International Conference on Learning Representations (ICLR)*, 2016.

Liu, Chenxi, Zoph, Barret, Shlens, Jonathon, Hua, Wei, Li, Li-Jia, Fei-Fei, Li, Yuille, Alan, Huang, Jonathan, and Murphy, Kevin. Progressive neural architecture search. arXiv:1712.00559, 2017.

Maclaurin, Dougal, Duvenaud, David, Johnson, Matthew, and Adams, Ryan P. Autograd: Reverse-mode differentiation of native Python, 2015. URL `http://github.com/HIPS/autograd`.

Neelakantan, Arvind, Le, Quoc V, and Sutskever, Ilya. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*, 2015.

Olah, Christopher. Neural networks, types, and functional programming, 2015. `http://colah.github.io/posts/2015-09-NN-Types-FP/`.

Osera, Peter-Michael and Zdancewic, Steve. Type-and-example-directed program synthesis. In *PLDI*, volume 50, pp. 619–630. ACM, 2015.

Parisotto, Emilio, rahman Mohamed, Abdel, Singh, Rishabh, Li, Lihong, Zhou, Dengyong, and Kohli, Pushmeet. Neuro-symbolic program synthesis. In *International Conference on Learning Representations (ICLR)*, 2016.

Paszke, Adam, Gross, Sam, Chintala, Soumith, and Chanan, Gregory. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration, 2017a.

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

Real, Esteban, Moore, Sherry, Selle, Andrew, Saxena, Saurabh, Suematsu, Yutaka Leon, Tan, Jie, Le, Quoc V., and Kurakin, Alexey. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*, 2017.

Reed, Scott and de Freitas, Nando. Neural programmer-interpreters. In *International Conference on Learning Representations (ICLR)*, 2016.

Riedel, Sebastian, Bosnjak, Matko, and Rocktäschel, Tim. Programming with a differentiable forth interpreter. *CoRR*, abs/1605.06640, 2016. URL `http://arxiv.org/abs/1605.06640`.

Rusu, Andrei A, Rabinowitz, Neil C, Desjardins, Guillaume, Soyer, Hubert, Kirkpatrick, James, Kavukcuoglu, Koray, Pascanu, Razvan, and Hadsell, Raia. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

Shankar, Asim and Dobson, Wolff. Eager execution: An imperative, define-by-run interface to tensorflow. `https://research.googleblog.com/2017/10/eager-execution-imperative-define-by.html`, 2017.

Socher, Richard, Perelygin, Alex, Wu, Jean Y, Chuang, Jason, Manning, Christopher D, Ng, Andrew Y, and Potts, Christopher. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.

Solar-Lezama, Armando. Program sketching. *STTT*, 15 (5-6):475–495, 2013.

Stallkamp, J., Schlipsing, M., Salmen, J., and Igel, C. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*, (0), 2012.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL `http://arxiv.org/abs/1605.02688`.

Thrun, Sebastian and Mitchell, Tom M. Lifelong robot learning. *Robotics and Autonomous Systems*, 15(1-2): 25–46, 1995.

Tokui, Seiya, Oono, Kenta, Hido, Shohei, and Clayton, Justin. Chainer: a next-generation open source framework for deep learning. In *Proceedings of NIPS Workshop on Machine Learning Systems (LearningSys)*, 2015.

Zhong, Zhao, Yan, Junjie, and Liu, Cheng-Lin. Practical network blocks design with q-learning. *CoRR*, abs/1708.05552, 2017.

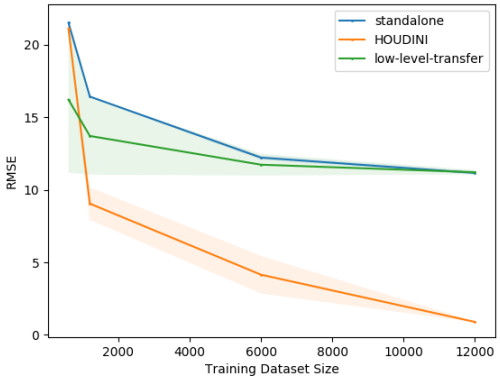
Zoph, Barret and Le, Quoc V. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017.

A. Programs

Tables 4-9 list the top 3 programs and the corresponding classification errors/RMSEs for all the tasks in the evaluated task sequences. Program terms with prefix "nn_" denote neural modules trained during the corresponding tasks whereas terms with prefix "lib." denote already trained neural modules in the library. For example, in Counting Sequence 1 (Table 4), "nn_cs1_1" is the neural module trained during Task 1 (recognize_digit(d_1)). After completion of this task, the neural module is added to the library and is available for use during the subsequent tasks. For example, the top performing program for Task 3 (count_digit(d_1)) uses the neural module "lib.nn_cs1_1" from the library (and a freshly trained neural module "nn_cs1_5") to construct a program for the counting task.

B. Summing Experiment

In this section we present the result from task sequence SS in Figure 3 of the main paper. This sequence was designed to demonstrate low-level transfer of a multi-class classifier as well as the advantage of functional methods like foldl in specific situations. The first task of the sequence is a simple MNIST classifier, on which all competing methods do equally well. The second task is a regression task, to learn to sum all of the digits in the sequence. Neither the standalone (no transfer) method nor low level transfer do very well on this task (note that the two lines are overplotted in the Figure), but the synthesized program from HOUDINI is able to learn this function easily because it is able to model with with a foldl operation.



(a) Task 2: Sum digits

Figure 7. Lifelong learning for "learning to sum" (Sequence SS).

Table 4. Counting Sequence 1(CS1). "CE" denotes classification error and "RMSE" denotes root mean square error.

Task	Top 3 programs	Error
Task 1: recognize_digit(d_1)	1. nn_cs1_1	∼0% CE
Task 2: recognize_digit(d_2)	1. nn_cs1_2	∼0% CE
	2. **compose**(nn_cs1_3, lib.nn_cs1_1)	42% CE
	3. **compose**(**repeat**(8, nn_cs1_4), lib.nn_cs1_1)	49% CE
Task 3: count_digit(d_1)	1. **compose**(**fold_l**(nn_cs1_5, zeros(1)), **map_l**(lib.nn_cs1_1))	0.13 RMSE
	2. **compose**(nn_cs1_6, **map_l**(lib.nn_cs1_1))	0.26 RMSE
	3. **compose**(nn_cs1_7, **map_l**(lib.nn_cs1_2))	2.14 RMSE
Task 4: count_digit(d_2)	1. **compose**(**fold_l**(lib.nn_cs1_5, zeros(1)), **map_l**(nn_cs1_8))	0.09 RMSE
	2. **compose**(**fold_l**(nn_cs1_9, zeros(1)), **map_l**(lib.nn_cs1_2))	0.12 RMSE
	3. **compose**(**fold_l**(lib.nn_cs1_5, zeros(1)), **map_l**(lib.nn_cs1_2))	0.13 RMSE

Table 5. Counting Sequence 2(CS2)

Task	Top 3 programs	Error
Task 1: recognize_digit(d_1)	1. nn_cs2_1	∼0% CE
Task 2: count_digit(d_1)	1. **compose**(**fold_l**(nn_cs2_2, zeros(1)), **map_l**(lib.nn_cs2_1))	0.14 RMSE
	2. **compose**(nn_cs2_3, **map_l**(lib.nn_cs2_1))	0.30 RMSE
Task 3: count_digit(d_2)	1. **compose**(**fold_l**(lib.nn_cs2_2, zeros(1)), **map_l**(nn_cs2_4))	0.08 RMSE
	2. **compose**(**fold_l**(nn_cs2_5, zeros(1)), **map_l**(lib.nn_cs2_1))	2.15 RMSE
	3. **compose**(nn_cs2_6, **map_l**(lib.nn_cs2_1))	2.15 RMSE
Task 4: recognize_digit(d_2)	1. **compose**(nn_cs2_7, lib.nn_cs2_4)	∼0% CE
	2. lib.nn_cs2_4	∼0% CE
	3. nn_cs2_8	∼0% CE

Table 6. Counting Sequence 3(CS3)

Task	Top 3 Programs	Error
Task 1: recognize_digit(d)	1. nn_cs3_1	∼0% CE
Task 2: count_digit(d)	1. **compose**(**fold_l**(nn_cs3_2, zeros(1)), **map_l**(lib.nn_cs3_1))	0.14 RMSE
	2. **compose**(nn_cs3_3, **map_l**(lib.nn_cs3_1))	0.28 RMSE
Task 3: count_toy(t)	1. **compose**(**fold_l**(lib.nn_cs3_2, zeros(1)), **map_l**(nn_cs3_4))	0.54 RMSE
	2. **compose**(**fold_l**(nn_cs3_5, zeros(1)), **map_l**(lib.nn_cs3_1))	2.15 RMSE
	3. **compose**(**fold_l**(lib.nn_cs3_2, zeros(1)), **compose**(**map_l**(nn_cs3_6), **conv_l**(**map_l**(lib.nn_cs3_1))))	2.16 RMSE
Task 4: is_toy(t)	1. nn_cs3_7	2% CE
	2. **compose**(nn_cs3_8, lib.nn_cs3_4)	4% CE
	3. lib.nn_cs3_4	4% CE

Table 7. Summing Sequence(SS)

Task	Top 3 programs	Error
Task 1: classify_digit	1. nn_ss_1	1% CE
Task 2: sum_digits	1. **compose**(**fold_l**(nn_ss_2, zeros(1)), **map_l**(lib.nn_ss_1))	0.88 RMSE
	2. **compose**(nn_ss_3, **map_l**(lib.nn_ss_1))	5.38 RMSE

Table 8. Graph Sequence 1(GS1)

Task	Top 3 programs	Error
Task 1: regress_speed	1. nn_gs1_1	0.39 RMSE
Task 2: shortest_path_street	1. **compose**(**repeat**(9, **conv_g**(nn_gs1_2)), **map_g**(lib.nn_gs1_1))	1.56 RMSE
	2. **compose**(**repeat**(10, **conv_g**(nn_gs1_3)), **map_g**(lib.nn_gs1_1))	1.68 RMSE
	3. **compose**(**conv_g**(nn_gs1_4), **map_g**(lib.nn_gs1_1))	5.51 RMSE

Table 9. Graph Sequence 2(GS2)

Task	Top 3 programs	Error
Task 1: regress_mnist	1. nn_gs2_1	1.35 RMSE
Task 2: shortest_path_mnist	1. **compose**(**repeat**(10, **conv_g**(nn_gs2_2)), **map_g**(lib.nn_gs2_1))	1.66 RMSE
	2. **compose**(**repeat**(9, **conv_g**(nn_gs2_3)), **map_g**(lib.nn_gs2_1))	1.71 RMSE
	3. **compose**(**conv_g**(nn_gs2_4), **map_g**(lib.nn_gs2_1))	4.56 RMSE
Task 3: shortest_path_street	1. **compose**(**repeat**(9, **conv_g**(lib.nn_gs2_2)), **map_g**(nn_gs2_5))	3.44 RMSE
	2. **compose**(**repeat**(10, **conv_g**(lib.nn_gs2_2)), **compose**(**conv_g**(lib.nn_gs2_2), **map_g**(nn_gs2_6)))	3.60 RMSE
	3. **compose**(**repeat**(10, **conv_g**(lib.nn_gs2_2)), **map_g**(nn_gs2_7))	4.96 RMSE