Learning, Searching, Querying, Parsing

Linas Vepstas

31 August 2020

Abstract

Shifting from a viewpoint of "variables and values" to a viewpoint of "connectors that connect" unites the theories of search (query and query languages) with the theories of parsing (deducing structure from input lacking structure). This provides a new and yet very natural theory of learning. Most notably, it solves the chicken-and-egg problem of "how do you search for something when you don't know what you are searching for?". It then recasts learning as a process that begins with a search for unknown things, and then categorizing the new discoveries based on observed relationships.

Introduction

The concepts of querying, parsing and learning are normally considered to be quite distinct. This text unifies these concepts by replacing the concept of "plugging in a value into a variable" by the concept of "connecting two things together". This text is part of a series of related texts, exploring and articulating a connectionist viewpoint of knowledge representation and knowledge discovery.

One of the central distinctions is that between "connecting" and "beta–reduction". The metaphor for "connecting" is the jigsaw-puzzle piece: a shaped item with connector tabs that can be joined together, only if they mate. The different shapes of the connectors are "types", in the formal sense of "type theory". This is contrasted to "beta–reduction", which is a technical term arising from lambda calculus. The metaphor there is that of taking a function, say $f(x)$, and plugging in a value, say 42, substituting it for x to obtain $f(42)$. The plugging-in, as described here, also feels very connectionist. It is even typed: one might say that if x is of type "integer", then only integers can be plugged in.

The primary difference between plugging-in and connecting is how the type-mating is achieved. In beta–reduction, the type of "integers" can be interpreted to be the set of all integers; of these, 42 is merely an exemplar: it is but one. By contrast, in the connectionist approach, the types are peers, rather than being above or below one-another. The peer relationship is more appropriate for many situations. The simplest examples may come from biochemistry: when one has some protein, and it binds to some structure on a cell wall, or say, some antibody binding to some antigen, the binding is jigsaw-puzzle-piece-like: it is possible because the physical shape of proteins fit together. One can ascribe types to these shapes. What one cannot do is to say that one is the super-class and the other is merely an exemplar, in the way one had a super-class of all integers, and specific exemplar of 42. The biochemical binding process is “symmetric”, in this sense.

Conventional notions of database query have the general flavor of beta–reduction. One has some query, for example: `(match x [(list a b c) #:when (= 6 (+ a b c))])`. This example is taken from Racket, a scheme dialect. The intent here is hopefully obvious: when `x` is of type “*list*”, and, specifically, is of type “*list-of-length-three*”, then accept the match if the arithmetic sum of the elements of the list total up to six. The search proceeds by process of “plugging in”: we find things, plug them in, and either they fit or they don’t. This is conventional pattern matching in it’s most basic form.

The distinction between “plugging in” and “connecting” is quite subtle, and one can easily get lost in technicalities. The goal here is to tackle the technicalities head-on, and draw in clear and vivid contrast how these two differ, in theory and in practice.

Querying

The “query” here refers to the conventional idea of database queries; the query of SQL or GraphQL. Such queries are generally formulated as a generalized “pattern matching”: one specifies a fixed part of a pattern, and a variable part, to be filled out. A textbook example of an SQL query might be “find all employees in sales”, which is understood to be the directive to look at the “employees” table, which contains two columns: the name of the employee, and the department they work in. The fixed part of the query is that the department must be “sales”; the variable is the “employees”. The actual syntax for such search, in SQL, is “`SELECT name FROM employees WHERE department='sales';`”. Graph query languages, and the subgraph isomorphism problem are similar in nature: one specifies a collection of vertexes and edges (which may be labeled or unlabeled, directed or undirected), taken to be “fixed”, and another region, variable, and then asks for all matching graphs that contain the fixed part. The variables are always named (unless there is only one, and then it can be anonymous): variables are X,Y,Z, ... and are generally typed (in SQL, the employee name is a string, the salary is a number.)

Consider then, the following query/pattern match. The database contains "*Mike threw a ball. Joe threw a ball. Ben ate a pizza. The engine threw a crankshaft.*", however that may be represented; for brevity, we skip the precise representation. Consider the query pattern “____ *threw a ball.*” or “*X threw a ball.*” in which *X* is a named variable. This is a basic fill-in-the-blanks search. The variable *X* may be typed: "*X must be a noun*" or perhaps "*X must be a given name*". Typing helps narrow the search, and make it more precise. In practical applications, typing often provides a huge performance boost, and avoids paradoxes and ambiguity. Anyway: this search is straight-forward, and yields the expected results.

Consider now the case of relational data. Suppose the database indicates relationships; for example:

```
    +------>WV------>+-----Os----+
    +-->Wd---+--Ss---+     +-Ds--+
    |        |       |     |     |
LEFT-WALL  Mike    threw   a   ball
```

This is a graph, indicating syntactic relationships: `S` is a subject-verb relationship, `O` is an object-verb relationship, `D` is a determiner, and so on. Consider the query "*connector-S threw a ball.*" This is a relational query; we are asking for all things that have a relation `S` to the fixed part of the query. For the same dataset, this query returns the same results. Yet the flavor of the query is different.

Perhaps this second form seems arcane, or the doorway to some sneaky cheat. In this case, a simpler way of posing the example would be the query "*(things that have the property of being to the left of) threw a ball.*" The "*things that have the property of being to the left of*" is a relationship. Let's call this relationship "`S`" for short. It is not a property of the thing itself (there is no "left-sided-ness"); it is a property of the relation.

Relational queries are fundamentally different than variable queries. For the first query form, "*noun*" and "*given name*" have to be known, tagged properties of "*John*", "*Mike*". Somehow, through some magic pre-processing, these nodes had been tagged with these properties, so that they are known before the search is started. Failure-to-tag means failure-of-search. If the database contains "*Flooglebarf threw a ball.*" and "*flooglebarf*" had not been previously tagged as a noun, the search will fail to find this candidate answer.[1]

For the second search form, the properties of "*Mike*" (or "*flooglebarf*") are unknown, not yet worked out or tagged. We just know that "*Mike*" participates in a network of relationships, in this case an `S`-type relationship. Oh wait, but `S` is a type! ... but that does not mean that "*Mike*" is of type `S`, only that "*Mike*" participates in a relationship of type `S`.

This allows one to find things, without knowing what they are in advance. One may have observed a network of relationships, but not know (yet) the properties of the thing-itself. Perhaps later, you are able to deduce that "*Mike*" is a noun, perhaps that it is a given name, and maybe even that "*Mike*" is a specific individual at some given point in space and time. There was no need to know any of this, before starting the search for ball-throwers.

The first form has a chicken-and-egg problem: how can I know "*Mike*" is a person? Because I saw Mike do the kind of things that people do. But how can I see Mike? If I can only see things of type "people", and if I don't already know that "*Mike*" (or "*Flooglebarf*") is of type "*people*", then search result fails; or rather, it fails to return these as answers to the question.

The second form does not have the chicken-and-egg problem. One may observe things participating in some relationship; after a while one may deduce that those things are "*people*" (have type "*people*").

[1] Caution, do not be confused. The query was "*X throw a ball where X is of type noun*". Of course, one may attempt untyped-variable searches. The discussion here is trying to pry apart the subtle distinctions between variables and relationships, and the role of typing in each.

Learning

The distinctions above may seem to be subtle, if not outright gratuitous or even confused and delusional. One may be tempted to make the claim that “*Mike*” is of type `T` that is able to participate in a relationship `S`. This would then be followed by a claim that the two forms are isomorphic and can transmuted into one another. In some abstract sense, this is correct. In practice, however, the type of `T` is much weaker than that of “*noun*” or “*given name*”.

Observing a relationship predates assignment of a type. Say, you are on a hike, and someone asks "*See that thing, next to the tree? What is that?*" The typing applied here is quite weak: the type `T` is the type "things you can see with your eyes". Of course, it’s that, but that is almost a universal type; it is much weaker than identifying the thing itself.

Suppose the thing is an antelope -- no one ever asks "*See that thing of type antelope next to the tree? What is it?*" Such a question presumes knowledge of the answer; the type of "things that can be seen" is weaker/broader than the type of all antelopes. The lesson here is that the relationship-based query is one of type induction: a narrowing of types from broad to fine, a movement from a low-knowledge state to a high-knowledge state.

A more concrete example might be looking at how the Link-Grammar (LG) parser deals with unknown words.[2] Consider the sentence "*flooglebarf threw a ball.*" The LG parser will parse it just fine. Now, "*flooglebarf*" is not in the dictionary of known words, so some guesses are made. The obvious first guess is that "flooglebarf" is a word *i.e.* it has type `T`=”*word*”. But this is kind of silly, because words are all that there are, for LG. Yes, it’s a type, but it’s the universal type.

The LG guessing rules include descriptions of generic nouns, generic verbs, generic adverbs, *etc.* and each of these are tried in turn. Eventually, a valid parse is found. Only *after* a valid parse is found, can one conclude that "oh, ah-hah, *flooglebarf* is of type “*noun*”, because nouns can participate in relationship `S`". Before parsing, the true type of “*flooglebarf*” was unknown.

To summarize, the claim that "*flooglebarf*” (or “*Mike*”) has some type `T` that is able to participate in a relationship of type `S` is tautological; its vacuously true. In practice, one doesn’t know what `T` is, until *after* the relationship was observed! This is the "chicken and egg problem" of typed-variable searches.

Parsing

The presentation of a connectionist approach to query above may seem to have employed a bit of switcheroo or mis-direction. After all, the initial discussion starts with conventional SQL queries, and yet it somehow invokes parsing as a prop or defense for the claims. Surely this is absurd?

Consider the following pattern-matching/query/search problem: “Here is a block of text, find all of the nouns in it.” You have to parse it, to find the nouns. Oh, but

[2]It is the software that produced the diagram above.

wait, you say, you can just POS-tag it![3] Of course you can: the POS-tagger has a built in list of nouns; if “flooglebarf” is not in the dictionary (and it isn’t), it will not be tagged as a noun. This is not SQL, where you have a table called “wordtags”, with one column called “words” and another called “pos” and you issue the search “`SELECT word FROM wordtags WHERE pos = 'noun';`”. Parsing is a different kind of activity, which can be used to generate tags from a set of relationship-rules. The tags are initially unknown, they are inferred.

This example may still appear to be silly, so let’s rephrase it in terms of deep-learning, artificial neural nets (ANN), and images. The pattern-matching/query/search problem is now: “Here is an image, find all of the animals in it”. If the ANN doesn’t know about antelopes, then the antelope in the image will be invisible to the ANN. It cannot be found. To do better, you have to parse the image, literally, to look for appropriate things: not-too-big, not-too-small, four-legged with a head, upright and not sideways, on the ground, not in the sky, center-of-mass appears balanced over legs...etc. Those constraints -- "is on the ground" -- those are "relationships" and we are performing a query to find all things satisfying a collection of relationships. If you see something blurry in the image, you try to figure out it’s relationship to everything else. At the end of the process, not at the beginning, can you say "ah hah, that thing is of type `T`".

Conclusions

“Oh,” you might say “ANN’s are smarter than that”. And perhaps they are: it is not entirely clear if they are lookup engines, or whether they perform analytical (parsing-like) processes under the hood - they are a bit of a black box, as to what is actually happening in there. The texts in this series hope to (eventually) elucidate this question, and determine more precisely what it is that ANN’s actually do, and how they compare to conventional symbolic AI.

The goal of this text was to lay the foundational underpinnings, to clarify the distinction between lookup and inference/deduction. Lookup is just “tabular lookup”, which has been painted here to seem a bit trite. Of course, vast tracts of quite advanced, sophisticated and complex theory has been devoted to this: the relational algebras as the formalization of SQL (as well as it’s categorical dual, noSQL). Much of this formalization can be teleported onto GraphQL and related query systems. The primary point here is that, to use SQL, you must first construct a table: one must have columns, one must populate those columns with consistent types. This happens *ab initio*, outside of SQL itself. Some other, earlier process generated the knowledge; SQL just queries it for you.

This is in contrast to parsing. It is an equally mechanical, algorithmic process; parsing also has a large deep and arcane literature behind it. The inputs to parsing are different: the inputs are unstructured, the relationships are unknown. Rules are applied to determine relationships, but the rules are applied not one-at-a-time, but together. One obtains a valid parse only if multiple rules can be joined, connected together, so

[3]Part Of Speech, or POS, for short.

that they fit the data. So, yes, of course: a parser recognizes a sentence drawn from a language if that sentence obeys the syntactic, grammatical rules. But that is not the point: the point is that it is an assemblage of those rules that accomplished the recognition task. The conclusion of the assemblage is that the unformed, unstructured input is now tagged with syntactic information.

By looking at search/query in terms of a connectionist viewpoint, the lookup of query languages can be unified with the parsing of "rule engines". It offers a simpler, easier foundation than the lambda-calculus-style beta-reduction of plug-a-value-into-a-variable approach.

Lest the reader object that this is all absurdly abstract pie-in-the-sky daydreaming, the author would like to remind the reader that a proof-of-concept of such a system already exists, and it is small, fast and efficient. Specifically, the Link Grammar parser. It is explicitly built on these connectionist principles. It is fast: parsing hundreds of sentences per second, and it is accurate: it appears to be the most accurate parser of the English language currently in existence.

Again, the goal of these texts is to elucidate these principles well enough that they can be clearly applied to AGI in a full general setting. This includes not only unstructured visual and audio inputs, but the full process of observation and deduction from observation.