2018 IEEE 38th International Conference on Distributed Computing Systems

Software-defined Software: A Perspective of Machine Learning-based Software Production

Rubao Lee, Hao Wang, and Xiaodong Zhang
Department of Computer Science and Engineering, The Ohio State University
{liru, wangh, zhang}@cse.ohio-state.edu

***Abstract*—As the Moore's Law is ending, and increasingly high demand of software development continues in the human society, we are facing two serious challenges in the computing field. First, the general-purpose computing ecosystem that has been developed for more than 50 years will have to be changed by including many diverse devices for various specialties in high performance. Second, human-based software development is not sustainable to respond the requests from all the fields in the society. We envision that we will enter a time of developing high quality software by machines, and we name this as Software-defined Software (SDS). In this paper, we will elaborate our vision, the goals and its roadmap.**

I. Introduction: Two Crises of Future Software Production

Software production (i.e. programming) has started after the birth of modern electronic computers, both of which have created a large and powerful CPU-based computing ecosystem. However, the real driving force behind this ecosystem, the Moore's Law [1] along with the Dennard Scaling Law [2], are ending due to the physical limit [3]. We are entering an era with a dramatic change to evolve into a new computing ecosystem where a variety of highly parallel, highly customized hardware accelerators co-exist with general-purpose processors, such as vector instruction hardware (e.g., Intel AVX) and NVIDIA GPUs. Unfortunately, the traditional programming approach in the CPU-based ecosystem are facing two fundamental challenges in the post Moore's law era.

- **The crisis of machine's and application's complexities:** Developing software for high performance in computing and data processing on advanced hardware require increasingly sophisticated programming and optimization efforts in order to deliver highly optimized code, which is based on a deep human understanding of both underlying hardware architecture and application domain knowledge.
- **The crisis of human resources:** The human-based programming approach is not sustainable due to the shrinking of the talented software developers' pool. Highly skilled programmers are rare and hard to be massively educated and trained. A parallel-programming expert typically needs 5-7 years of Ph.D. training, which is individual-based in low production, and cannot be in batch mode for a high production. In other words, one expert's knowledge, skills and experiences cannot be automatically inherited by others.

The problems of low performance in machine execution and low production of software development in human resources have been hidden in the CPU-dominated computing era for two reasons. First, the Moore's Law can automatically improve the execution performance by increasing the number of transistors in CPU chips to enhance the capabilities of on-chip caches and computing power. Thus, execution performance continues to be improved without major software modifications. Second, the development of CPU-dominated computing ecosystem for many years has created multilayer abstractions in a deep software stack, e.g. from IAS, to LLVM, to JAVA/C, and to JavaScript/Python. This software stack promotes software productivity by connecting programmers at different layers to co-work together.

However, in the post Moore's Law era, the accelerator-oriented computing environment cannot afford such a multi-layer software stack, which instead requires sophisticated programming to directly interact with heterogeneous hardware devices [4]. Although recent research efforts have been made on developing an intermediate representation (IR) for code generation, such as Weld [5], we strongly argue to fundamentally address the two above mentioned crises by **making the machines to develop software automatically**. In this vision paper, we call this approach as **Software-Defined Software (SDS)**.

II. The SDS Approach

Motivated by recent advances of machine learning, which show its significant capabilities in solving image/voice/video recognition, natural language processing, recommendation systems, and in other domains, we propose the SDS approach in the programming area. The machine learning-based approach has shown promising problem cases that could only be solved by humans in the past. Furthermore, regarding playing the ancient Chinese Go game, which is believed that top professional players are usually human genius, the recent invention of DeepMind Alpha Go program [6] and the ultimate Alpha Zero program [7], has shown that machines can not only beat humans, but also given a strong evidence that self-learning machines can reinvent and enhance human expert's knowledge by a huge speedup (from thousand years to a few days). Thus, we ask a question: *Can a machine do the job of software production? and even better than humans?*

In this vision paper, we boldly believe the answer is YES. On one hand, the machine learning research domains have accumulated intensive experiences of solving various difficult problems defined by humans, with a number of new learning

2575-8411/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDCS.2018.00126

frameworks and models, such as Deep Learning [8], Deep Reinforcement Learning [9], and Transfer Learning [10], [11]. On the other hand, recent automatic programming research work, including DeepMind's Neural Programmer-Interpreters [12] and Microsoft's DeepCoder [13], have opened the door of using neural networks to generate computer programs, although it is still at a very primordial stage.

A. The Core of SDS: Automatic Programming by Deep Neural Networks

Like traditional software development by humans, the SDS approach also needs to consider three factors: *requirement*, *environment*, and *deployment*. First, what are the software requirements (e.g., functionality, performance)? Second, under what environment will be the software developed (e.g, hardware platforms, software layers, programming languages)? Finally, how to deploy the software (e.g, integrating with other tools, end users, input/output)? Therefore, to make a machine write a program, the user has to tell the machine all the three factors. However, the telling can be either in traditional software requirement descriptions (e.g., UML) or by natural language interactions, considering the significant capability of AI for natural language processing and speech recognition.

The core of SDS is automatic programming for a given requirement and environment. We define three levels of automatic programming using the following example. Considering to sort four numbers *[4,2,3,1]* into the ascending order *[1,2,3,4]* using the C programming language. We here do not focus on how the machine understands the requirements. Instead, we focus on how the machine will finish this specific programming task.

Level 1: Programming with Building Blocks: The machine can assemble a program on the basis of a set of available standard library functions or primitives. In this case, the machine understands that the task is to sort an array. Therefore, it decides to choose the standard C library *qsort* function, and write the following program. We here ignore the compare function for *qsort*.

```
int numbers = [4,3,2,1];
qsort(numbers, 4, sizeof(int), a_cmp_func);
```

Level 2: Programming from Scratch: The machine can write its own library function or building blocks without replying on an available function pool. In this case, the machine understands that the requirements can be divided into two steps. First, it should write a sort function. Then it should use the sort function to sort the input numbers. The machine will determine which sort algorithm should use, or a human hint is given to specify the sort algorithm. In this way, the machine will generate the code as follows.

```
void machine_sort(int* input)
{
...
}
int numbers = [4,3,2,1];
machine_sort(numbers);
```

Level 3: Application-Specific Programming: The machine can write a very specific program for the requirements. For this sort example, there is no need to first create a general-purpose sort function and use it to process the input data. The machine can exploit its neural network models to execute the sort, in a way that cannot be explicitly expressed as a sort program. By removing the function calls caused by an independent sort function, the machine can generate specific code directly serving the application.

B. Human-defined Software vs Software-defined Software

The major differences between Human-defined Software (HDS) and Software-defined Software (SDS) are not only the quality of generated software, but the changes in the fundamental way of how software should be composed and utilized. We summarize three dramatic differences as follows.

1) Difference 1: Write a Program vs Be the Program: Although a person can write a program, he/she cannot be the program. Even if the human knows exactly how the sort program works, he/she cannot do the job of the sort program. However, a program that can write a program can make its own copy as the program it writes. This unique advantage makes it possible to achieve quickly re-programming and dynamic optimization, which are thought as challenging programming issues for human developers due to the long-latency and iterative programming based on observations and feedbacks from program executions.

2) Difference 2: Explicit Algorithm Design vs Implicit Algorithm Design: A key feature of a machine-generated program is that it may not use explicit and specific algorithms to execute a task. Instead, the program is based on a combination of multiple neural network models that are trained for mixed functionalities. This opens the door of self-growing software by re-training the underlying Deep Neural Network (DNN) models instead of current reduction-based software structure. A recent example of using a neural network model to replacing a general-purpose B-tree-like index is proposed in [14], which exploits implicit indexing on top of the underlying datasets.

3) Difference 3: Pre-programming vs On-demand Field Programming: Compared to the nature of pre-programming by human software development, another significant capability of SDS is that it makes field programming possible in an on-demand way. For example, a self-driving car could meet an unexpected situation that is out of the scope of any pre-defined rules or algorithms. For human-defined software, any exception handling mechanism can only handle well-defined exceptions. However, unexpected exceptions or runtime optimization opportunities can only be handled by software automatically and timely.

III. The SDS Goal: Making Machines Serving for Humans

We envision the SDS approach will pave the way for machines to execute software production in the following three

ways, namely programming, optimization, and debugging. In each way, the machine can either be a totally independent worker or just an assistant to human software developers.

A. Programming

Programming is the core task of software production. According to the capability levels we defined in the above section, a set of programming work is highly possible that machines can do for humans. According to the three levels we illustrated in the above section, we present typical scenarios of each level's capability.

The Level 1 programming is best suitable for two jobs. One is for application-level script programming that focuses on assembling a set of available script commands or well-defined primitives to satisfy a special application, such as Linux Bash Programming or HTML/JavaScript/CSS programming. Another workplace is for back-end compiler optimizations for code generations, which can work either at the level of LLVM or the underlying instruction sets. The commonplace of these two cases is that their core task is to search a solution based on the combinational possibilities on a set of basic instructions, which has been successfully proven that a machine can do much better than humans, as shown by Alpha Go.

The Level 2 programming is beyond assembling in a way that it can not only pick up existing building blocks (e.g., an instruction or a library function) but also creates new ones adaptively according to the programming requirements. We believe that the most possible and useful scenarios for this level programming is to write library functions or primitives for a new programming language/framework based on self-learning from both implementation examples from other language/frameworks and new language/framework features. For example, if a new language called NL is designed, implementing its standard library (e.g., string functions) is a tedious programming task for human developers. However, it is totally possible if a Level 2 programming machine does the job after it has understood (1) how those functions are implemented in the C library and (2) how the new language NL is different from the C language.

The Level 3 programming is best suitable for applications that have well-defined behaviors but do not imply a clear way of how to do it, for example typical AI-related applications including image/voice recognition, natural language processing, and language translations. In this case, automatic programming provides a possibility of end-to-end programming that directly serve the end-user requirements without a clear reduction of how each step is implemented. Specifically, when the software requirements are based on multiple functionalities (e.g., both voice recognition and voice synthesis), a Level 3 programming capability may provide the only solution that combines separate DNNs for the final requirements.

B. Optimization

Another useful possibility for automatic programming is that it can be used to optimize human-defined software. We believe there are two possibilities for the machine-generated optimizations. The first one is automatic parallelization for a given sequential program written by a human programmer. For example, a human can simply write a quick sort program, which can be further automatically transformed into parallel programs executed on a variety of parallel hardware, such as Intel AVX instructions, GPU, or clusters. The second one is automatic optimization for a given architecture-independent program that can be optimized and re-implemented to be an architecture-dependent program with the considerations of all possible optimization opportunities, such as exploiting locality, prefetching to-be-used data, and best utilizing hardware devices.

Optimization can also happen in a dynamic runtime way considering the feasibility of software-defined software that can make on-demand decisions during program executions. Such dynamic program re-optimizations are critically important for database query execution, datacenter optimizations, graph computations, where unexpected scenarios can happen without prior knowledge. Recently, there are several studies proposed to use machine learning and deep learning-based methods to optimize system parameter configurations [15], [16]; and significant performance improvements have been reported. This also illustrates the automatic programming can optimize human-defined software with great potential.

C. Debugging

Beside programming and optimizations, debugging is another important machine function for the approach of SDS. A well-trained machine can do the debugging job by detecting possible anomalies from executing traces of a set of given test cases. We believe two levels of debugging work are possible for automatic programming machines. First, a machine can work as a regular software tester, who designs test cases against target programs. By understanding the software requirements, regardless how they are defined, e.g., by special instructions (e.g., UML) or by natural languages, a machine can automatically design test cases and behave like a human software tester.

The second level is that a machine can work beyond simple software tests but be able to discover hidden software bugs based on its capability of detecting anomalies. For example, the data race problem for parallel programming is notoriously hard to solve, but it is totally possible for a machine to detect the problem if its underlying DNN models are well-trained by a set of experiences and knowledges on a collection of data race problems. Simply speaking, because a machine can have a huge memory and a fairly faster computation speed, it can certainly discover more software bugs than human testers, if the machine can understand how humans do the job of debugging.

IV. The SDS Roadmap: What Humans Can Do for Machines?

After we have described the SDS approach and its possible usage for software development, we are in a position to answer the ultimate question: *How can we make it happen?* In this

section, we provide a R&D roadmap to turn the SDS approach into a reality in the near future, which essentially solves the problem of *"what humans can do for machines"*.

A. Building Learning Models with Logics

Unlike simple pattern recognitions in image/voice domains which are clearly suitable for deep learning-based approaches, automatic programming is believed to be much harder, which requires reasoning and optimizations based on logics and mathematics proving. Therefore, we believe the first critical step for automatic programming is to combine the neural network based deep learning approach with the logic reasoning based formal approach into a unified framework that shrinks the possible search space for a programming requirement. In this direction, a research problem is how to embed a Z3 [17] or Coq [18]-like tool into a deep learning framework for the purpose of avoiding unnecessary search in a large space.

B. Building Learning Materials/curriculum

As we know, a supervised learning-based approach for a specific purpose must be based on some materials to be learned, which formed the ground truth knowledge. For example, ImageNet [19] is the key for the success of image recognition efforts based on its labelled data sets. However, there is no such a golden standard for automatic programming. To teach a machine efficiently learn how to write a program, even with the possibility of using reinforcement learning, humans should provide a clearly-defined example program set to be the curriculum for the learning programs. We believe different domains, such as GPU programming, distributed programming, and Web programming, must have their own specific curriculum. It is unclear and yet an open question whether it is possible to have a general-purpose program set that can be used for various programming tasks.

C. Building Software Frameworks for Machines

The foundation for automatic programming is that there exist a set of available building blocks that can either be basic units for a machine to use to assemble a program or learning examples to write similar blocks. However, current computing frameworks (e.g., Hadoop or Spark) or programming languages (e.g. C or Java) are designed for human developers which lack machine-understandable definitions for each function or primitive. For example, the C library function *qsort* can only be understood by a human programmer for its semantic meaning. To make automatic programming really happen, it is our human's job to define clear specifications for machines.

D. Building Security/Trust Specification/Mechanisms

For a machine-generated program, the ultimate question is *"Can we trust it"?* It is easier if the program can be clearly expressed into a human-readable program, while much harder, if possible, when the program is expressed by a deep neural network model where the program behavior is hidden inside the model structure and weights of neural connections. Humans must design the security and trust specifications and corresponding enforcement mechanisms before deploying any software-defined software into real-world applications. The problem can be more complicated by the possibility that the check program is also generated by another program, which poses challenges for humans to design an architecture with defense walls that determine whether we should allow machine-generated programs to pass.

V. Research Directions for SDS

Besides the abovementioned tasks, in this section we present four research directions for academic researchers, which we believe are necessary, even not sufficient efforts to implement the SDS approach.

A. On Machine Learning Techniques

The recent rising of deep learning techniques is a key turning point in the development of program synthesis [20]. As the core enabling techniques for SDS, automatic programming, if really evolving from science fiction into reality, must be fundamentally relied on the research advancements of machine learning and neural network techniques. However, unlike the convincing successes in object recognition and game playing, it is still unclear how to organize and train deep neural networks to do the job of programming, which seems beyond the scope of a single task, for example game Go and ImageNet classification. Therefore, the most important research directions for SDS is to enhance machine learning in various aspects. One aspect is to build unified models for different tasks, as shown in [21], which would be a key requirement for general-purpose programming. Another aspect is to further develop novel neural network structures and abstractions towards a deeper understanding and simulations of how human brains do various recognition and intellectual jobs, for example the recent Capsule network model in [22].

B. On Human-Computer Interaction

We are aware of that the most important question for programming or software production is What instead of How? Understanding the user intent correctly plays a more important role than finding a way of how to generate a program according to given requirements. As conventional software engineering wisdom [23] says, *"The hardest single part of building a software system is deciding precisely what to build."*, the SDS approach must solve the problem of how to allow machines understand human intents correctly and efficiently. Therefore, we estimate that, in the near future, it is very important to continue our current research on objective recognition tasks (including images, videos, and voices) in order to deliver a trustful human-computer interaction for the SDS-based software production. Another trend is to combine human-computer interaction and programming into a unified effort as shown by recent research work (e.g., [24]) that aims to translate natural language into computer programs. Without solving the HCI problem, it is still human being's responsibility to write software specifications in a formal or

visual way, which essentially can be understood as *yet another way of programming*, even not using a typical programming language (e.g., C or Java).

C. On Domain-specific Automatic Programming

Since no one has pointed out a clear path of achieving the goal of general-purpose automatic programming, we have to start from domain-specific automatic programming in order to building the SDS approach. There are two important natures of domain-specific programming: One is the clearly-defined requirement and the other is the feasibility of result verification. One of the state-of-the-art program synthesis technique is to implement Program by Example (PbE), as shown by DeepCoder [13], RobustFill [25], and NGDS [26]. However, it is unclear whether and how such a specific automatic programming approach can be applied to general programming without clearly-defined input/output examples.

Another recently focused research topic is to apply deep learning techniques into requirement-fixed programming, which aims to exploit neural networks to solve certain specific difficult problems, which can traditionally rely on well experienced and highly skilled human programmers. A representative example is occurring in the database domain where a number of automatic programming and optimizing techniques are rising, including data indexing [14], query optimization [27], and relational join order enumeration [28]. Although it is hard to estimate how these efforts can contribute to the final goal of SDS, we believe that they can at least be helpful for examining the capabilities and limitations of various machine learning models and algorithms when applied into program synthesis.

D. Basic Infrastructure: Hardware, Software, and Language

The final research direction is about a necessary condition for making SDS happen, which is to build efficient hardware/software infrastructures so that a lot of machine learning applications can be easily and efficiently developed, which could finally provide the foundation on which software-defined software can be produced. An example is the Internet/WWW ecosystem where we humans rely on. The rising of such an ecosystem heavily relies on the massive availability of microprocessors (X86 and ARM), operating systems (Windows and Linux), and high-level programming languages (JavaScript and Python). Therefore, although we don't know how to certainly make SDS happen, we know that it is a critically necessary requirement to build the underlying hardware, software, and language environment to support diverse AI applications.

For hardware, the ending of Moore's Law triggers a high-demand of building customized chips for specific applications [29]. For software, TensorFlow [30] can provide an OS-like functionality between hardware and applications. However, the development of such hardware and software cannot remove the requirement for developing user-friendly language to build deep learning applications with both flexibility and high performance, as shown in recent efforts [31] [32]. We believe that combined efforts on building efficient AI infrastructures are important research topics for the SDS approach.

VI. A Visionary Conclusion

The number of world-class Go masters is much smaller than the number of highly skilled programmers in the world due to a very different intellectual level requirement. If machines can play the roles of highly intelligent Go masters, we believe that they can also become top programmers to deliver best quality code interacting with all kinds of hardware devices. This is the way of Software-defined Software (SDS), and our future way.

VII. Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments and suggestions. This work has been partially supported by the National Science Foundation under grants CCF-1513944, CCF-1629403, and CCF-1718450.

References

[1] M Mitchell Waldrop, "The chips are down for moores law," vol. 530, pp. 144, 02 2016.

[2] *Dennard Scaling*, https://en.wikipedia.org/wiki/Dennard_scaling.

[3] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2011, ISCA '11, pp. 365–376, ACM.

[4] Jay Jay Billings, Alexander J. McCaskey, Geoffroy Vallée, and Gregory R. Watson, "Will humans even write code in 2040 and what would that mean for extreme heterogeneity in computing?," *CoRR*, vol. abs/1712.00676, 2017.

[5] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia, "Weld: Rethinking the interface between data-intensive applications," *CoRR*, vol. abs/1709.06416, 2017.

[6] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016.

[7] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al., "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354, 2017.

[8] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436, 2015.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529, 2015.

[10] Sinno Jialin Pan and Qiang Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[11] Rajat Raina, Alexis Battle, Honglak Lee, Benjamin Packer, and Andrew Y Ng, "Self-taught learning: transfer learning from unlabeled data," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 759–766.

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

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

[14] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis, "The case for learned index structures," *arXiv preprint arXiv:1712.01208*, 2017.

[15] Yan Li, Kenneth Chang, Oceane Bel, Ethan L. Miller, and Darrell D. E. Long, "Capes: Unsupervised storage performance tuning using neural network-based deep reinforcement learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2017, SC '17, pp. 42:1–42:14, ACM.

[16] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, New York, NY, USA, 2017, SIGMOD '17, pp. 1009–1024, ACM.

[17] Leonardo De Moura and Nikolaj Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[18] Adam Chlipala, *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*, MIT Press, 2013.

[19] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 248–255.

[20] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh, "Program synthesis," *Foundations and Trends in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[21] Lukasz Kaiser, Aidan N. Gomez, Noam Shazeer, Ashish Vaswani, Niki Parmar, Llion Jones, and Jakob Uszkoreit, "One model to learn them all," *CoRR*, vol. abs/1706.05137, 2017.

[22] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton, "Dynamic routing between capsules," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., pp. 3856–3866. Curran Associates, Inc., 2017.

[23] Frederick P. Brooks, Jr., "No silver bullet essence and accidents of software engineering," *Computer*, vol. 20, no. 4, pp. 10–19, Apr. 1987.

[24] Pengcheng Yin and Graham Neubig, "A syntactic neural model for general-purpose code generation," *arXiv preprint arXiv:1704.01696*, 2017.

[25] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli, "Robustfill: Neural program learning under noisy I/O," *CoRR*, vol. abs/1703.07469, 2017.

[26] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani, "Neural-guided deductive search for real-time program synthesis from examples," in *International Conference on Learning Representations*, 2018.

[27] J. Ortiz, M. Balazinska, J. Gehrke, and S. Sathiya Keerthi, "Learning State Representations for Query Optimization with Deep Reinforcement Learning," *ArXiv e-prints*, Mar. 2018.

[28] R. Marcus and O. Papaemmanouil, "Deep Reinforcement Learning for Join Order Enumeration," *ArXiv e-prints*, Feb. 2018.

[29] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, no. 2, pp. 21–29, Mar 2018.

[30] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2016, OSDI'16, pp. 265–283, USENIX Association.

[31] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman, "Latte: A language, compiler, and runtime for elegant and efficient deep neural networks," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2016, PLDI '16, pp. 209–223, ACM.

[32] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *CoRR*, vol. abs/1802.04730, 2018.