Neural Inference of API Functions from Input–Output Examples

Rohan Bavishi, Caroline Lemieux, Neel Kant, Roy Fox, Koushik Sen, Ion Stoica
Department of Computer Science
University of Califonia, Berkeley, USA
{rbavishi,clemieux,kantneel,royf,ksen,istoica}@cs.berkeley.edu

Abstract

Because of the prevalence of APIs in modern software development, an automated interactive code discovery system to help developers use these APIs would be extremely valuable. Program synthesis is a promising method to build such a system, but existing approaches focus on programs in domain-specific languages with much fewer functions than typically provided by an API. In this paper we focus on 112 functions from the Python pandas library for DataFrame manipulation, an order of magnitude more than considered in prior approaches. To assess the viability of program synthesis in this domain, our first goal is a system that reliably synthesizes programs with a single library function. We introduce an encoding of structured input–output examples as graphs that can be fed to existing graph-based neural networks to infer the library function. We evaluate the effectiveness of this approach on synthesized and real-world I/O examples, finding programs matching the I/O examples for 97% of both our validation set and cleaned test set.

1 Introduction

Program synthesis, the process of automatically generating a program conforming to a higher-level specification, is a long-standing goal of computer science research. General synthesis remains elusive due to the huge search problem it entails — for a language with n functions, taking an average of m argument values, the number of sequential programs of length k grows as $(nm)^k$. The most successful approaches restrict the search space either by imposing user-specified structure, such as program sketches [18], or by only considering programs in a small subset of a general-purpose language or small domain-specific language (*DSL*) [9, 4, 5].

A practical application of program synthesis is the generation of programs in a particular application programming interface (*API*). Nowadays, developers have to contend with a growing number of APIs [2], whose development outpaces the completeness, clarity, and even correctness of the documentation. Thus, when trying to perform a particular operation in an API, developers often consult API experts in online forums, such as StackOverflow, with a description of the operation in terms of input–output (I/O) examples. Our motivation is to take a step towards automating this interaction.

Our goal is to build an I/O-based synthesis algorithm that can infer programs in large, real-world libraries with hundreds of API functions, each taking in a variety of arguments from different domains, that also have dependencies amongst themselves. In this setting, we observed that a purely algorithmic, exhaustive approach, could, given the correct library function(s), efficiently find arguments to fit the I/O example. But, it could not efficiently search for the function(s) to use. Instead of spending years tuning the exhaustive approach's search heuristics [18, 9], we decided to take a hybrid approach [10, 15], which relies on a neural inference mechanism to predict the function(s) and on exhaustive search to find argument values in the complex argument space. The library we target in this work is pandas [1], a library of transformations of DataFrames, which are tabular data structures. pandas is used extensively in machine learning and data science applications.
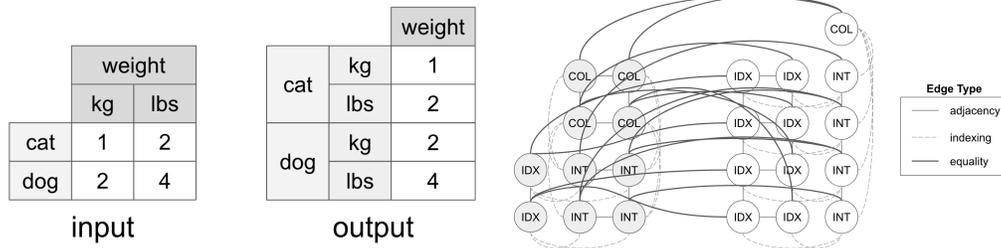
(a) DataFrame I/O Example. White cells are data; pale gray are row indices, and dark grey are column names.

(b) Graph Representation. Gray nodes come from the input DataFrame, white nodes from the output.

Figure 1: A DataFrame input-output example and its graph abstraction, corresponding to the operation `output = input.stack(level=[1], dropna=True)`.[1] From the graph, our network predicts `stack` with 99% confidence.

In this work, we investigate the viability of synthesis in this domain by restricting ourselves to a simpler problem: synthesize `pandas` programs consisting of a single function call. In order to do this, we present a technique to *encode* the I/O example into a neural-network interpretable format and a network which can predict the functions to use from this encoding. Unlike the domain elements of prior work [16, 7, 4, 5], our I/O domain elements, DataFrames—2D structures which can contain arbitrary Python objects—are not easily encodable into a fixed-size format. We propose a graph-based encoding of DataFrame I/O examples and a network architecture based on GGNN [12] to predict the `pandas` function. We will extend the system to synthesize longer `pandas` programs in future work.

We evaluate our technique by measuring its top-1 and top-5 prediction accuracy on a synthetic validation set and a set of real-world I/O examples collected from StackOverflow, the book *Python for Data Analysis*, and the `pandas` Data School video series. We find that our network achieves high top-5 accuracy in both cases, but that generalizing to real-world data requires data cleaning. Finally, we present an ablation study on the features of our graph-based abstraction.

Related Work   Neuro-Symbolic Program Synthesis [16] and RobustFill [7] both use neural inference in the FlashFill scenario of string transformations in a DSL. Bunel et al. [5] synthesize programs in the Karel DSL. DeepCoder [4] uses neural inference to determine the program elements most likely to solving an input-output example. They focus on a small functional language with 17 higher-order functions and 17 other program elements. In all these cases, the neural inference technique gives a ranking or probability over the entire set of program features — functions and arguments. This cannot scale in our case where the pool of valid arguments for the 112 `pandas` functions can be of size 5000 for a 4x4 input DataFrame.

While Morpheus [8] targets the same space as our work—DataFrame transformations—, it considers only 10 R functions, and since it uses an SMT solver to prune the search space, it does not provide a way to encode the I/O examples. The cross-correlational networks, LSTMs, and LSTMs with attention used to encode string-based examples and lists of bounded integers do not apply to DataFrames [16, 10, 7, 4]. Since the DataFrames are not of a fixed size, the CNNs used to encode fixed-size grid-based examples [5] also do not apply.

2   Method

The goal of the method described here is to map a given I/O example to a `pandas` function which performs the transformation specified by the example. At a high-level, our method works by (1) preprocessing I/O examples into a graph, (2) feeding these examples into a trainable neural network which learns a high-dimensional representation for each node of the graph, and (3) pooling to output of the neural network and applying softmax to select a `pandas` function. After this, we use exhaustive search to find the correct arguments; those details are beyond the scope of this paper.

2.1   Graph Abstraction

The input-output domain of `pandas` transformations is the set of DataFrames, 2D structures which can contain arbitrary Python objects as primitive elements. We noticed that the operation used in an

---

[1] https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.stack.html

I/O example is often captured by the relationships amongst the elements, rather than the concrete data itself. Therefore, we use a *graph-based abstraction* of an I/O example pair which *does not bound* the space of primitive components which make up the DataFrame. Figure 1 shows an I/O example (Figure 1a) and the graph outputted by the process (Figure 1b). The graph is constructed as follows.

Nodes. Every data cell in the input and output DataFrame is represented as a single node. The schemas are represented by nodes for each row index and column name of the DataFrame. Multiple levels of column names or row indices appear as additional nodes—in Figure 1, note that the two layers of column names in the input DataFrame become four distinct nodes in the graph.

Instead of containing concrete values, a node is labeled with a type tuple *(data type, is input)* which designates its data type and whether it belongs to the input. The data type is either the concrete type of the value (integer, float, string, etc.) for cells, or a special data type for row indices and column names. In Figure 1, the shaded nodes represent nodes from input DataFrames and the white nodes represent nodes from output DataFrame, with the string label designating the data type of the node.

Edges. Since we abstract away the concrete values, we add edges to represent the relationships between them, which are key to identifying the transformation applied. First we add *equality edges* between any nodes with the same value: we hypothesize that these edges are enough to distinguish a large subset of `pandas` functions. These can exist between column name, row index, and data nodes. For example, in Figure 1b, an edge exists between the *kg* column name from the input and the two *kg* row indices in the output.

We also add edges that represent the basic structural characteristics of the DataFrames. We add adjacency edges, thin solid lines in Figure 1b, between two data cells, column name cells, or row index cells are adjacent to each other (diagonals do not count). We add indexing edges, thin dotted lines in Figure 1b, between a column name (resp. row index) and all the data nodes that belong to that column (resp. row).

2.2   Network

To predict the function label associated with the I/O example, we use a gated graph neural network, introduced by Li et al. [12]. We adjust the implementation by Microsoft [14, 3] to output label class probabilities rather than estimates for regression. The input to our network is a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{X})$ where $\mathcal{V}$ is the set of nodes and $\mathcal{X}$ is a mapping from nodes to node features $\mathcal{X} : \mathcal{V} \to \mathbb{R}^D$ for hyper-parameter D. Our $\mathcal{X}$ maps nodes to the one-hot encoding of a label from a defined set of labels. An edge $e \in \mathcal{E}$ is a 3-tuple (v_s, v_t, t_e) where v_s and v_t are the source and target nodes and t_e is the type of the edge. In our setting, $t_e \in \{adjacency, indexing, equality\}$.

Every node v has a corresponding state vector $h^{(v)} \in \mathbb{R}^D$, initialized as the feature vector of that node. Information is propagated using *message passing* across k rounds. For any edge (v_s, v_t, t_e), v_s sends to v_t the message $m_{v_s} = f_k(h^{(v_s)}, t_e)$. In our case, f_k is a single linear layer. For each node v_t, the incoming messages are aggregated as $m_t = g(\{m_{v_s} | (v_s, v_t, t_e) \in \mathcal{E}\})$. We use an element-wise mean for g. The new node state vector $\hat{h}^{(v_t)}$, for the next round is computed as $\hat{h}^{(v_t)} = GRU(m_t, h^{(v_t)})$ where GRU is the gated recurrent unit [6]. We use two rounds of message passing, as we noticed that increasing the number of message passing rounds did not increase validation accuracy.

After the message passing, we element-wise sum-pool the node state vectors $h^{(v)}$ into a graph state vector h. We pass h though a multi-layer perceptron with one hidden layer, and apply softmax activation on the output layer to produce a probability distribution over the target classes: the `pandas` functions. We train the network with the ADAM optimizer [11] on cross-entropy loss.

3   Evaluation

In order to train our network, we require a large amount of *(I/O example, function)* pairs. In the absence of a large, standardized dataset of such pairs, we synthesize a training set of size $N = 10^6$ and a validation set $N = 10^5$. To have class-balanced training and validation data, we select each function N/k number of times where k is the total number of functions.

For each function, we create its DataFrame arguments by generating a random DataFrame generator. This generator chooses a random DataFrame size, column names, and row indices, then populates each column with random values of either the integer, float, or string type. It allows some amount of duplication of values to model categorical columns or otherwise repeated values. For the other arguments, we rely on the hand-built *argument generators* which are used in the exhaustive argument

Table 1: Accuracy in predicting the ground-truth or a correct function for I/O examples.

	Ground-Truth		Success Rate	
	Top-1	*Top-5*	*Top-1*	*Top-5*
Validation	65%	94%	82%	97%
Test	59%	83%	69%	83%
Clean Test	66%	97%	83%	97%

Table 2: Effect of graph abstraction features on Top-1 validation accuracy.

Control	Acc.
No Node Features	57%
No Edge Features	63%
No Structural Edges	61%
No Equality Edges	46%

search part of the synthesis task. These argument generators are programs which return values in the space of valid argument combinations; we built these generators by consulting the `pandas` documentation, and, when it was lacking, observing the dynamic behavior the the given function on some arbitrary DataFrames.

With the function and the synthesized arguments in hand, we create the I/O example by running the function to produce the output. We discard the example if there are execution errors. We then trained the network on 10^6 examples generated in this manner, with encoding of the input/output DataFrames (Section 2.1) as the input to the network, and the name of the function as the output class/label.

3.1 Accuracy Results

We compute the accuracy of our network in predicting the function used for (1) our synthesized validation set and (2) I/O examples taken from real-world sources, including questions on StackOverflow and examples taken from an introductory `pandas` book [13] and video series [17]. Table 1 presents the accuracy for the prediction task. Overall, we see that the correct function is generally in the top-5 predictions, a very good result for the synthesis task over 112 functions.

The ground-truth accuracy, on the left-hand-side of Table 1, is conservative in that the higher-ranked predictions may in fact fit the I/O example. This is especially common in `pandas` — a number of functions have overlapping semantics. Therefore we also present the *Success Rate*—calculated by checking whether some argument combination of a top-k predicted function can produce the output—in right-hand columns of Table 1. The top-1 accuracy using this metric is significantly higher for both the datasets, suggesting that many "misclassifications" by our network are in fact correct classifications from a functional standpoint.

Additionally, we noticed a large gap in accuracy between our validation set and our test set. We hypothesized that the issue was, in many cases, due to the presence of a large number of *spurious equality edges*. These are equality edges that exist between nodes whose values do not actually influence each other. For example, in Figure 1b, the edge between the data cell (0,1) from the input and the data cell (2,0) in the output. So, we manually edited the test set examples to remove spurious equality edges. The last row of Table 1 presents the accuracy on this cleaned test set. The results are much better, in line with the numbers on the validation dataset. This suggests that the network generalizes to real-world examples, but is sensitive to the noise introduced by our graph abstraction.

3.2 Ablation Study

We also evaluate the effects of removing node features, edge features, structural edges and equality edges on the accuracy on our validation set. Table 2 shows the results. We see that the presence of equality edges has the largest impact on the accuracy, while that of structural edges or edge types is small. This suggests that the behavior of a function is primarily characterized by the connections between nodes introduced by equality edges, as they transcend the boundary between input and output nodes. The removal of node features has a relatively smaller but significant impact on the accuracy, suggesting that data-type information is also vital in distinguishing between functions.

4 Conclusion & Future Work

We presented promising results on a model for predicting `pandas` API functions given an I/O example, which is part of our larger synthesis project targeting large APIs. In the future, we intend to improve our graph abstraction by refining equality edges, and adding other kinds of edges that may be able to distinguish between computational functions such as `add`. We also plan to develop a smarter dataframe-to-graph encoder and incorporate attention mechanisms in our network to counter the effect of noise in the I/O examples (Section 3.1). Finally, we also want to be able to predict sequences containing more than one `pandas` function to synthesize more complex programs.

References

[1] The pandas project. `https://pandas.pydata.org`, 2014. Accessed October 11th, 2018.

[2] a16z Podcast. The API Economy – The Why, What, and How. https://a16z.com/2018/03/13/api-economy-why-what-how/, 2018. Accessed October 22nd, 2018.

[3] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[4] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. DeepCoder: Learning to Write Programs. *CoRR*, abs/1611.01989, 2016.

[5] R. Bunel, M. J. Hausknecht, J. Devlin, R. Singh, and P. Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *CoRR*, abs/1805.04276, 2018.

[6] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734. Association for Computational Linguistics, 2014.

[7] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli. RobustFill: Neural Program Learning under Noisy I/O. In *ICML 2017*, March 2017.

[8] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. *SIGPLAN Not.*, 52(6):422–436, June 2017.

[9] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[10] A. Kalyan, A. Mohta, O. Polozov, D. Batra, P. Jain, and S. Gulwani. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. *ArXiv e-prints*, Apr. 2018.

[11] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *ArXiv e-prints*, Dec. 2014.

[12] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel. Gated Graph Sequence Neural Networks. *CoRR*, abs/1511.05493, 2015.

[13] W. McKinney. *Python for Data Analysis*. O'Reilly, 2012.

[14] Microsoft. Gated Graph Neural Network Samples. https://github.com/Microsoft/gated-graph-neural-network-samples, 2017. Accessed October 17th, 2018.

[15] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine. Neural Sketch Learning for Conditional Program Generation. *ArXiv e-prints*, Mar. 2017.

[16] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. Neuro-Symbolic Program Synthesis. In *ICLR 2017*, February 2017.

[17] D. School. Easier data analysis in Python with pandas (video series). https://www.dataschool.io/easier-data-analysis-with-pandas/, 2016. Accessed October 22nd, 2018.

[18] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM.