Notes on applying adaptor grammars/combinatory logics to Bonan's project

Chris Lucas

2019-11-26

This won't make sense without having looked at Liang et al., but hopefully it will clarify some aspects of the paper, which introduces two key concepts: (1) Combinatory logic as an alternative to lambda calculus; (2) Adaptor grammars as an alternative to PCFGs. I recommend understanding them one at a time; if you go directly to trying to make sense of their combination, it could be difficult. For adaptor grammars, I recommend reading the original paper by Johnson et al., and perhaps Tim O'Donnell's book (I've only skimmed the latter but it looks accessible). Don't worry about the paper's discussion of refactoring moves – they're a way to do inference efficiently.

I'll focus on an example of a program expressed in combinatory logic and how it can be sampled, rather than getting into adaptor grammars. Adaptor grammars are important for transfer learning, but for a single task all they really do is allow recursive functions; let's not bother with them for now.

Notation

A few bits of notation that I've introduced or are worth repeating from Liang et al.:

- o: Object (f_1, f_2) where f_i denotes the i^{th} feature; let f_1 be shape and f_2 be color.
- s: Intermediate type; see Liang et al. for details.
- $x\ y\ z$ as a function: Shorthand for $(x\ y)\ z$, which is to say that x is called on y, and the result of that is called on z. We can express any function in this "curried" form. For example, multiplication is $*\ x\ y$ where if we pass it one number, it returns a unary function that multiplies the input by that number. Unlike lambda calculus, there is no need to bind anything to variable names.

Grammar

Here's a minimal grammar that can express the example as well as some other relationships like "magic stones turn everything purple."

Primitives and their types:

- The features themselves. Shapes have type f_1 and colors have type f_2.
- $bind(f_1, f_2)$, which has type $f_1\ f_2\ o$. For example, $bind\ square\ red) = (square, red)$.
- $get_1(o)$ and $get_2(o)$, which have the types of_1 and of_2. For example, $get_1\ square\ red)) = square$.

Some natural extensions of the grammar might be:

- Booleans
- equality of features, $f_1\ f_1\ bool$
- logical operations including AND, OR ($bool\ bool\ bool$), and NOT ($bool\ bool$)
- if statements, e.g., $o\ o\ bool\ o$.

Getting fancier, we could add sets as a type, and include map, filter, and reduce functions.

Getting fancier still, we could introduce probabilistic functions, e.g., "red half the time, blue half the time", which would be more elegant than just having exception probabilities.

Example: Recipient's color (f_2) matches the cause's color

For the stone scenarios, we want a function from a cause object (stone) and an effect object to the object the effect becomes. In the notation of Liang et al., our target type signature is $o\ o\ o$.

In order to walk through the steps involved in sampling from PCFGs, described in Figure 3 of Liang et al., let's imagine we've gotten lucky and sampled exactly the right function, namely,

$$g((f_1, f_2)_1, (f_1', f_2')) = (f_1, f_2').$$

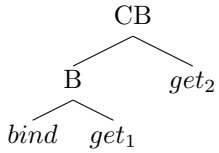
How would that happen?

Steps:

1. Sample a router. In this case, suppose we've sampled CB, which means "route argument 1 to left side, route argument 2 to right side". There are three routers of order 1 (C,B, and S), 9 routers of order 2 (CC, CB, CS, ...), and so on.
2. Sample an intermediate type s – we can think of this as the "bridge" between the left and right side. Suppose we sample f_2 (color) as the type.

3. Based on our router and s, for our right side we need something with the signature $o\ f_2$.
    - Here, get_2 has the type signature we need; we'll use that.
4. For the left side, we need something with signature $o\ f_2\ o$. The gist of the function we want is "overwrite the color of the first argument with the second argument".
    - There isn't any primitive function of that form, but we can create it by applying our same procedure recursively: the right side is of_1 (where get_1 is a matching primitive), and the left side is $f_1\ f_2\ o$, where *bind* is a primitive with the correct signature.

Our final program can be expressed as a tree:



We can get the (un-normalized) probability of this program by looking at every place we drew a sample:

- The router. If we imagine that we're just considering order-0, order-1, and order-2 routers and believe all are equally likely there are 1+3+9 options: 1/13.
- The intermediate type. We might keep things simple and restrict our prior over types to support just f_1 and f_2 (or assign probability epsilon to more complicated types), giving us 0.5.
- The probabilities of the primitives. Here, this are one because we just have one primitive for each type.
- The probability of sampling the right side – we apply the same process recursively for any composed programs.

Inference

As you may have noticed, the branching factor in this space is pretty large, and obviously it's impossible to compute a normalization constant over an infinite space of programs. The search problem gets worse with adaptor grammars, because there's an ever-growing set of cached programs as new tasks arise, and random sampling can create arbitrarily deep trees. In summary, inference under this model is really really hard.

One response is to enumerate all programs to a very shallow depth – a kind of hard simiplicity bias. Dechter et al. combine such an approach with pruning, selling the approach it as a kind of "conceptual bootstrapping", which is appealing, their approach requires 0/1 loss and one could be more Bayesian about it.