Consider learning to use a new machine, draw a picture, or speak a new language. Humans can learn to solve all these problems, and an endless range of other tasks also. They can do so from relatively modest data, can explain what they’ve learned in ways that other humans can understand and build on, and can creatively compose their knowledge to produce new qualitatively different concepts and artifacts. If we want to one day build machines that do the same, we need a style of learning and knowledge representation that supports compositional reuse, so that what is learned can be explained and transferred to new problems; while also flexibly adapting with experience, to overcome the brittleness of hand-coded representations. My work investigates how program induction – learning programs from data – can contribute to the engineering of these aspects of intelligence. I’ve focused on three different angles on this approach:

- How can we use structured representations to bridge low-level perception, for example from pixels, and high-level symbolic reasoning, like the kind found in a programming language?
- If programs encode solutions to learning and reasoning problems, then how can we learn to write code, and in doing so, learn to learn, and learn to reason?
- How can machines discover knowledge about the causal processes in the world, in an interpretable format which humans can understand and contribute to?

My work on these questions integrates neurosymbolic setups for interfacing with perceptual data and guiding combinatorial program searches; hierarchical Bayesian approaches, for learning-to-learn from sparse data; and program synthesis techniques from the programming languages community. Below are overviews of progress toward answering these three questions, followed by future challenges.

**Inferring visual programs from perceptual data**

Vision is rich—we perceive objects, parts of objects, and relations between those parts— and vision is also abstract: humans can perceive the symmetries in a spiral staircase or starfish, or the repetitive structure of a metal chain or a wire fence. We use these abilities productively: we impute missing object parts in repetitive scenes based on the scene’s global structure, and we can effortlessly extrapolate our percepts, like when a child sees an interleaved Lego brick wall and then constructs a much bigger version.

To build machines with a similarly high-level understanding of visual scenes, I’ve cast different visual perception problems as program induction. Generative graphics programs can capture repeated parts via iterative loops, geometric relationships via algebra over coordinates, and symmetries via higher-order functions. In collaboration with Dan Ritchie at Brown, I introduced a system which instantiates this idea by learning to infer graphics programs from hand-drawn images. These programs (Fig. 1a) generate diagrams inspired by the figures that we make in our machine learning papers, but also more broadly by the kinds of high-level visual perception that comes naturally to us as humans, even without formal training in geometry [1].

The system supports strong abstract generalization: it can extrapolate the visual patterns in an input drawing, constructing in zero-shot a much larger version (Fig. 1b). Imagine sketching the start of a repetitive graphical model or neural network architecture, and having a machine automatically infer and continue the pattern. The end state of visual program inference supports this zero-shot generalization in ways that just detecting the objects in the scene would not; it essentially provides a high-level description of the input image, and this top-down program description can also be used to correct for mistakes in bottom-up image parsing by preferring low-level parses which lead to more parsimonious programs (Fig. 1c).

This work has contributed to a growing body of literature showing that program induction from images can support the kinds of strong generalization and top-down influences upon perception that many believe necessary for more flexible and sample-efficient learning from perceptual data. After its publication at NeurIPS [3], other researchers applied this framing to naturalistic images [4, 5] and to 3D models, where I played an auxiliary role in applying this framing to ShapeNet models [6], and took a lead role in applying it to 3D CAD models [2]; see Fig. 1d-e. But fully realizing the goal of synthesizing programs from perceptual input will require new techniques for teaching machines to write code, and these 3D extensions of my framing required new algorithmic developments. Next, I’ll cover the developments that are making progress toward learning to synthesize the kinds of programs that are most relevant to AI.

**Learning to build programs**

Can machines learn to write code? Classic program synthesis views each synthesis problem in isolation. Yet when humans write code, they draw on prior coding experiences to inform and constrain the program-writing process. They leverage algorithmic building blocks created when solving earlier problems, and can intuit, prior to even solving a problem, which of these algorithmic building blocks are likely useful, and where to use them.
for (i < 3) rectangle(3*i,-2*i+4, 3*i+2,6) for (j < i + 1) circle(3*i+1,-2*j+5)

reflect(y=8) for(i<3) if(i>0) rect(3*i-1,2,3*i,3) circle(3*i+1,3*i+1)

Figure 1: (a) bottom-up neural network and top-down constraint-based program synthesizer learn to infer graphics program from drawing. (b) automatically extrapolating drawing by increasing loop bounds. (c) when bottom-up network makes mistakes, automatically fix these by preferring simpler programs. Corrections in red. (d-e) latest work [2] extends to 3D programs.

Drawing on these observations, I’ve built a system that learns to code, and which automatically discovers these algorithmic building blocks while solving a corpus of programming tasks. Jointly the system trains a statistical model (a neural net) to predict which discovered building block to deploy at each stage of program construction. This inventory of algorithmic building blocks implements a symbolic “library” of functions useful to the domain. We’ve found this neurosymbolic combination of explicit symbolic knowledge (the library) and implicit statistical intuition (the neural network) to be critical for solving many diverse classes of problems. In [7, 8] I describe applications to eight domains (Fig. 2), including classic problems like generating text, creative domains like drawing pictures and planning how to build structures, and different kinds of equation discovery. In addition to conventional program synthesis domains, these problems align with AI goals such as planning, generative modeling, and inductive reasoning. With this generic toolkit for learning-to-program, these AI problems can be productively viewed through the lens of program synthesis.

The system discovers interpretable, human-understandable code both in its learned library and in its inferred problem solutions. This interpretability hinges upon library learning: For example, when synthesizing physics equations from simulated data, it finds a short, 6-token program for Coulomb’s law. This program works by invoking induced library routines for a general inverse-square law schema, and an induced routine for vector subtraction: the system was not told about vector algebra, and had to rediscover these latent operators. The equivalent Coulomb’s law program rewritten without its learned library is uninterpretable and so long (50 tokens) that is effectively out of range of a reasonably bounded search. In short, by synthesizing its own symbols, the system outputs become both more compact, and also more interpretable.

These systems interleave program synthesis with library learning, incrementally growing a library by recursively composing functions learned earlier into larger and more powerful routines. It builds this library by refactoring its code, compressing out reused program pieces to maximize a Bayesian criterion. But the number of refactorings grows exponentially with program size. For example, when given the Y combinator and tasked with recursive programming exercises, the model refactors its solutions and rediscovers ‘map’, but in this example there were $\approx 10^{14}$ possible refactorings, only one of which involves reinventing ‘map.’ To tame this combinatorial explosion I developed a new algorithm that integrates version space algebra with ideas from equivalence graphs to efficiently explore the space of refactorings: it calculates the ‘map’ refactorings in minutes using a version space data structure with $10^6$ nodes, which would otherwise take centuries to explicitly enumerate.

This work took inspiration from library building, which is only one of the ways human programmers build increasingly complex and powerful software: they also use debuggers, version control, and interpreters, any of which can be reinterpreted and repurposed as a tool for a program induction system. I believe program synthesis still has a lot to learn from this toolkit, and in 2019, I organized a team with three other grad students with the goal of exploring how an interpreter could be used by a learned program synthesizer. Our model, published this year at NeurIPS [2], teaches a deep reinforcement learning agent to interact with an interpreter while it writes code. This new model scales to much longer programs than previous approaches, synthesizing 29-line graphics routines in seconds, and outperforms state-of-the-art neural program synthesis for text editing.

While program induction is one of the oldest theories of learning within AI, it has largely been studied in theoretical and small-scale contexts, or in specialized applications. I believe these approaches—taking inspiration from the tools human coders
use, and combining neural and symbolic representations, both jointly learned—paves the way toward more general ways for learning to program, which helps make program induction more generally useful for AI.

Building models and discovering knowledge

One of the most distinctive features of human learners is our ability to build explanatory causal models and theories: this is most evident in the historical development of science, but it also happens on a smaller scale in children’s learning of intuitive theories, and in everyday thinking, like when we learn to play a new game, or learn to use a new device. Yet model structure learning remains a notoriously difficult challenge. To help get theory induction off the ground, I proposed both a new testbed domain and a new style of approach. I propose that we study synthesizing theories of natural language grammar, for several reasons: there are corpora from many diverse languages to benchmark our algorithms; both children and linguists learn these grammars from modest data, so the problem is tractable given the right inductive biases; and decades of research in computational linguistics gives us ready-to-deploy computational formalisms.

I’ve been collaborating with Timothy O’Donnell, a computational cognitive scientist at McGill, to both build a benchmark for linguistic theory induction, and to see what progress we can make by viewing these language learning problems as program induction [9]. We’ve discovered that correctly solving these linguistics problems requires a finely-tuned inductive bias (or “universal grammar”), but that much of this inductive bias can actually be learned: instead of giving the system a single language to analyze, I give it a spectrum of 70 datasets from 58 languages, and then it does hierarchical Bayesian inference over both each language-specific model, and of the “metamodel” that constrains and biases the space of allowed models. Each language’s data set is modest by NLP standards – at most a couple hundred words – and span a diverse range of the world’s languages, from Russian to Tibetan to Kerewe. Yet children and linguists can pick up on all these patterns very quickly, and linguists can articulate their knowledge of these language patterns in ways that other scientists can understand and build on.

We were excited to see that our program induction model, with the right learned inductive bias, could behave similarly. Within computational linguistics, ours is the first model to successfully acquire so many highly diverse linguistic phenomena, and a team of computer scientists and linguists at UCSD have recently published a paper [10] using our benchmarks as a testbed for new grammar induction algorithms.

Beyond language, one of the most useful model building activities for an embodied agent is to construct causal theories of the physical world, of objects and their physical properties. The program induction toolkit developed here has matured to the point where it can make real traction on problems like these: we can infer structured 3D programs from perceptual input, grow reusable libraries of physical concepts like ‘table’, ‘gear’ or ‘pulley’, and learn-to-learn these models from less data by sharing statistical strength across related problems.

Looking forward

Despite much progress in AI we remain far off from machines that come close the human abilities to solve a diverse range of problems, while using prior experiences to learn-to-learn to solve those problems more quickly and effectively, simultaneously structuring and communicating the knowledge they distill from their prior learning experiences in compact, understandable forms. My future research will push toward making machines more intelligent along these dimensions, and while my work so far has largely studied how program induction can contribute to this picture, the most important outstanding problems involve connecting structured knowledge (including but not limited to programs) to broader issues surrounding machine intelligence.

A critical ingredient in both human and machine intelligence is compositionality, the structured and systematic reuse of representational subcomponents. In the era of big-data machine learning, we’ve enthusiastically adopted approaches for representation learning, but have mostly sidelined rich composition. The first waves of AI may have been limited, but compositionality played a central role. I think the future of the field will lie with approaches that build on compositional insights, but which learn the underlying representations, interleaving program-like structures with modular neural components. These integrations are most immediately useful for perceptual programs (e.g., graphics or visual routines) but also speak to broader issues of knowl-
edge representation: combining the best of the neural and symbolic traditions, while keeping explicit compositional structure, but learning it from experience.

If program induction is going to evolve into a standard part of the AI toolkit, then what we need is an analog of an ‘Imagenet moment’, a challenging AI task where the performance of these new program induction techniques far outstrips the prior state-of-the-art. Automated model building is a promising candidate. One version of this is sample-efficient model-based reinforcement learning, such as solving Atari from minutes of gameplay with modest prior knowledge. But this would still be a small-scale demonstration, and the real long-term vision is to scale these kinds of automatic theory induction, like my work on theories of natural language, toward actual models of the world. These future automatic model builders could prove critical for sample-efficient robot learners, and for future induction algorithms that could help scientific model building.

AI research is improved through interdisciplinary connections. My first work on visual programs was a collaboration with Dan Ritchie, now a graphics professor, and my later work (in prep) tackles modern CAD problems, in collaboration with Wojciech Matusik, a graphics professor at MIT. The root motivation for my research thus far has been taking inspiration from human intelligence, and program induction is closely allied with ‘probabilistic language of thought’ [11] models in cognitive science. My collaboration with Tim O’Donnell draws upon this common ground.

My work thus far has focused on how program induction can contribute to machine learning, but this toolkit goes both ways. Learning reusable libraries of routines is important for AI, but also for program synthesis; neurally-guided program search helps make synthesis scale for AI problems, but also for computer-aided programming. I believe these techniques can circle back to help further move program synthesis out of the lab and into the real world, impacting software engineering and computer science more broadly.

References


