

Logic in Hilbert space

YKY

January 28, 2021

Summary

- It seems possible to construct a model of the **untyped** λ -calculus in Hilbert space, with function application $f(g)$ implemented as $\llbracket g \rrbracket \circ \llbracket f \rrbracket$.
- Doing so allows **self-application** of logic predicates (Curry's paradox can be avoided by the fuzzy truth value “don't know”)
- The notion of **continuous maps** may be advantageous in machine-learning because **generalization** seems to work best with “continuous” domains (as opposed to maps acting on symbolic logic representations which may be discontinuous).
- Elements in the infinite-dimensional \mathcal{H} can be realized on a computer as **neural networks** (which are functions $\mathbb{R}^n \rightarrow \mathbb{R}^n$ **finitely** generated from sets of weights).

0 Inspiration

In the 1960's Dana Scott constructed a model for untyped λ -calculus, using a domain D_∞ with the property $D_\infty^{D_\infty} \cong D_\infty$. This started off the field known as **domain theory**.

Scott initially believed that such models cannot exist, but later discovered that they can be constructed. In retrospect, this is not surprising because the Church-Rosser theorem demonstrated that the untyped λ -calculus is consistent.

Scott's idea is to begin with an initial domain D_0 and define D_{n+1} to be the function space $D_n \rightarrow D_n$.

Thus it is guaranteed, for any domain $d \in D_\infty$, one can always find a function space $d \rightarrow d$. Therefore the space D_∞ is isomorphic to $D_\infty \rightarrow D_\infty$.

The detailed definition of D_∞ involves building a cumulative hierarchy of infinite sequences, with pairs of operators ψ_n, Ψ_n going up and down levels. For a gentle exposition one may read (Stenlund 1972), Ch.1 §6.

1 Requirements

In order to define a logical calculus in \mathcal{H} , what we need are:

- a family of functions dense in \mathcal{H}
- self-application: maps can act on other maps; how to define $f(g)$?
- define the **S**, **K**, **I** combinators in combinatory logic
- how does an element $e \in \mathcal{H}$ translate to and from a (syntactic) logic formula?

2 Elements in \mathcal{H}

The structure of D_∞ is reminiscent of Cantor's ordinal number ε_0 :

$$\varepsilon_0 = \omega^{\omega^{\omega^{\cdot^{\cdot^{\cdot}}}}} = \sup\{\omega, \omega^\omega, \omega^{\omega^\omega}, \omega^{\omega^{\omega^\omega}}, \dots\} \quad (2.1)$$

but this number is still “smaller” than the **continuum**, ie. the real line \mathbb{R} . This led me to think that models of λ -calculus may be found in the Hilbert space of continuous functions.

But such a Hilbert space would be ∞ -dimensional. Next I consider the **neural network** as a function f :

$$\begin{array}{ccc} \mathbb{R}^n & \xrightarrow{f} & \mathbb{R}^n \\ x & \mapsto & y \end{array} \quad (2.2)$$

and notice that f and x, y are “unequal” because f can **act on** x, y but not the other way round. This is partly because f is ∞ -dimensional whereas x, y are finite-dimensional. So, what if we increase n to ∞ , then perhaps x, y would become the same kind of objects as f ? In an informal sense \mathcal{H} can be regarded as \mathbb{R}^∞ .

The Universal Approximation theorem of (Cybenko 1989) (with later extensions by others) states that the family of neural networks is **dense** in the space of continuous functions, with respect to the supremum norm. This means that we have a nice way of generating elements in \mathcal{H} that can be finitely represented in a computer.

3 Function application

Now we lack the notion of a function **applying** to another function, such as $f(g)$. Since we only need the functions as elements of \mathcal{H} , the domains \mathbb{R}^n is somewhat “immaterial”. We might as well assume \mathbb{R}^n to be common among all functions (the dimension n can be fixed for implementation considerations), and thus function **composition** such as $f \circ g$ always exists. So we define:

$$\llbracket f(g) \rrbracket = \llbracket g \rrbracket \circ \llbracket f \rrbracket \quad (3.1)$$

where $\llbracket \cdot \rrbracket$ denotes “model of”. (The reversed order $g \circ f$ is to prepare for dealing with tuples; see below.)

4 Combinatory logic in \mathcal{H}

We need to implement the combinators defined by:

- $\mathbf{I}x = x$

- $\mathbf{K}xy = x$
- $\mathbf{S}xyz = xz(yz)$

which obviously requires that some functions take on > 1 arguments. So we need the notion of **tuples** and of functions applying to tuples.

Tuples

We can implement tuples like (g, h) simply by stacking them vertically:

$$(g, h) = \begin{bmatrix} g \\ h \end{bmatrix} \quad (4.1)$$

which is a $\mathbb{R}^{2n} \rightarrow \mathbb{R}^{2n}$ function. This can be extended to dimension kn . Function application can be implemented as:

$$\llbracket f(g, h) \rrbracket = \begin{bmatrix} \llbracket g \rrbracket \\ \llbracket h \rrbracket \end{bmatrix} \circ \llbracket f \rrbracket \quad (4.2)$$

where f is of type $\mathbb{R}^{2n} \rightarrow \mathbb{R}^n$. We can visualize the right hand side like this:

$$\begin{array}{c} g \\ h \end{array} \begin{array}{c} \searrow \\ \nearrow \end{array} f \text{ ---} \quad (4.3)$$

Interpreting logic formulas

It is helpful to bear in mind that a functional form

$$f(g, h) = e \quad (4.4)$$

in Hilbert space corresponds to a logic formula

$$g \wedge h \xRightarrow{f} e \quad (4.5)$$

where f plays the role of an **implication**, or logical **consequence**, $g \wedge h \vdash e$. In fact f encodes the *entire* formula $g \wedge h \Rightarrow e$, but even more than that, f can be

applied to other arguments. (Readers may recognize that the function f realizes an implication statement in logic via the **Curry-Howard isomorphism**.)

With tuples, we can easily implement the combinators **S**, **K**, **I**. The treatment for λ -calculus would be analogous, and I would add that to a later version of this paper when I have time.

Arities

There is a remaining problem with arities. Because we allow tuples, compositions with tuples leave us with functions with arity > 1 . For example, the following two applications of f yield the same result c , and we would like to make these definitions of f consistent:

$$\begin{array}{ccc} \begin{array}{c} a \\ b \end{array} \begin{array}{c} \diagdown \\ \diagup \end{array} f & \text{---} & c & \quad & f(a, b) = c \\ d & \text{---} & f & \text{---} & c & \quad & f(d) = c \end{array} \quad (4.6)$$

but the **arity** of f appears different in the equations. My solution is to adjoin a **null** input to the single input, that is:

$$\begin{array}{ccc} d \\ \emptyset \end{array} \begin{array}{c} \diagdown \\ \diagup \end{array} f \longrightarrow c \quad \quad f(d, \emptyset) = c \quad (4.7)$$

In implementation, this can be done “on demand”, as the number of conjunctions in the premise of a logic rule is usually a small number.

5 Usage as AI logic

Usually there would be a “big” neural network, representing a special function f , that plays the role of a **knowledge base**, consisting of (the conjunction of) a huge number of logic rules of the form $\Gamma \vdash \Delta$. This f would send various logic formulas to their conclusions. In this sense f embodies all the *knowledge* in the AI system.

What is special with this Hilbert-space model is that we can now construct arbitrary logic formulas as elements in \mathcal{H} , which can be acted upon by f .

For example, “if we touch fire we may get burnt” is a logic formula of the form $A \Rightarrow B$. In a primitive AI (or animal), this formula is *implicitly* stored in the transition function f , which performs the logic inference $A \vdash B$. The animal or AI has no conscious reckoning of $A \Rightarrow B$. Now with Hilbert space we can create an element $A \Rightarrow B$, ie. an object that can **participate** in logic inference.

6 Conclusion

I am not highly confident of this invention, as its technical details are a bit complicated (We could build a simpler AI using a big neural network to act on *symbolic* logic formulas; indeed the BERT model is a case in point). If human-level intelligence turns out to be very hard to learn by an AGI, at least we can turn to this, more powerful approach.

Questions, comments welcome ☺

Dana Scott (1932-) and Sören Stenlund (1943-2019):



References

Cybenko (1989). “Approximation by superposition of a sigmoidal function”. In: *Mathematics of control, signals and systems*.

Stenlund (1972). *Combinators, λ -terms and proof theory*.