Systems Should Automatically Specialize Code and Data

Brandon Lucia Todd Mytkowicz
Microsoft Research

Generality & Programmability are at Odds with Specialization & Efficiency

Today's languages, systems, and architectures include considerable support for *generality* and *programmability*. Systems are built to enable simple implementation of a broad class of programs, and to provide an acceptable level of performance and efficiency. The profusion of high-level languages, runtime and system support for tasks like garbage collection and virtualization, and architectural mechanisms like virtual memory speak to the importance placed by system designers on supporting programmability for a broad class of programs. While systems go to these great lengths for generality and programmability, efficiency is also a key design requirement because the amount of time, space, and energy a computation requires ultimately determines its cost. Unfortunately, system support for generality is frequently at odds with the need for high performance and efficiency. Mechanisms and abstractions built to provide generality and programmability come at a cost in efficiency and programmers frequently go to lengths to elide those mechanisms and abstractions to make their computations more efficient.

Specialization is one approach taken by programmers to trade generality for efficiency. Broadly, specialization exploits structure present in a problem to increase the efficiency of its implementation. To specialize a program, a programmer makes assumptions about things like the nature of expected program inputs or the machine that will run the program. The programmer then carefully writes an implementation to take advantage of those assumptions to improve efficiency. Specialization can affect a program's data representation, the algorithm that manipulates that data representation, and the implementation of both.

For example, say a programmer has implemented a program that sorts numbers and wants it to be more efficient. If the programmer knows that their general sorting code will only ever sort peoples' ages, they may choose to specialize their data structures and sorting code to store and manipulate only 7 bit values, assuming no one will live beyond 128 years. The age-specialized sorting implementation takes advantage of structure in the program's input, making more efficient use of resources than a general implementation that stores and manipulates 64 bit numbers.

While such *ad hoc*, manual specialization can be fruitful, it has several drawbacks. First, even for a particular program, different execution environments may deal with different input sets that vary in the structure they have and whether or not they have structure at all. That variation requires the programmer to adapt the program to each different environment, which is onerous. Second, specialization may make programming more difficult, as programmers avoid general purpose abstractions and take advantage of specific data or machine characteristics. Such mechanisms often exist to insulate programmers from the complexity of data and machine characteristics, so manual specialization may lead to more complex (and potentially error-prone) programming.

Vision

It is our vision that specialization does not need to be ad hoc *and does not need to be applied manually.* We propose that it is a worthy and attainable research agenda to develop systems – programming languages, system software, and computer architectures – that automate the process of program specialization. We believe that to develop those systems we must borrow techniques from machine learning, especially from deep learning.

Strategy

Our strategy for automatic specialization is to use deep learning to produce specialized data representations and algorithms and to use techniques for disciplined approximation to control or tolerate any imprecision introduced by what is learned.

Automatic Specialization with Deep Learning

Deep learning is the science of automatically learning abstract data representations. Deep learning algorithms are built up from a series of processing stages and each stage learns a "layer" of abstraction. Each layer learns a more abstract data representations than the abstractions learned at the previous layer. Deep learning algorithms learn by looking at examples of data and then tuning the abstraction at each layer according to some optimization function, like how much error the layers of abstraction introduce.

One of the key insights of deep learning is that an "algorithm" is trivial if the problem's data representation contains a simple representation of the answer. For example, if the final layer in a deep architecture contains a single bit which encodes whether an image contains a cat, the algorithm for determining whether an image contains a cat is trivial (is the bit set?). Deep learning recognizes that data representation and algorithm are inseparably intertwined and as such has developed learning algorithms that explicitly learn both data representation *and* algorithm at the same time.

This paper suggests that learned data representations and algorithms are inherently specialized. Deep learning algorithms learn representations that are derived from example data, but that generalize to unseen examples that share characteristics of the example data.

Example As an illustrative example, say a programmer wants to manually implement a program that manipulates images of faces to determine the disposition of the person in the image. The naive way to do that is to store the images' entire pixel arrays and metadata and do the disposition computation over the full images. That naive implementation might then do a search over the image to find certain known mouth or eye shapes characteristic of certain dispositions.

Knowing the images are all of faces and the nature of the task, the programmer may try to specialize the code and data representation. One way to do that is to store and manipulate an abstract data type representing eye and mouth characteristics only. While that specialization may improve efficiency by reducing storage and computation overhead, in general, it is hard to predict how efficient a particular specialization strategy will be. Additionally, the programmer is forced to do lots of work to apply the specialization. We think deep learning can address both of these issues. Rather than forcing a programmer to come up with good face abstractions and code that uses them, a deep learner explicitly learns an optimized data representation and algorithm at the same time. The programmer need only provide a the set of face images, their unspecialized reference implementation, and a characterization of the amount of error that is acceptable for the task (*e.g.*, 1% of the time, the disposition can be mischaracterized).

Note that while this example and many existing deep learning successes focus on domains such as image recognition and natural language processing, there is nothing integral to the approach of deep learning that limits is utility to more general domains.

Specialization vs. Approximation It is important to note that specialized implementations learned using deep learning may introduce some imprecision, but there is no inherent reason the approach needs to be approximate. In particular, given a complete *behavioral* specification of a set of inputs, instead of just example data, precise program synthesis techniques can be used to exactly implement the layers in the deep learner.

When solutions are approximate, however, systems must provide support to control or tolerate imprecision introduced by the learned specialization. Recent results in the approximate computing literature show that it is possible to reason about imprecision in the type system [3], reason about programs that manipulate uncertain values [1], and make assertions about potentially imprecise values in a program [2, 4]. We believe these mechanisms for managing and imprecision are moves in the right direction and will play be integral to an automatic specialization system.

Research Challenges

To follow this strategy of deep learning to realize our vision of automatic specialization, there are several important research challenges. The key research challenges we have identified are related to providing system support for three things: (1) Support for collecting and managing example data; (2) Support for specifying an error evaluation procedure and moderating imprecision; and (3) Support for specifying additional optimization criteria besides error to the learner.

Data Challenges To use deep learning for specialization, systems must provide support to collect example data and to incorporate data management into the software development process. Data collection may be static, where the programmer simply collects or creates a set of representative examples. In that case, compiler or static analysis support is required to take a collection of data and pass it to the deep learner so it can specialize the implementation.

Error Challenges How do programmers specify an application's tolerance to error? The simplest way to specify error computation support is for the programmer to provide a reference implementation of the program to be specialized and a metric function that computes the difference between two points in the output space of a computation (e.g., using probabilistic post-conditions [3]). In this case, the deep learner can quantify how specialization impacts an application's quality.

Even this simple approach to handling error presents challenges. Programmers must understand, quantify, and bound the error of the output of their computation. For many programmers and tasks, doing so may be unintuitive and difficult. We need programming model support for reasoning about how a specialized implementation's error affects a system's overall behavior.

Once a specialized implementation is learned, it may be beneficial to track error as it propagates through layers of the system both to help diagnose bugs and enforce correctness. We envision using Uncertain<T>, a generic type and runtime which provides strong guarantees about the total error in a computation by propagating uncertainty through an entire program's execution [1]. In addition to software support, systems may implement debugging and correctness support in hardware, calling for ISA extensions to propagate error from software to hardware and *vice versa*.

Optimization Challenges Deep learning traditionally aims to minimize the output for a set of examples and ensure that a learned data representation and algorithm generalize to unseen, similar examples. Using deep learning for automatic specialization creates the opportunity to add extra terms to the deep learner's minimization goal that optimize for other desirable system properties, like constraints imposed by target hardware.

For example, minimizing the size of the data representation may be beneficial if a system has hardware support for operations on small vectors. Likewise, a specialized implementation may be mapped to programmable hardware, like an FPGA, which thus imposes hard constraints that limits logic depth, buffer size, and logical complexity.

Automatic Specialization Now!

As an early illustration of the ideas we proposed in this work, we built a simple automatic specialization system that uses a one-layer learner. We used our system to automatically specialize a computation task that stores images and then performs face recognition. In particular, we used our system to learn a data representation based on a small set of 10 standard, 32-bit RGB training images taken from a face recognition benchmarks suite. We used a learning optimization that limited the amount of time the learner was allowed to run (1 hour) and the maximum size of each pixel in the data representation (16 bits). Our learner took advantage of the fact that pixels in images of faces reside in a subspace of the RGB spectrum and found a specialized, 16-bit pixel representation. The specialized representation was approximate and introduced imprecision. We wanted to evaluate whether the imprecision introduced by our specialization was problematic. We did so by running face recognition over the original data representation (with 32-bits) and the specialized representation (encoded to 16 bits, then lossily decoded to 32 again). We found about 0.1% face recognition error using the specialized implementation.

We are building on our experience with this application. Our work going forward is to build a memory system that automatically specializes its data representation based on data characteristics. We are working on an encoding and decoding memory interface that reaps the benefits of specialization with little programmer overhead. We see automatic specialization in the memory system as a good starting point for more general research into the use of deep learning for automatic specialization.

References

[1] J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain<T>: A First-order Type for Uncertain Data. In *ASPLOS 2014*.

[2] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In *PLDI 2014*. To appear.

[3] 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 *PLDI*, pages 164–174, New York, NY, USA, 2011. ACM.

[4] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *PLDI 2013*.