(This is a shortened workshop version of the paper at https://arxiv.org/abs/1608.04428).

TERPRET: A Probabilistic Programming Language for Program Induction

Alexander L. Gaunt[1], **Marc Brockschmidt**[1], **Rishabh Singh**[1], **Nate Kushman**[1], **Pushmeet Kohli**[1], **Jonathan Taylor**[2]*, **Daniel Tarlow**[1]
[1]Microsoft Research [2]perceptiveIO
{t-algaun, mabrocks, risin, nkushman, pkohli, dtartlow}@microsoft.com
jtaylor@perceptiveio.com

Abstract

We study machine learning formulations of inductive program synthesis; that is, given input-output examples, synthesize source code that maps inputs to corresponding outputs. Our key contribution is TERPRET, a domain-specific language for expressing program synthesis problems. A TERPRET model is composed of a specification of a program representation and an interpreter that describes how programs map inputs to outputs. The inference task is to observe a set of input-output examples and infer the underlying program. From a TERPRET model we automatically perform inference using four different back-ends: gradient descent (thus each TERPRET model can be seen as defining a differentiable interpreter), linear program (LP) relaxations for graphical models, discrete satisfiability solving, and the SKETCH program synthesis system. TERPRET has two main benefits. First, it enables rapid exploration of a range of domains, program representations, and interpreter models. Second, it separates the model specification from the inference algorithm, allowing proper comparisons between different approaches to inference.

We illustrate the value of TERPRET by developing several interpreter models and performing an extensive empirical comparison between alternative inference algorithms on a variety of program models. To our knowledge, this is the first work to compare gradient-based search over program space to traditional search-based alternatives. Our key empirical finding is that constraint solvers dominate the gradient descent and LP-based formulations.

1 Introduction

Learning computer programs from input-output examples, or Inductive Program Synthesis (IPS), is a fundamental problem in computer science, dating back at least to Summers [1977] and Biermann [1978]. The field has produced many successes, with perhaps the most visible example being the FlashFill system in Microsoft Excel [Gulwani, 2011, Gulwani et al., 2012].

There has also been significant recent interest in neural network-based models with components that resemble computer programs [Giles et al., 1989, Joulin and Mikolov, 2015, Grefenstette et al., 2015, Graves et al., 2014, Weston et al., 2014, Kaiser and Sutskever, 2016, Reed and de Freitas, 2016, Neelakantan et al., 2016, Kurach et al., 2015, Andrychowicz and Kurach, 2016]. These models combine neural networks with external memory, external computational primitives, and/or built-in structure that reflects a desired algorithmic structure in their execution. However, none produce programs as output. Instead, the program is hidden inside "controllers" composed of neural networks that decide which operations to perform, and the learned program can only be understood in terms of the executions that it produces on specific inputs.

*Work done while author was at Microsoft Research.

1st Workshop on Neural Abstract Machines & Program Induction (NAMPI), @NIPS 2016, Barcelona, Spain.

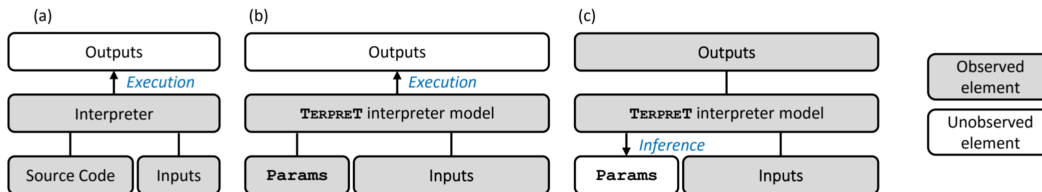


Figure 1: A high level view of the program synthesis task. (a) Forward execution of a traditional interpreter. (b) Forward execution of a TERPRET model. (c) Inference in TERPRET model.

Technique name	Optimizer/Solver	Description
FMGD (Forward marginals, gradient descent)	TensorFlow	A gradient descent based approach which generalizes the approach used by Kurach et al. [2015].
(I)LP ((Integer) linear programming)	Gurobi	A novel linear program relaxation that supports Gates [Minka and Winn, 2009].
SMT	Z3	Translation of the problem into a first-order logical formula with existential constraints.
SKETCH [Solar-Lezama, 2008]	SKETCH	Cast the TERPRET model as a partial program (the interpreter) containing holes (the source code) to be inferred from a specification (input-output examples).

Table 1: TERPRET back-end inference algorithms.

In this work we focus on models that represent programs as simple, natural source code [Hindle et al., 2012], i.e., the kind of source code that people write. There are two main advantages to representing programs as source code rather than weights of a neural network controller. First, source code is interpretable; the resulting models can be inspected by a human and debugged and modified. Second, programming languages are designed to make it easy to express the algorithms that people want to write. By using these languages as our model representation, we inherit inductive biases that can lead to strong generalization (i.e., generalizing successfully to test data that systematically differs from training). Of course, natural source code is likely not the best representation in all cases. For example, programming languages are not designed for writing programs that classify images, and there is no reason to expect that natural source code would impart a favorable inductive bias in this case.

Optimization over program space is known to be a difficult problem. However, recent progress in neural networks showing that it is possible to learn models with differentiable computer architectures, along with the success of gradient descent-based optimization, raises the question of whether gradient descent could be a powerful new technique for searching over program space.

These issues motivate the main questions in this work. We ask (a) whether new models can be designed specifically to synthesize interpretable source code that may contain looping and branching structures, and (b) how searching over program space using gradient descent compares to the combinatorial search methods from traditional IPS.

To address the first question we develop models inspired by intermediate representations used in compilers like LLVM [Lattner and Adve, 2004] that can be trained by gradient descent. These models interact with external storage, handle non-trivial control flow with explicit `if` statements and loops, and, when appropriately discretized, a learned model can be expressed as interpretable source code. We note two concurrent works, Adaptive Neural Compilation [Bunel et al., 2016] and Differentiable Forth [Riedel et al., 2016], which implement similar models.

To address the second question, concerning the efficacy of gradient descent, we need a way of specifying many IPS problems such that the gradient based approach can be compared to alternative approaches in a like-for-like manner across a variety of domains. The alternatives originate from rich histories of IPS in the programming languages and inference in discrete graphical models. To our knowledge, no such comparison has previously been performed.

Both of the above benefit from being formulated in the context of TERPRET. TERPRET provides a means for describing an *execution model* (e.g., a Turing Machine, an assembly language, etc.) by defining a a program representation and an interpreter that maps inputs to outputs using the

```
T = 5
#################################################
#  Source code parametrisation                  #
#################################################
ruleTable = Param(2)[2, 2]

#################################################
#  Interpreter model                            #
#################################################
tape = Var(2)[T]
initial_tape = Input(2)[2]
final_tape = Output(2)
tape[0].set_to(initial_tape[0])
tape[1].set_to(initial_tape[1])

for t in range(1, T - 1):
    with tape[t] as x1:
        with tape[t - 1] as x0:
            tape[t + 1].set_to(ruleTable[x0, x1])

final_tape.set_to(tape[T - 1])
```

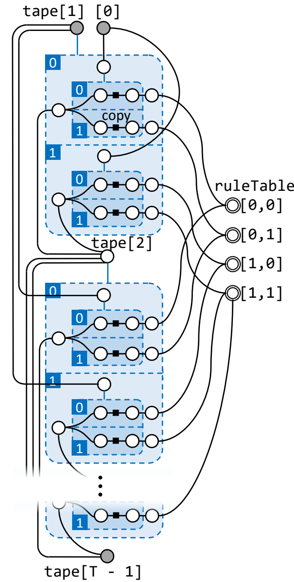


Figure 2: Illustrative example of a TERPRET model and corresponding factor graph which describe a toy automaton that updates a binary `tape` according to the previous two entries and a `rule` (refer to long version for definition of graphical symbols).

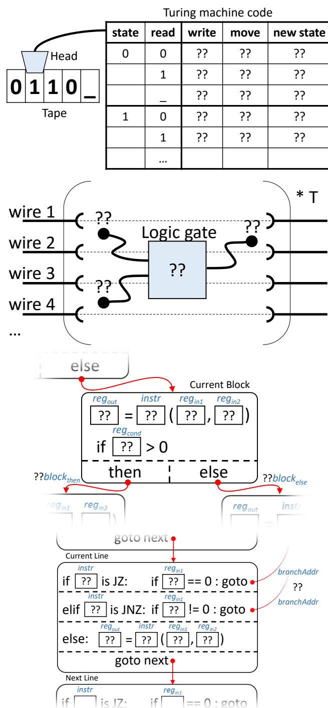
parametrized program. The TERPRET description is independent of any particular inference algorithm. The IPS task is to infer the execution model parameters (the program) given an execution model and pairs of inputs and outputs. An overview of the synthesis task appears in Fig. 1. To perform inference, TERPRET is automatically compiled into an intermediate representation which can be fed to a particular inference algorithm. Table 1 describes the inference algorithms currently supported by TERPRET. Interpretable source code can be obtained directly from the inferred model parameters. The driving design principle for TERPRET is to strike a subtle balance between the breadth of expression needed to precisely capture a range of execution models, and the restriction of expression needed to ensure that automatic compilation to a range of different back-ends is tractable.

2 TERPRET Language

Due to space, we give just a simple example of how TERPRET models are written. For a full grammar of the language and several longer examples, see the long version Gaunt et al. [2016b]. TERPRET obeys Python syntax, and we use the Python `ast` library to parse and compile TERPRET models. The model is composed of `Param` variables that define the program, `Var` variables that represent the intermediate state of computation, and `Inputs` and `Outputs`. Currently, all variables must be discrete with constant sizes. TERPRET supports loops with ranges defined by constants[2], if statements, arrays, array indexing, and user-defined functions that map a discrete set of inputs to a discrete output. In total, the constraints imply that a model can be converted into a gated factor graph [Minka and Winn, 2009]. The details of how this works are in the longer version. A very simple TERPRET model appears in Fig. 2. In this model, there is a binary tape that the program writes to. The program is a rule table that specifies which state to write at the current position of a tape, conditional on the pattern that appeared on the two previous tape locations. The first two tape positions are initialized as Inputs, and the final tape position is the program output.

We have used TERPRET to build much more complicated models, including a Turing Machine, boolean circuits, a Basic Block model that is similar to the intermediate representation used by LLVM, and an assembly-like model.

[2]These constants define maximum quantities, like the maximum number of timesteps that a program can execute for, or the maximum number of instructions that can be in a program. Variable-length behavior can be achieved by defining an absorbing end-state, by allowing a no-op instruction, etc.



TURING MACHINE	Description
Invert	Perfom bitwise inversion of a binary string on the tape.
Prepend zero	Right shift all symbols and insert a “0" in the first cell.
Binary decrement	Decrement a binary representation on the tape by 1.
BOOLEAN CIRCUITS	**Description**
2-bit controlled shift register	Swap bits on wires 2 & 3 iff wire 1 is on.
full adder	Perform binary addition of two bits including carries.
2-bit adder	Perform binary addition on two-bit numbers.
BASIC BLOCK	**Description**
Access	Access the kth element of a contiguous array.
Decrement	Decrement all elements in a contiguous array.
List-K	Access the kth element of a linked list.
ASSEMBLY	**Description**
Access	
Decrement	As above.
List-K	

Table 2: Overview of benchmark problems, grouped by execution model. We illustrate the basic structure of each model, with parameters of the model to be inferred denoted by ??.

3 Experimental Results

Table 2 gives an overview of the benchmark programs which we attempt to synthesize. We created TERPRET descriptions for each execution model, we designed three synthesis tasks per model which are specified by up to 16 input-output examples. We measure the time taken by the inference techniques listed in Table 1 to synthesize a program for each task. Results are summarized in Table 3.

With the exception of the FMGD algorithm, we perform a single run with a timeout of 4 hours for each task. For the FMGD algorithm we run both a Vanilla form and an Optimized form with additional heuristics such as gradient clipping, gradient noise and an entropy bonus (Kurach et al. [2015]) to aid convergence. Even with these heuristics we observe that several random initializations of the FMGD algorithm stall in an uninterpretable local optimum rather than finding an interpretable discrete program in a global optimum. In the Vanilla case we report the fraction of 20 different random initializations which lead to any globally optimal solution consistent with the input-output specification and also the wall clock time for 1000 epochs of the gradient descent algorithm (the typical number required to reach convergence on a successful run). In the Optimized FMGD case, we randomly draw 120 sets of hyperparameters from a manually chosen distribution and run the learning with 20 different random initializations for each setting. We then report the success rate for the best hyperparameters found and also for the average across all runs in the random search.

Our results show that traditional techniques employing constraint solvers (SMT and Sketch) outperform other methods (FMGD and ILP), with SKETCH being the only system able to solve all of these benchmarks before timeout[3]. Furthermore, Table 3 highlights that the precise formulation of the interpreter model can affect the speed of synthesis. Both the Basic Block and Assembly models are equally expressive, but the Assembly model is biased towards producing straight line code with minimal branching. In cases where synthesis was successful the Assembly representation is seen to outperform the Basic Block model in terms of synthesis time.

In addition, we performed a separate experiment to investigate the local optima arising in the FMGD formulation. Using TERPRET we describe the task of inferring the values of a bit string of length K from observation of the parity of neighboring variables. It is possible to show analytically that the number of local optima grow exponentially with K, and Table 4 provides empirical evidence that these minima are encountered in practice and they hinder the convergence of the FMGD algorithm.

[3] Additional (pre)processing performed by SKETCH makes it slower than a raw SMT solver on the smaller tasks but allows it to succeed on the larger tasks

4 Conclusion

We presented TERPRET, a probabilistic programming language for specifying IPS problems. The flexibility of the TERPRET language in combination with the four inference backends allows like-for-like comparison between gradient-based and constraint-based techniques for inductive program synthesis. The primary take-away from the experiments is that constraint solvers outperform other approaches in all cases studied. However, our work is (intentionally) only measuring the ability to efficiently search over program space. We remain optimistic about extensions to the TERPRET framework which allow differentiable interpreters to handle problems involving perceptual data [Gaunt et al., 2016a], and about using machine learning techniques to guide search-based techniques [Balog et al., 2016].

References

Marcin Andrychowicz and Karol Kurach. Learning efficient algorithms with hierarchical attentive memory. *arXiv preprint arXiv:1602.03218*, 2016.

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

Alan W Biermann. The inference of regular lisp programs from examples. *IEEE transactions on Systems, Man, and Cybernetics*, 8(8):585–600, 1978.

Rudy Bunel, Alban Desmaison, Pushmeet Kohli, Philip H. S. Torr, and M. Pawan Kumar. Adaptive neural compilation. *CoRR*, abs/1605.07969, 2016. URL `http://arxiv.org/abs/1605.07969`.

Alexander L Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Lifelong perceptual programming by example. *arXiv preprint arXiv:1611.02109*, 2016a.

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

C. Lee Giles, Guo-Zheng Sun, Hsing-Hen Chen, Yee-Chun Lee, and Dong Chen. Higher order recurrent networks and grammatical inference. In *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*, pages 380–387, 1989.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014. URL `http://arxiv.org/abs/1410.5401`.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pages 1828–1836, 2015.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

Sumit Gulwani, William Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, Aug 2012.

Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*, pages 190–198, 2015.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. In *Proceedings of the 4th International Conference on Learning Representations.*, 2016.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In *Proceedings of the 4th International Conference on Learning Representations 2016*, 2015. URL `http://arxiv.org/abs/1511.06392`.

	$\log_{10}(D)$	T	N	FMGD Time	FMGD Vanilla	FMGD Best Hypers	FMGD Average Hypers	ILP Time	SMT Time	SKETCH Time
TURING MACHINE										
Invert	4	6	5	76.5	100%	100%	51%	0.6	0.7	3.1
Prepend zero	9	6	5	98	60%	100%	37%	17.0	0.9	2.6
Binary decrement	9	9	5	163	5%	25%	2%	191.9	1.6	3.3
BOOLEAN CIRCUITS										
2-bit controlled shift register	10	4	8	-	-	-	-	2.5	0.7	2.7
Full adder	13	5	8	-	-	-	-	38	1.9	3.5
2-bit adder	22	8	16	-	-	-	-	13076.5	174.4	355.4
BASIC BLOCK										
Access	14	5	5	173.8	15%	50%	1.1%	98.0	14.4	4.3
Decrement	19	18	5	811.7	-	5%	0.04%	-	-	559.1
List-K	33	11	5	-	-	-	-	-	-	5493.0
ASSEMBLY										
Access	13	5	5	134.6	20%	90%	16%	3.6	10.5	3.8
Decrement	20	27	5	-	-	-	-	-	-	69.4
List-K	29	16	5	-	-	-	-	-	-	16.8

Table 3: Benchmark results. For FMGD we present the time in seconds for 1000 epochs and the success rate out of {20, 20, 2400} random restarts in the {Vanilla, Best Hypers and Average Hypers} columns respectively. For other back-ends we present the time in seconds to produce a synthesized program. The symbol - indicates timeout ($>$ 4h) or failure of any random restart to converge. N is the number of provided input-output examples used to specify the task in each case.

	$K=4$	$K=8$	$K=16$	$K=32$	$K=64$	$K=128$
Vanilla FMGD	100%	53%	14%	0%	0%	0%
Best Hypers	100%	100%	100%	70%	80%	0%
Average Hypers	84%	42%	21%	4%	1%	0%

Table 4: Percentage of runs that converge to the global optimum for FMGD on the Parity Chain example of length K.

Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

Tom Minka and John Winn. Gates. In *Advances in Neural Information Processing Systems*, pages 1073–1080, 2009.

Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. In *Proceedings of the 4th International Conference on Learning Representations 2016*, 2016.

Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. In *Proceedings of the 4th International Conference on Learning Representations 2016*, 2016.

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

Armando Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Dept., UC Berkeley, 2008.

Phillip D Summers. A methodology for lisp program construction from examples. *Journal of the ACM (JACM)*, 24(1):161–175, 1977.

Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. In *Proceedings of the 3rd International Conference on Learning Representations 2015*, 2014. URL `http://arxiv.org/abs/1410.3916`.