From Programs to Interpretable Deep Models and Back

Eran Yahav(✉)

Technion, Haifa, Israel
yahave@cs.technion.ac.il

Abstract. We demonstrate how deep learning over programs is used to provide (preliminary) augmented programmer intelligence. In the first part, we show how to tackle tasks like code completion, code summarization, and captioning. We describe a general path-based representation of source code that can be used across programming languages and learning tasks, and discuss how this representation enables different learning algorithms. In the second part, we describe techniques for extracting interpretable representations from deep models, shedding light on what has actually been learned in various tasks.

1 Introduction

We describe a journey from programs to interpretable deep models, and back. First, we show how to apply neural networks to learn interesting facts about programs, and build (interpretable) models for several programming-related tasks. Then, we show how to extract finite-state automata from a given recurrent neural network, providing some insight on what a network has actually learned.

1.1 Motivating Tasks

Semantic Labeling of Code Snippets. Consider the code snippet of Figure 1. This snippet only contains low-level assignments to arrays, but a human reading the code may (correctly) label it as performing the *reverse* operation. Our goal is to be able to predict such labels automatically. The right hand side of Fig. 1 shows the labels predicted automatically using our approach. The most likely prediction (77.34%) is *reverseArray*. Alon et al. [3] provide additional examples.

Intuitively, this problem is hard because it requires *learning a correspondence* between the *entire content of a code snippet* and a semantic label. That is, it requires aggregating possibly hundreds of expressions and statements from the snippet into a single, descriptive label.

E. Yahav—Joint work with Uri Alon, Yoav Goldberg, Omer Levy, Gail Weiss, and Meital Zilberstein.

© The Author(s) 2018
H. Chockler and G. Weissenbacher (Eds.): CAV 2018, LNCS 10981, pp. 27–37, 2018.
https://doi.org/10.1007/978-3-319-96145-3_2

```
String[] f(final String[] array) {
 final String[] newArray = new String[array.length];
 for (int index = 0; index < array.length; index++) {
  newArray[array.length - index - 1] = array[index];
 }
 return newArray;
}
```

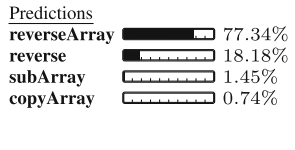


Fig. 1. A code snippet and its predicted labels as computed by our model.

```
iTextSharp.text.pdf.PdfReader reader = new iTextSharp.text.pdf.PdfReader(
    new iTextSharp.text.pdf.RandomAccessFileOrArray(@"C:\PDFFile.pdf"), null);
```

Prediction: *get the text of a pdf file in C#*

Fig. 2. A code snippet and its predicted caption as computed by our model.

Captioning Code Snippets. Consider the short code snippet of Fig. 2. The goal of *code captioning* is to assign a natural language caption that captures the task performed by the snippet. For the example of Fig. 2 our approach automatically predicts the caption *"get the text of a pdf file in C#"*. Intuitively, this task is harder than semantic labeling, as it requires the generation of a natural language sentence in addition to capturing (something about) the meaning of the code snippet.

```
OkHttpClient ok = new OkHttpClient();
Request request = new Request.Builder().url("programming.ai").build();
Response response =
```

Prediction: `ok.newCall(request).execute()`

Fig. 3. A code snippet and its predicted completion as computed by our model.

Code Completion. Consider the code of Fig. 3. Our code completion automatically predicts the next steps in the code: `ok.newCall(request).execute()`. This task requires prediction of the missing part of the code based on a given context. Technically, this can be expressed as predicting a completion of a partial abstract syntax tree.

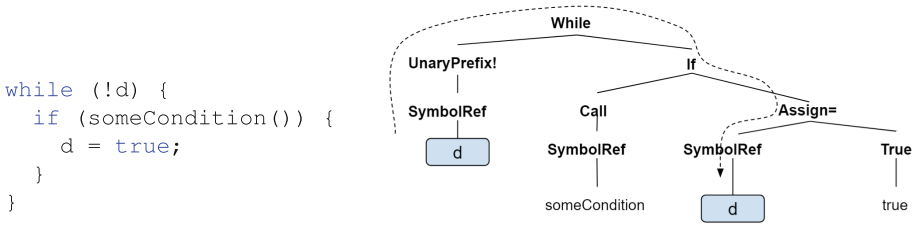
In the next section, we show how techniques based on neural networks address all of these tasks, as well as other programming-related tasks.

2 From Programs to Deep Models

2.1 Representation

Leveraging machine learning models for predicting program properties such as variable names, method names, and expression types is a topic of much recent interest [1,2,6,8,9]. These techniques are based on learning a statistical model from a large amount of code and using the model to make predictions in new programs. A major challenge in these techniques is how to represent instances of the input space to facilitate learning [10]. Designing a program representation that enables effective learning is a critical task that is *often done manually for each task and programming language.*

Our Approach. We present a program representation for learning from programs. Our approach uses different *path-based abstractions of the program's abstract syntax tree.* This family of path-based representations is natural, general, fully automatic, and works well across different tasks and programming languages.



(a) A simple JavaScript program. **(b)** The program's AST, and a path.

Fig. 4. A JavaScript program and its AST, along with an example of one of the paths.

AST Paths. We define AST paths as paths between nodes in a program's abstract syntax tree (AST). To automatically generate paths, we first parse the program to produce an AST, and then extract paths between nodes in the tree. We represent a path in the AST as a sequence of nodes connected by up and down movements, and represent a program element as the set of paths that its occurrences participate in. Figure 4a shows an example JavaScript program. Figure 4b shows its AST, and one of the extracted paths. The path from the first occurrence of the variable `d` to its second occurrence can be represented as:

$$\text{SymbolRef} \uparrow \text{UnaryPrefix!} \uparrow \text{While} \downarrow \text{If} \downarrow \text{Assign=} \downarrow \text{SymbolRef}$$

This is an example of a pairwise path between leaves in the AST, but in general the family of path-based representations contains n-wise paths, which

do not necessarily span between leaves and do not necessarily contain all the nodes in between. We consider several choices of subsets of this family in [4].

Using a path-based representation has several major advantages:

1. Paths are generated automatically: there is no need for manual design of features aiming to capture potentially interesting relationships between program elements. This approach extracts unexpectedly useful paths, without the need for an expert to design features. The user is required only to choose a subset of our proposed family of path-based representations.
2. This representation is useful for any programming language, without the need to identify common patterns and nuances in each language.
3. The same representation is useful for a variety of prediction tasks, by using it with off-the-shelf learning algorithms or by simply replacing the representation of program elements in existing models (as we show in [4]).
4. AST paths are purely syntactic, and do not require any semantic analysis.

2.2 Code2vec: Learning Code Embeddings

In [3], we present a framework for predicting program properties using neural networks. The main idea is a neural network that learns *code embeddings* - continuous distributed vector representations for code. The code embeddings allow us to model correspondence between code snippet and labels in a natural and effective manner. By learning code embeddings, our long term goal is to enable the application of neural techniques to a wide-range of programming-languages tasks. A live demo of the framework is available at https://code2vec.org.

Our neural network architecture uses a representation of code snippets that *leverages the structured nature of source code*, and learns to aggregate multiple syntactic paths into a single vector. This ability is fundamental for the application of deep learning in programming languages. By analogy, word embeddings in natural language processing (NLP) started a revolution of application of deep learning for NLP tasks.

The input to our model is a code snippet and a corresponding tag, label, caption, or name. This tag expresses the semantic property that we wish the network to model, for example: a tag, name that should be assigned to the snippet, or the name of the method, class, or project that the snippet was taken from. Let $\mathcal{C}$ be the code snippet and $\mathcal{L}$ be the corresponding label or tag. Our underlying hypothesis is that *the distribution of labels can be inferred from syntactic paths in* $\mathcal{C}$. Our model therefore attempts to learn the tag distribution, conditioned on the code: $P(\mathcal{L}|\mathcal{C})$.

Model. For the full details of the model, see [3]. At a high-level, the key point is that a code snippet is composed of a bag of contexts, and each context is represented by a vector that its values are learned. The values of this vector capture two distinct goals: (i) the semantic meaning of this context, and (ii) the amount of attention this context should get.

The problem is as follows: given an arbitrarily large number of context vectors, we need to aggregate them into a single vector. Two trivial approaches

would be to learn the most important one of them, or to use them all by vector-averaging them. These alternatives are shown to yield poor results (see [3]).

Our main observation is that *all* context vectors need to be used, but the model should learn how much focus to give each vector. This is done by learning how to average context vectors in a weighted manner. The weighted average is obtained by weighting each vector by its dot product with another global attention vector. The vector of each context and the attention vector are trained and learned *simultaneously*, using the standard neural approach of backpropagation.

Interpreting Attention. Despite the "black-box" reputation of neural networks, our model is partially interpretable thanks to the attention mechanism, which allows us to visualize the distribution of weights over the bag of path-contexts. Figures 5 and 6 illustrates a few predictions, along with the path-contexts that were given the most attention in each method. The width of each of the visualized paths is proportional to the attention weight that it was allocated. We note that in these figures the path is represented only as a connecting line between tokens, while in fact it contains rich syntactic information which is not expressed properly in the figures.

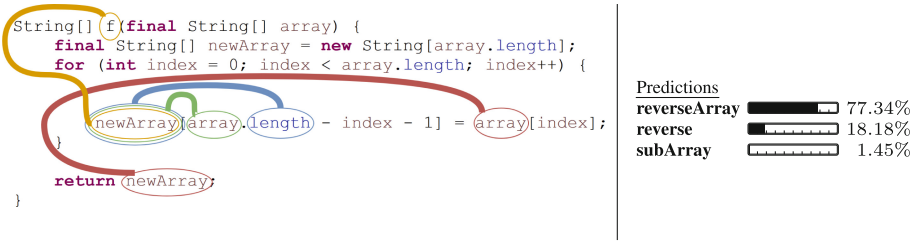


Fig. 5. Predictions and attention paths for the program of Fig. 1. The width of a path is proportional to its attention.

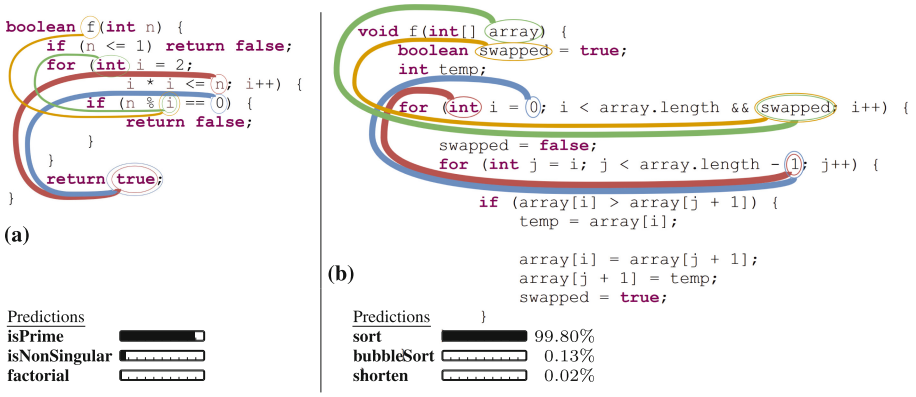


Fig. 6. Example predictions from our model. The width of a path is proportional to its attention.

The examples of Figs. 5 and 6 are interesting since the top names are accurate and descriptive (`reverseArray` and `reverse`; `isPrime`; `sort` and `bubbleSort`) but do not appear explicitly in the code snippets. The code snippets, and specifically the most attended path-contexts describe lower-level operations. Suggesting a descriptive name for each of these methods is difficult and might take time even for a trained human programmer.

2.3 Code2seq: Generating Sequences from Structured Representations of Code

In contrast to classical (and widespread) seq2seq models for translation, we introduce a new model that performs encoding over source code, and decoding to natural language.

Following [3,4], we introduce an approach for encoding source code that leverages the unique syntactic structure of programming languages. We represent a given code snippet as a set of paths over its abstract syntax tree (AST), where each path is compressed to a fixed-length vector. During decoding, code2seq attends over a different weighted sum of the path-vectors to produce each output token, much like NMT models attend over contextualized token representations in the source sentence. A live demo of the framework is available at https://code2seq.org.

3 From Deep Models to Automata

In this section, we focus on extraction of finite-state automata from recurrent neural networks (RNNs). In recent years, there has been significant interest in the use of recurrent neural networks (RNNs), for learning languages. Like other supervised machine learning techniques, RNNs are trained based on a large set of examples of the target concept. While neural networks can reasonably approximate a variety of languages, and even precisely represent a regular language [5], they are in practice unlikely to generalize exactly to the concept being trained, and *what they eventually learn in actuality is unclear* [7]. Our goal in this work is to provide some insight into what a given trained network has actually learned, without requiring changes to the network architecture, or access to the original training data.

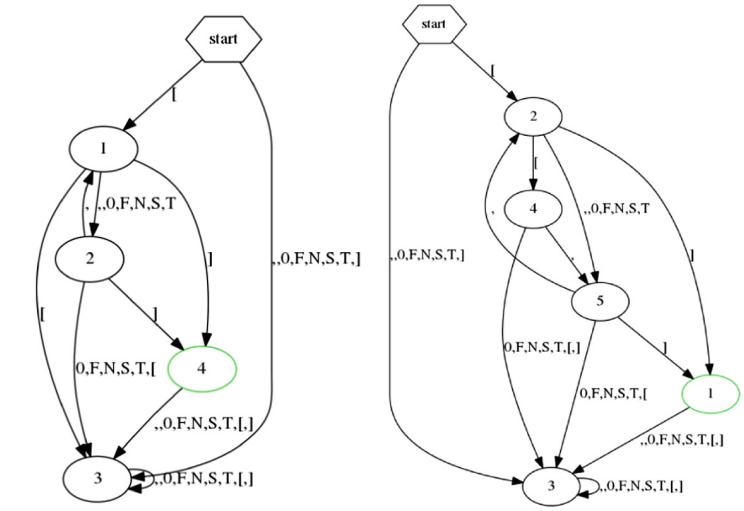
Recurrent Neural Networks. Recurrent neural networks (RNNs) are a class of neural networks which are used to process sequences of arbitrary lengths. When operating over sequences of discrete alphabets, the input sequence is fed into the RNN on a symbol-by-symbol basis. For each input symbol the RNN outputs a *state vector* representing the sequence up to that point. A state vector and an input symbol are combined for producing the next state vector. The RNN is essentially a parameterized mathematical function that takes as input a state vector and an input vector, and produces a new state vector. The state vectors can be passed to a classification component that is used to produce a binary or multi-class classification decision. The RNN is trainable, and, when

trained together with the classification component, the training procedure drives the state vectors to provide a representation of the prefix which is informative for the classification task being trained. We call a combination of an RNN and a classification component an *RNN-acceptor*.

A trained RNN-acceptor can be seen as a state machine in which the states are high-dimensional vectors: it has an initial state, a well defined transition function between internal states, and a well defined classification for each internal state.

Problem Definition. Given an RNN-acceptor R trained to accept or reject sequences over an alphabet Σ, our goal is to extract a deterministic finite-state automaton (DFA) A that mimics the behavior of R. That is, our goal is to extract a DFA A such that the language $L \subseteq \Sigma^*$ of sequences accepted by A is observably equivalent to that accepted by R. Intuitively, we would like to obtain a DFA that accepts *exactly* the same language as the network, but this is generally practically impossible as we do not know in advance any bound on the maximum sample length necessary in order to observe all of its behavior.

Extraction Using Queries and Counterexamples. In [11], we present a framework for extracting a finite state automaton from a given RNN. The main idea is to use the L^* learning algorithm to learn an automaton while using the RNN as the teacher.



(a) (b)

Fig. 7. Two DFAs resembling, but not perfectly, the correct DFA for the regular language of tokenised JSON lists, (\[\])|(\[[S0NTF](,[S0NTF])∗\])$. DFA (a) is almost correct, but accepts also list-like sequences in which the last item is missing, i.e. there is a comma followed by a closing bracket. DFA (b) is returned by L* after the teacher (network) rejects (a), but is also not a correct representation of the target language—treating the sequence `[,` as a legitimate list item equivalent to the characters `S`, `0`, `N`, `T`, `F`.

3.1 What Has a Network Learned?

Tokenized JSON Lists. We trained a GRU network with 2 layers and hidden size 100 on the regular language representing a simple tokenized JSON list with no nesting,

$$(\backslash[\backslash])|(\backslash[[\mathrm{S0NTF}](,[\mathrm{S0NTF}])*\backslash])\$$$$

over the 8-letter alphabet {`[`, `]`, `S`, `0`, `N`, `T`, `F`, `,`}, to accuracy 100% on a training set of size 20000 and a test set of size 2000, both evenly split between positive and negative examples. As before, we extracted from this network using our method.

Within 2 counterexamples (1 provided and 1 generated), our method extracted the automaton shown in Fig. 7a, which is almost but not quite representative of the target language. A few seconds later it returned a counterexample to this DFA which pushed L^* to refine further and return the DFA shown in Fig. 7b, which is also almost but not quite representative of zero-nesting tokenized JSON lists.

Ultimately after 400 s, our method extracted (but did not reach equivalence on) an automaton of size 441, returning the counterexamples listed in Table 1 and achieving 100% accuracy against the network on both its train set and all

Table 1. Counterexamples returned to the equivalence queries made by L^* during extraction of a DFA from a network trained to 100% accuracy on both train and test sets on the regular language $(\backslash[\backslash])|(\backslash[[\mathrm{S0NTF}](,[\mathrm{S0NTF}])*\backslash])\$$ over the 8-letter alphabet {`[`, `]`, `S`, `0`, `N`, `T`, `F`, `,`}. Counterexamples highlighting the discrepancies between the network behaviour and the target behaviour are shown in bold.

Counterexample generation for the non-nested tokenized JSON-lists language			
Counterexample	Generation time (seconds)	Network classification	Target classification
[]	provided	True	True
[SS]	3.49	False	False
[[,]	7.12	True	False
[S,,	8.61	True	False
[0, F	8.38	True	False
[N, 0,	8.07	False	False
[S, N, 0,	9.43	True	False
[T, S,	9.56	False	False
[S, S, T, []	15.15	False	False
[F, T, [	3.23	False	False
[N, F, S, 0	10.04	True	False
[S, N, [,,,,	27.79	True	False
[T, 0, T,	28.06	True	False
[S, T, 0,],	26.63	True	False

sampled sequence lengths. As before, we note that each state split by the method is justified by concrete inputs to the network, and so the extraction of a large DFA is a sign of the inherent complexity of the learned network behavior.

3.2 Counterexamples

For many RNN-acceptors that train to 100% accuracy and exhibit perfect test set behavior on large test sets, our method was able to find many simple examples which the network misclassifies.

For instance, for a network trained to classify simple email addresses over the 38-letter alphabet {a,b,...,z,0,1,...,9,@,.} as defined by the regular expression

$$\text{[a-z][a-z0-9]*@[a-z0-9]+.(com|net|co.[a-z][a-z])\$}$$

with 100% accuracy on a 40,000 sample train set and 100% accuracy on a 2,000 sample test set (i.e., a seemingly perfect network), the refinement-based L^* extraction quickly returned several counterexamples, showing words that the network classifies incorrectly (e.g., the network accepted the non-email sequence `25.net`). While we could not extract a representative DFA from the network in the allotted time frame, our method did show that the network learned a far more elaborate (and incorrect) function than needed.

Beyond demonstrating the counterexample generation capabilities of our extraction method, these results also highlight the brittleness in generalization of trained RNN networks, and suggests that evidence based on test-set performance should be taken with extreme caution.

4 Conclusion

We provide a brief description of a journey from programs to (somewhat) interpretable deep models that work well across different tasks and different programming languages. As we gained experience with these models, the question of *what have they actually learned* became more important (and subtle). Attention over AST paths provides some insight on what drives the predictions performed by (some of) the models, but a different approach is required for RNN-based models. This motivated the second part of our journey, trying to extract an interpretable model from a given RNN acceptor. This also motivated future work on classifying what can and cannot be learned by different kinds of RNNs [12].

References

1. Allamanis, M., Barr, E.T., Bird, C., Sutton, C.: Suggesting accurate method and class names. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 38–49. ACM, New York (2015). http://doi.acm.org/10.1145/2786805.2786849

2. Allamanis, M., Peng, H., Sutton, C.A.: A convolutional attention network for extreme summarization of source code. In: Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, 19–24 June 2016, pp. 2091–2100 (2016). http://jmlr.org/proceedings/papers/v48/allamanis16.html
3. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: learning distributed representations of code. arXiv preprint arXiv:1803.09473 (2018)
4. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: A general path-based representation for predicting program properties. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018 (2018)
5. Casey, M.: Correction to proof that recurrent neural networks can robustly recognize only regular languages. Neural Comput. **10**(5), 1067–1069 (1998). https://doi.org/10.1162/089976698300017340
6. Maddison, C.J., Tarlow, D.: Structured generative models of natural source code. In: Proceedings of the International Conference on Machine Learning, ICML 2014, vol. 32, pp. II-649–II-657. JMLR.org (2014). http://dl.acm.org/citation.cfm?id=3044805.3044965
7. Omlin, C.W., Giles, C.L.: Symbolic knowledge representation in recurrent neural networks: insights from theoretical models of computation. In: Cloete, I., Zurada, J.M. (eds.) Knowledge-Based Neurocomputing, pp. 63–116. MIT Press, Cambridge (2000). http://dl.acm.org/citation.cfm?id=337224.337236
8. Raychev, V., Bielik, P., Vechev, M.: Probabilistic model for code with decision trees. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, pp. 731–747. ACM, New York (2016). http://doi.acm.org/10.1145/2983990.2984041
9. Raychev, V., Vechev, M., Krause, A.: Predicting program properties from "big code". In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, pp. 111–124. ACM, New York (2015). http://doi.acm.org/10.1145/2676726.2677009
10. Shalev-Shwartz, S., Ben-David, S.: Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press, New York (2014)
11. Weiss, G., Goldberg, Y., Yahav, E.: Extracting automata from recurrent neural networks using queries and counterexamples (2017). http://arxiv.org/abs/1711.09576
12. Weiss, G., Goldberg, Y., Yahav, E.: On the practical computational power of finite precision RNNs for language recognition. In: 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018 (2018)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.