Overparameterization: A Connection Between Software 1.0 and Software 2.0

Michael Carbin
MIT CSAIL, Cambridge, MA, USA
mcarbin@csail.mit.edu

Abstract

A new ecosystem of machine-learning driven applications, titled *Software 2.0*, has arisen that integrates neural networks into a variety of computational tasks. Such applications include image recognition, natural language processing, and other traditional machine learning tasks. However, these techniques have also grown to include other structured domains, such as program analysis and program optimization for which novel, domain-specific insights mate with model design. In this paper, we connect the world of Software 2.0 with that of traditional software – *Software 1.0* – through *overparameterization*: a program may provide more computational capacity and precision than is necessary for the task at hand.

In Software 2.0, overparamterization – when a machine learning model has more parameters than datapoints in the dataset – arises as a contemporary understanding of the ability for modern, gradient-based learning methods to learn models over complex datasets with high-accuracy. Specifically, the more parameters a model has, the better it learns.

In Software 1.0, the results of the approximate computing community show that traditional software is also overparameterized in that software often simply computes results that are more precise than is required by the user. Approximate computing exploits this overparameterization to improve performance by eliminating unnecessary, excess computation. For example, one – of many techniques – is to reduce the precision of arithmetic in the application.

In this paper, we argue that the gap between available precision and that that is required for either Software 1.0 or Software 2.0 is a fundamental aspect of software design that illustrates the balance between software designed for general-purposes and domain-adapted solutions. A general-purpose solution is easier to develop and maintain versus a domain-adapted solution. However, that ease comes at the expense of performance.

We show that the approximate computing community and the machine learning community have developed overlapping techniques to improve performance by reducing overparameterization. We also show that because of these shared techniques, questions, concerns, and answers on how to construct software can translate from one software variant to the other.

2012 ACM Subject Classification Software and its engineering → General programming languages; Computing methodologies → Machine learning

Keywords and phrases Approximate Computing, Machine Learning, Software 2.0

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.1

Funding This work was supported in part by the Office of Naval Research (ONR N00014-17-1-2699) and the National Science Foundation (NSF CCF-1751011).

Acknowledgements We would like to thank Jonathan Frankle, Benjamin Sherman, Jesse Michel, and Sahil Verma for the research contributions summarized in this work.

3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 1; pp. 1:1–1:13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Software 2.0

Software 2.0 is the vision that future software development will consist largely of developing a data processing pipeline and then streaming the pipeline's output into a neural network to perform a given task [25]. The implication is then that *Software 1.0* – the development of software using traditional data structures, algorithms, and systems – will move lower in the software stack, with less of the overall software engineering effort dedicated to building software in this traditional way.

In domains such as image recognition and natural language processing, this Software 2.0 vision has been largely realized: neural networks have replaced hand-engineered models for vision, natural language processing, and other traditional, machine learning tasks. The key observation in these domains is that building models by hand is labor-intensive, requires significant expertise in domain-specific modeling, and is difficult to adapt across similar tasks. Such engineering can include both hand-developing analytical models as well as engineering features to use as inputs to machine learning models, such as Support Vector Machines.

The promise of Software 2.0 is that neural networks have the capability to automatically learn high-dimensional representations of the system's inputs from raw data alone. For example, the word2vec [33] model automatically learns high-dimensional vector-valued representations of natural language words that capture semantic meaning and can automatically be used in a variety of natural language tasks. This stands in contrast to traditional approaches that require experts to develop algorithms that manually identify important words or subwords within a piece of text. With this opportunity, Software 2.0 – in its most extreme form – holds out the promise of replacing large, hand-developed components of traditional software systems with neural networks.

1.1 Overparameterization

A key component of the success of Software 2.0 is overparameterization. Contemporary explanations for the relative ease of training neural networks on large, complicated datasets with relatively simple optimization methods such as gradient descent posit that overparameterization is a key ingredient. Specifically, overparameterization results in improved learning both in total accuracy and accuracy as a function of data points of training [29, 1, 15, 2].

Overparameterization is a condition in which a machine learning model – such as a neural network – has more parameters than datapoints. In such a regime, the contemporary understanding is that a learning algorithm can identify parameters for the model that can perfectly memorize the data. The connotation of the term *over*parameterization is therefore a note that – in principle, for well-posed datasets – it is possible to design a model that has fewer parameters than the number of datapoints for which a learning algorithm can identify an effective setting for those parameters for the task at hand.

Although overparameterization seems uniquely restricted to the domain of machine learning models and Software 2.0, in this paper we argue that overparameterization is a key ingredient in the development of Software 1.0 and that therefore overparameterization is binding force to relate Software 1.0 and Software 2.0.

1.2 Overparmeterization in Software 1.0

We define overparameterization in Software 1.0 as a condition in which the system performs more computation than is necessary for the task at hand. As a simple example, the traditional software development practice of uniformly choosing single- or double-precision for all real-like

numbers in a system is a simplifying assumption that ignores the fact that different quantities in the program may require different precisions. Choosing precision in this regime requires the precision of all operations to be that of the operation that the program needs to be the most precise. However, for some applications, data and operations may require only limited precision, such as half-precision or even less, while still enabling the application to produce an acceptable result.

The difference between selecting uniform precision and selecting an appropriate precision per operation is an inherent trade-off: uniformly selecting a high-precision is easier than the detailed numerical analysis required to soundly select a precision per operation [12]. However, uniformly selecting precision may not yield optimal performance: more precise floating-point operations are more computationally and memory intensive than less precise operations.

1.3 Reducing Overparameterization (Approximate Computing)

In both Software 1.0 and Software 2.0, researchers have sought techniques to address the fact that overparameterization results in reduced performance, increased energy consumption, and decreased ubiquity (requiring significant resources from the user to execute the software).

For Software 1.0, the approximate computing community has shown that it is possible to eliminate unnecessary computation and, correspondingly, improve performance. Specific techniques have included floating-point precision tuning (i.e., choosing less precise arithmetic) [12, 11, 44, 10, 34], loop perforation (eliding computations entirely) [36, 35, 49], and function substitution (replacing entire sub-computations whole cloth with less expensive, approximate implementations) [23], and more [13, 16, 17, 27, 38, 39, 45, 9].

For Software 2.0, the machine learning community has developed a variety of new techniques to reduce overparameterization in neural networks. *Quantization* chooses new, low-precision representations of the parameters and arithmetic in neural networks [40]. *Pruning* ignores subsets of a neural network's parameters [26, 21, 41]. *Distillation* trains a new, smaller network to mimic the behavior of a large, well-trained network [5, 22]. Each of these techniques have direct analogues to techniques in approximate computing: precision-tuning, loop perforation, and function substitution, respectively.

1.4 Shared Questions

The core premise of this paper is that overparameterization in both Software 1.0 and Software 2.0 and the overlap in techniques for reducing overparameterization in both, enable us to interpolate between Software 1.0 and Software 2.0, mapping observations and questions from one software construction methodology to the other and vice-versa.

For example, the first question we ask is, is overparameterization is necessary? For Software 1.0, it is assumed that developers – perhaps with significant effort – can develop optimized implementations from the outset given a specification of the system's requirements. The analogous question for Software 2.0 is if it is possible to train – from scratch – a neural network with significantly fewer parameters to model a given problem. We recount our results on the Lottery Ticket Hypothesis [18] that demonstrate that from scratch training for standard problems is possible.

Second, a claim for Software 2.0 is that the parameters of its design – such as the specific neural network architecture – can be optimized alongside the system's objective, provided that these parameters are differentiable. The second question we therefore ask is, can we integrate approximation transformations from the start – and throughout the lifetime – of a Software 1.0 system? We survey our recent results on noise-based sensitivity analysis for

programs by showing that if we embrace Software 2.0's aim to minimize its expected behavior – versus its worst-case behavior – then it is possible to integrate approximations into the optimization of the system's overall objective using gradient descent.

Finally, we discuss several questions on the compositionality and correctness of future of Software 1.0 and Software 2.0 systems and propose directions forward.

1.5 Directions

Software 1.0 and Software 2.0 share the phenomenon of overparameterization, share the same approaches for reducing overparameterization, and – once connected – share overlapping challenges to their construction. By drawing these connections, we hope to identify principled techniques to approach software design, correctness, and performance jointly for both Software 1.0 and Software 2.0.

2 Reducing Overparameterization

Researchers in both approximate computing and machine learning have sought techniques to automatically eliminate overparameterization. In the machine learning setting, there are a variety of techniques that reduce parameter counts by more than 90% while still maintaining the accuracy of the neural network on the end task. The goal of reducing parameter counts is multi-fold: 1) reducing the representation size of the network reduces storage and communication costs [21, 22], 2) reducing parameters eliminates computation, and 3) the combination of effects overall improves performance and energy consumption [50, 37, 31].

For example, the following code implements a neural network layer that computes a matrix-vector product of its internal weights (`weights`) with its `m`-dimensional input (`x`). The result is an `n`-dimensional vector passed through a Rectified Linear Unit activation function (`max(0, output)`). The `n`-dimensional result denotes the output of `n` neurons.

```
1  float x[] = { ... };
2  float weights[][] = { ... };
3  float output[] = { ... };
4  for (int i = 0; i < n; ++i)
5  {
6      for (int j = 0; j < m; ++j)
7      {
8          output[i] += weights[i][j] * x[j];
9      }
10 }
11 return max(0, output);
```

Quantization reduces the number of bits used to represent each weight and compute each operation within the neural network's computation. We can capture this optimization as classic precision selection where, for example, this program could be written to use 16-bit precision floating-point instead of 32-bit `float` types.

Pruning takes a trained large model and eliminates weights by, for example, removing the weights of the smallest magnitude. For this example program, this is equivalent to eliding a subset of the loop iterations – i.e., loop perforation – based upon a pre-determined mask. Eliding iterations of the loop over `j`, elides individual weights in each of the `n` neurons while eliding iterations of the loop over `i` elides neurons in their entirety. Both options have been explored in the literature [21].

Distillation takes a large, trained model and trains a smaller network to mimic the outputs of this model. In this example, this entire layer could be substituted with an alternative implementation. For example, this computation is no more than a matrix-vector product and therefore it is to possible accelerate this computation by instead learning a fast, low-rank approximation of `weights` and computing with that instead [14].

3 Fewer Parameters from the Start

The standard conceptualization of the approximate computing workflow applies approximation transformations after an initial, end-to-end development of a system. Specifically, the standard workflow requires that a developer write an additional *quality-of-service* specification [36] after developing their program. This specification states how much error – measured with respect to the ground truth – that the user can tolerate in their program's output. Given this specification, an approximate computing system then searches the space of approximation transformations to produce a program that meets the quality-of-service specification.

Reducing overparameterization in Software 2.0 follows a similar methodology. Standard methodologies apply pruning, quantization, and distillation to a fully trained neural network. However, the reality that these techniques can be applied lends itself to the question: why not simply use a smaller, more efficient neural network from the start?

Until our recent results [18], contemporary understanding in the machine learning community was that small neural networks are harder to train, reaching lower accuracy than the original networks when trained from the start. Alternatively, overparameterization is required for effective learning.[1] Figure 1 illustrates this phenomenon. In this experiment, we randomly sample and train small networks. These networks are sampled from the set of subnetworks of several different neural network architectures for the MNIST digit recognition benchmark and the CIFAR10 image recognition benchmark. Across various sizes relative to the original reference networks, the sparser the network (fewer parameters), the slower it learned and the lower its eventual test accuracy.

However, this figure also shows the results of our techniques for identifying *winning tickets* [18]: small networks that do in fact train as well (or better) than the original network. The solid lines in the graph are winning tickets, for which these graphs show that even down to a small size, these networks achieve high-accuracy as well as train quickly.

We identify winning tickets by pruning: randomly initialize a neural network $f(x; W_0)$, train it to completion, and prune the weights with the lowest magnitudes (and repeat over multiple iterations). To initialize the winning ticket, we reset each weight that survives the pruning process back to its value in W_0.

The success of this technique has lead us to pose and test the *lottery ticket hypothesis*: any randomly-initialized, neural network contains a subnetwork that is initialized such that – when trained in isolation – it can learn to match the accuracy of the original network after at most the same number of training iterations. In our work, we have supported this hypothesis through experimental evidence on a variety of different neural network architectures. However, our approach to identify winning ticket still requires training the full network: the remaining science is to determine a technique to identify winning tickets earlier in the training process.

[1] "Training a pruned model from scratch performs worse than retraining a pruned model, which may indicate the difficulty of training a network with a small capacity." [28] "During retraining, it is better to retain the weights from the initial training phase for the connections that survived pruning than it is to re-initialize the pruned layers...gradient descent is able to find a good solution when the network is initially trained, but not after re-initializing some layers and retraining them." [21]

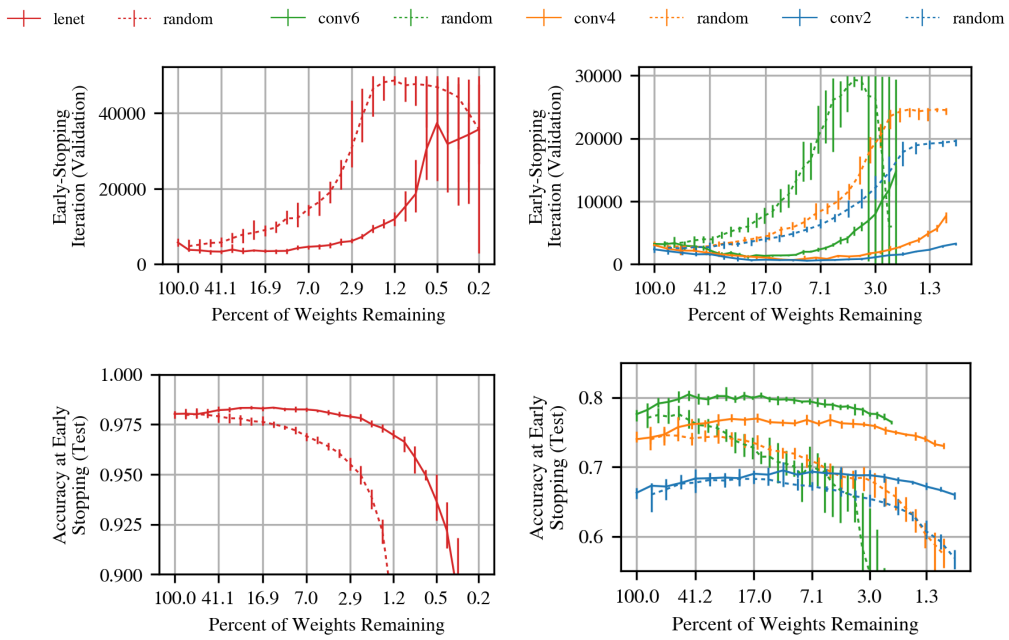

Figure 1 The iteration at which early-stopping would occur (left) and the test accuracy at that iteration (right) of the lenet architecture for MNIST and the conv2, conv4, and conv6 architectures for CIFAR10 ([18] Figure 2) when trained starting at various sizes. Dashed lines are randomly sampled sparse networks (average of ten trials). Solid lines are winning tickets (average of five trials).

4 Approximation from the Start

Recent work has investigated techniques to introduce pruning, quantization, early in the training process with limited success [40, 30, 19]. The key idea behind these approaches is that the parameters of these techniques can be designed to be differentiable. Notably, in the standard Software 2.0 methodology, a learning algorithm identifies settings of the system's parameters that minimize the system's expected error with respect to a ground truth given by a training dataset. Within this umbrella, an error specification is therefore builtin to this methodology and – typically – the learning algorithm selects the system's parameters through gradient descent. Therefore, if the parameters of these reduction techniques are differentiable, then they can be learned alongside the system's standard parameters.

Approach

Inspired by Software 2.0, in recent results we have developed a sensitivity and precision selection technique for traditional numerical programs that is differentiable. Our approach models sensitivity as random noise – e.g., sampled from a gaussian distribution – added to each operation in the program such that the standard deviation of the noise's distribution indicates the sensitivity of the program's expected error to changes in the operation's output.

If an operation's noise distribution can have large standard deviation without perturbing the expected error of the program, then the program is relatively less sensitive to perturbations in that operation's results. On the other hand, if an operation's noise distribution must have small standard deviation to avoid perturbing the expected error of the program, then the program is relatively more sensitive to changes in that operation's result.

Table 1 We compare the absolute error bound from FPTuner against the empirically determined root mean squared error from our approach. Mean bits is the average number of bits in the mantissa of the approximate program in contrast to FPTuner's 52-bit mantissa (from doubles).

Benchmarks	FPTuner	RMSE	Mean Bits
verlhulst	3.79e-16	1.13e-16	51
sineOrder3	1.17e-15	7.64e-16	50
predPrey	1.99e-16	1.90e-16	50
sine	8.73e-16	8.34e-17	51
doppler1	1.82e-13	3.30e-14	53
doppler2	3.20e-13	3.30e-14	53
doppler3	1.02e-13	8.22e-14	53
rigidbody1	3.86e-13	1.37e-13	51
sqroot	7.45e-16	4.00e-16	50
rigidbody2	5.23e-11	6.08e-12	51
turbine2	4.13e-14	2.57e-14	50
carbon gas	1.51e-08	3.01e-09	49
turbine1	3.16e-14	8.69e-15	51
turbine3	1.73e-14	5.09e-15	52
jet	2.68e-11	2.45e-11	54

The goal of our approach is to identify the maximum standard deviations for the distribution of each operator such that the resulting program still delivers acceptable expected error. Our technique poses this goal as an optimization problem and solves the problem through stochastic gradient descent. We have shown that these sensitivities are informative by developing a precision selection approach that takes as input the set of sensitivities for the operations in the program and produces an assignment of precision to each operator.

Case Study

We have applied our approach to a set of scientific computing benchmarks used by FPTuner's developers to develop and evaluate FPTuner [10]. FPTuner is a tool that can identify an assignment of precisions to operators such that the resulting program satisfies a user-provided worst-case error. FPTuner uses a combination of static error analysis and quadratic programming to automatically identify an assignment that satisfies the provided bound. To evaluate our sensitivity analysis, we have devised a technique to map the sensitivity of each operator to the number of bits to use in an arbitrary precision library (i.e., MPFR).

Table 1 presents our preliminary results of our approach. We configured our approach to produce sensitivities such that expected error of the program is less than FPTuner's absolute error. The results are that our approach generates tighter expected error bounds using fewer bits than FPTuner for 10 out of 15 benchmarks. For the remaining 5 benchmarks, our approach requires at most 2 extra bits on average than in FPTuner.

Sensitivity analysis provides critical information to an approximate computing system – the sensitivity of the program's output to changes in semantics of an operation. By embracing expected error and working within a differentiable setting, we have arrived at an approach that mates Sofware 1.0 (approximate computing) with Software 2.0.

5 Discussion

Software 2.0 – replacing core components of traditional-developed software systems with learned components – is a lofty goal. A particular challenge to this goal is that an integral component of Software 2.0 – neural networks – have so far proved to be difficult to interpret and reason about for the purposes of validating the resulting system's behavior. However, the approximate computing community has faced similar difficulties with 1) understanding the composition of approximation with the original system, 2) giving the resulting approximate system a useful behavioral specification, and 3) developing analysis frameworks for reasoning about that behavioral specification. Through the shared tie of overparameterization, Software 2.0 and approximate computing can share techniques to solve their common challenges.

Compositionality

In a Software 2.0 system that composes both neural networks and traditional computation, reasoning about the behavior of the resulting system is challenging. Specifically, the high-dimensional representations that neural networks learn are not necessarily directly interpretable by humans. For example, word2vec represents words as n-dimensional vectors, where n is often large (i.e., greater than 256), with limited semantic meaning assigned to each dimension. Moreover, interpreting the composed behavior of the neural network with the larger system is challenging because the network's task may not easily permit a compositional specification of its behavior for which global faults can be reduced to local reasoning.

The approximate computing community has faced similar challenges. In its most idealized form, the community's agenda has advocated for a variety of techniques that are together composed with the program in a blackbox manner with limited interaction with the developer or user. Techniques such as loop perforation and function substitution may remove or replace large fractions of a system's computation with the result being limited understanding of the system's semantics. As a consequence, a developer may receive an approximate system for which failures are hard to address because it is not clear if they are resident in the original, non-approximate program program or a created anew through approximation [6].

One avenue for Software 2.0 to follow is the direction of our work in approximate computing to apply the concept of *non-interference* to support compositional reasoning [6]. Our proposed programming methodology argues that a developer should develop a program and establish its *acceptability properties* – the basic invariants that must be true of the program to ensure that its execution and ultimate results are acceptable for the task at hand [42]. Example invariants include standard safety properties – such as memory safety – but also include application-specific *integrity properties* [6, 7]. For example, a computation that computes a distance metric between two values should return a nonnegative result, regardless of the extent of its approximation.

Given a program and its acceptability properties, the developer communicates to the approximate computing system points in the application at which approximation opportunities are available and do not interfere with the properties established for the original program. Therefore by non-interference, we mean that if the original program satisfies these properties, then the approximate program satisfies these properties. Reasoning about non-interference can include reasoning about information flow to ensure that approximations do not change the values of data and computations that are involved in the invariants. For Software 2.0, this methodology could enable existing software components to be replaced with learned variants that do not interfere with the program's acceptability properties. This framework can, for example, underpin recent work on adding assertions to machine learning models [24].

Correctness

The act of developing a specification of the full functional correctness of a software system as can be done for traditional software does not directly translate to either Software 2.0 or approximate systems. For many Software 2.0 systems – such as those in computer vision and natural language processing – the correctness specifications for these problems is either not known or not well posed. For example, specifying that the system correctly classifies an image of a cat as a cat does not have a declarative, logical specification. Therefore developers typically evaluate a Software 2.0 system based on its expected error over a sample of data from its input distribution. The correctness specification for approximate systems is challenging in that – by definition – an approximate system returns different results from its original implementation. By this nature, an approximate system's natural measure of its behavior is its error with respect to the original program.

In both domains, there has developed a shared understanding that statistically bounding the error of the system is a potential direction for improving the confidence in a system. The core conceptual challenge is that while the error of either type of system can be measured on a test set drawn as a sample of the system's input distribution to give confidence, the key property to bound is the system's *generalization*: its error on unseen data.

In the statistics and machine learning community, such bounds are known as *generalization bounds*. A simplified structure for these bounds is that for $\delta \in (0, 1)$, with probability $1 - \delta$ over a random sample s from an input distribution $\mathcal{D}$, $\mathcal{L} \leq B(\hat{\mathcal{L}}(s), \delta)$. Here $\mathcal{L}$ is the expected error (or loss) of the system on unseen data, $\hat{\mathcal{L}}(s)$ is the observed error of the system on s, and B denotes a function that computes the bound. The computed bound is at least $\hat{\mathcal{L}}(s)$ – i.e., the error on unseen data is no better the error on observed data – and decreasing in δ – the less confidence one requires of the bound, the closer it is to the observed error.

The approximate computing community has also developed statistical bounds on the behavior of approximate systems. These analyses include bounding the probability the system produces the correct result [8, 34], bounding the probability it produces a result exceeding a specified distance from that of the original program [43, 35], and bounding its expected error [51]. The shared focus and results on statistical bounds between Software 2.0 and approximate computing suggests that these bounds may be integral specifications for ensuring the behavior of future systems, including both Software 2.0 and Software 1.0.

Analysis

If Software 2.0 takes hold, then the reasoning methods we have used to build software will need to change. Traditional software construction methodologies have been designed around the classic building blocks of Computer Science: discrete and deterministic math, algorithms, and systems. However, the basic analysis that underpins reasoning about Software 2.0 is based on continuous math and statistics: neural networks are formalized as functions on real numbers and formalizing generalization relies on statistical analysis. The required analysis for approximate systems also relies on continuous math and statistics: these systems compute on reals in their idealized mathematical specification and statistical bounds are the preferred framework for reasoning about their correctness.

A resulting challenge is that the formal methods, programming languages, and systems communities – communities that are major contributors to the mission to formalize and deliver automated reasoning systems – has invested less in understanding real-valued and/or probabilistic computations than for discrete computations. The result is that there is a significant gap between the needs of future systems and the capabilities of existing analysis. Therefore, the next generation of systems will need new computational building blocks.

For example, our work on programming with continuous values makes it possible to soundly compute on real numbers to arbitrary precision as well as soundly combine real-valued computation with discrete computation [47, 46]. Specifically, discrete computations on the reals – e.g., testing if two real numbers are equal – is undecidable in general. This fact stands as a contradiction to modern programming languages that expose floating-point as an approximation of the reals and permit developers to test them for equality.

As another example, many of the reasoning tasks for future software – such as profiling or computing bounds on their behavior – will be probabilistic computations. One can pose these analyses as queries on the behavior of probabilistic models, as currently captured by the community around *probabilistic programming*: representing probabilistic models as programs with stochastic choices [20]. In these systems, the supporting programming system can perform *inference* to compute answers to questions such as, what's the probability that the program produces a value that exceeds a given bound?

However, beneath the covers of these programming systems lies a space of inference algorithms, which are algorithms that manipulate probability distributions. If probabilistic computations are to be integral to future software, then future and developers and systems will need to understand and manipulate inference algorithms and therefore need to understand and manipulate probability distributions as first-class values. In this space, there are open questions about the architecture of programming systems for implementing inference algorithms that we – along with others – are exploring [4, 3, 32, 48].

In sum, the future of Software 2.0 – and the extent to which we can reason about its behavior – critically depends on the development of new programming models and abstractions for continuous math and statistics.

6 Conclusion

Software 1.0 and Software 2.0 appear radically different. The development methodology for Software 1.0 revolves around developers manually architecting the overall structure and constituent algorithms of a system. In contrast, the mantra of Software 2.0 is to delegate much of the system's algorithms and – perhaps – even its structure to neural networks or other machine learning methods. However, overparameterization is a shared connection between both methodologies.

In the case of Software 1.0, developers rely on coarse, general-purpose abstractions that are easy to program with but that perform more computation than is necessary for the task at hand. In the case of Software 2.0, results have shown that larger neural networks learn more easily than their smaller counterparts, but – in principle – smaller networks are capable of representing the task. The trade-off for both of these methodologies is that the increased ease in development comes at the expense of performance.

To address this problem, both the approximate computing community and the machine learning community have coalesced on techniques to reduce overparameterization in Software 1.0 and Software 2.0, respectively, while still preserving ease of development. Based on this shared goal, this paper offers the viewpoint that questions, challenges, and techniques from both communities can translate from one to the other. As the Software 2.0 future unfolds, new questions about the composition and correctness of these systems will arise. However, these questions can be addressed jointly within both Software 1.0 and Software 2.0.

References

1 Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A Convergence Theory for Deep Learning via Over-Parameterization. In *International Conference on Machine Learning*, pages 242–252. PMLR, 2019.

2 Sanjeev Arora, Simon S. Du, Wei Hu, Zhiyuan Li, and Ruosong Wang. Fine-Grained Analysis of Optimization and Generalization for Overparameterized Two-Layer Neural Networks. In *International Conference on Machine Learning*, pages 322–332. PMLR, 2019.

3 E. Atkinson, C. Yang, and M. Carbin. Verifying Handcoded Probabilistic Inference Procedures. *CoRR*, 2018. arXiv:1805.01863.

4 Eric Atkinson and Michael Carbin. Towards Correct-by-Construction Probabilistic Inference. In *NIPS Workshop on Machine Learning Systems*, 2016.

5 Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, 2014.

6 M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving Acceptability Properties of Relaxed Nondeterministic Approximate Programs. In *Conference on Programming Language Design and Implementation*, pages 169–180. ACM, 2012. doi:10.1145/2254064.2254086.

7 M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Verified integrity properties for safe approximate program transformations. In *Workshop on Partial Evaluation and Program Manipulation*, pages 63–66. ACM, 2013. doi:10.1145/2426890.2426901.

8 M. Carbin, S. Misailovic, and M. Rinard. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In *International Conference on Object Oriented Programming Systems Languages & Applications*, pages 33–52, 2013. doi:10.1145/2509136.2509546.

9 S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving Programs Robust. In *Symposium on the Foundations of Software Engineering and European Software Engineering Conference*, pages 102–112, 2011. doi:10.1145/2025113.2025131.

10 Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Rigorous Floating-point Mixed-precision Tuning. In *Symposium on Principles of Programming Languages*, pages 300–315. ACM, 2017.

11 Eva Darulova, Einar Horn, and Saksham Sharma. Sound Mixed-precision Optimization with Rewriting. In *International Conference on Cyber-Physical Systems*, pages 208–219. IEEE Press, 2018. doi:10.1109/ICCPS.2018.00028.

12 Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *Symposium on Principles of Programming Languages*. ACM, 2014. doi:10.1145/2535838.2535874.

13 M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *International Symposium on Computer Architecture*, pages 497–508. ACM, 2010. doi:10.1145/1815961.1816026.

14 Emily Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation. In *Advances Neural Information Processing Systems*, 2014.

15 Simon S. Du, Jason D. Lee, Haochuan Li, Liwei Wang, and Xiyu Zhai. Gradient Descent Finds Global Minima of Deep Neural Networks. In *International Conference on Machine Learning*, pages 1675–1685. PMLR, 2019.

16 D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *International Symposium on Microarchitecture*, pages 7–18, 2003. doi:10.1109/MICRO.2003.1253179.

17 H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–312. ACM, 2012. doi:10.1145/2150976.2151008.

18 Jonathan Frankle and Michael Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *International Conference on Learning Representations*, 2019.

19 Trevor Gale, Erich Elsen, and Sara Hooker. The State of Sparsity in Deep Neural Networks. *CoRR*, 2019. arXiv:1902.09574.

20 Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Conference on Uncertainty in Artificial Intelligence*, pages 220–229. AUAI Press, 2008.

21 Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, 2015.

22 Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. In *NIPS Workshop on Deep Learning*, 2014. arXiv:1503.02531.

23 H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic Knobs for Responsive Power-Aware Computing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–212. ACM, 2011. doi:10.1145/1950365.1950390.

24 Daniel Kang, Deepti Raghavan, Peter Bailis, and Matei Zaharia. Model Assertions for Debugging Machine Learning. In *NeurIPS MLSys Workshop*, 2018.

25 Andrej Karpathy. Software 2.0, November 2017.

26 Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, 1990.

27 L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra. ERSA: error resilient system architecture for probabilistic applications. In *Design, Automation and Test in Europe*, pages 1560–1565, 2010. doi:10.1109/DATE.2010.5457059.

28 Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *International Conference on Learning Representations*, 2017.

29 Yuanzhi Li and Yingyu Liang. Learning Overparameterized Neural Networks via Stochastic Gradient Descent on Structured Data. In *International Conference on Neural Information Processing Systems*, pages 8168–8177. Curran Associates Inc., 2018.

30 Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through L_0 regularization. In *International Conference on Learning Representations*, 2018.

31 Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *International Conference on Computer Vision*, pages 5068–5076, 2017. doi:10.1109/ICCV.2017.541.

32 Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. Probabilistic Programming with Programmable Inference. In *Conference on Programming Language Design and Implementation*, pages 603–616, 2018. doi:10.1145/3192366.3192409.

33 Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, 2013.

34 S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. Rinard. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. In *International Conference on Object Oriented Programming Systems Languages & Applications*, pages 309–328. ACM, 2014. doi:10.1145/2660193.2660231.

35 S. Misailovic, D. Roy, and M. Rinard. Probabilistically Accurate Program Transformations. In *International Symposium on Static Analysis*, pages 316–333. Springer, 2011. doi:10.1007/978-3-642-23702-7_24.

36 S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *International Conference on Software Engineering*, pages 25–34, 2010. doi:10.1145/1806799.1806808.

37 Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning Convolutional Neural Networks for Resource Efficient Inference. In *International Conference on Learning Representations*, 2017.

38 S. Narayanan, J. Sartori, R. Kumar, and D. Jones. Scalable stochastic processors. In *Design, Automation and Test in Europe*, pages 335–338. IEEE Computer Society, 2010. doi:10.1109/DATE.2010.5457181.

39 K. Palem. Energy aware computing through probabilistic switching: A study of limits. *IEEE Transactions on Computers*, 2005.

40 Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. In *International Conference on Learning Representations*, 2018.

41 Samyam Rajbhandari, Yuxiong He, Olatunji Ruwase, Michael Carbin, and Trishul Chilimbi. Optimizing CNNs on Multicores for Scalability, Performance and Goodput. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 267–280. ACM, 2017. doi:10.1145/3037697.3037745.

42 M. Rinard. Acceptability-oriented computing. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 221–239. ACM, 2003. doi:10.1145/949344.949402.

43 M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *International Conference on Supercomputing*, pages 324–334. ACM, 2006. doi:10.1145/1183401.1183447.

44 C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 27:1–27:12. ACM, 2013. doi:10.1145/2503210.2503296.

45 A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *Conference on Programming Language Design and Implementation*, pages 164–174. ACM, 2011. doi:10.1145/1993498.1993518.

46 Benjamin Sherman, Jesse Michel, and Michael Carbin. Sound and robust solid modeling via exact real arithmetic and continuity. In *International Conference on Functional Programming*. ACM, 2019.

47 Benjamin Sherman, Luke Sciarappa, Adam Chlipala, and Michael Carbin. Computable decision making on the reals and other spaces: via partiality and nondeterminism. In *Symposium on Logic in Computer Science*, pages 859–868. ACM, 2018. doi:10.1145/3209108.3209193.

48 Benjamin Sherman, Jared Tramontano, and Michael Carbin. Constructive probabilistic semantics with non-spatial locales. In *Workshop on Probabilistic Programming Languages, Semantics, and Systems*, 2018.

49 S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance vs. Accuracy Trade-offs With Loop Perforation. In *Symposium on the Foundations of Software Engineering*, pages 124–134. ACM, 2011. doi:10.1145/2025113.2025133.

50 Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Conference on Computer Vision and Pattern Recognition*, pages 6071–6079, 2017. doi:10.1109/CVPR.2017.643.

51 Z. Zhu, S. Misailovic, J. Kelner, and M. Rinard. Randomized Accuracy-Aware Program Transformations for Efficient Approximate Computations. In *Symposium on Principles of Programming Languages*, pages 441–454. ACM, 2012. doi:10.1145/2103656.2103710.