Lifelong Formal Modeling Agents – Definitions and Implementation Strategies

Jacques Basaldúa[1]*

[1]Lead author of the Jazz platform
Senior Data Scientist @ BBVA AI Factory

Abstract

This paper summarizes the current vision and results of our research since 2016, the beginning of our Jazz platform project. We explain the design of Lifelong Formal Modeling Agents by starting with describing and defining basic ideas and discussing their importance.

We present a complete architecture built upon these ideas, summarize our experimental results where available and discuss how they represent minimum requirements towards building reliable human understandable AI agency. Finally, we briefly touch on how these agents could produce more natural human-computer interfaces.

It should be noted that the definitions presented in this paper of intelligence, understanding, concept, object, symbol and other terms are intended for practical implementation and will not completely align with other academic definitions of the same terms.

1. Introduction

This paper could be the summary of a 700 pages handbook on intelligence engineering. Just a short description of each idea and definition, how it fits the whole design without precise implementation details. That handbook does not exist as such, but it could be written some day from our collection of papers, experimental reports and our private wiki that covers two decades of AI research starting in game research including foundational discussions on Monte-Carlo Tree Search.

Precise implementation details can be found in older papers (Basaldúa 2020a) or the implementations that have been released. This includes three different proofs of concept: JazzARC (Basaldúa 2020b), The Tangle[1] and TLSS (Bop applied to language solving the tasks from Facebook's bAbI (Weston et al. 2015) project). The Jazz platform itself contains the final form of ideas that made it into industrial quality C++ code.[2].

This work is too wide and this position paper too short to properly survey and give attribution to existing published work on similar ideas. We apologize for its low academic standards.

*kaalam@kaalam.ai

[1]https://github.com/kaalam

[2]https://github.com/kaalam/Jazz

2. Code

We focus our research on agents that use code as Knowledge Representation (KR). Agents run and learn new code by tree search. Before describing how code is implemented, we start with short arguments in favor and against code as KR.

Why code?

We can summarize classic arguments in favor of code.

Von Neumann's take on the Church-Turing Thesis Von Neumann is, to our knowledge, the first to realize that AI being possible at all is a direct consequence of two statements: "The human brain being, just like any living creature, biomolecular machinery." aka. there is no "magic" involved and the Church-Turing Thesis. The implication being that: up to speed and resources, any computing machine can be simulated on another. Von Neumann uses the phrase "the short code of the central nervous system" to refer to the code that would emulate the brain on a computer (Von Neumann 1958).

Therefore, if we believe there is no magic involved, code is enough and we do not know what the necessary minimum is.

Homoiconicity and classic AI Since McCarthy invented lisp and, according to Marvin Minsky, before that in the design of von Neumann's architecture, there was always the intention to make computers write their own code. This made the concept of homoiconicity central in AI oriented languages. *"A language is homoiconic if a program written in it can be manipulated as data using the language."* Also Winograd's work on language understanding (Winograd 1980) is about converting informal language into code. Rather than having a "We tried and failed." attitude, we embrace a "Some results were already achieved in times were transistor counts were lower that today by a factor of hundred million, CPU performance and storage by a million each, just imagine what we can build today!"

All models are code anyway, why restrict ourselves Schmidhuber, already wrote *"Let us view a network with a fixed topology as a computer. Its program is the weight matrix."* over thirty years ago (Schmidhuber 1990). Of course, everything is code. Therefore, we have two options:

1. Do everything within algebra and end-to-end differentiation, which results in better learning and much less efficient running. Wait until the convolutional neural network learns to multiply, so to say.
2. Use code, multiply with a single CPU instruction and find new ways of learning even if they are less efficient based on "To a smart agent, you only have to teach something once." principles.

Biological inspiration Another way to put is could be: Code is 3.7 billion years old, at least on Earth. Life is code and it created the human brain which is our "baseline implementation".

Code and complexity

The main argument against agents built on self learned code is the size of the space of formal systems.

Tackling combinatorial problems Informally, we can talk about "the class of combinatorial problems" as what we always work with in computer science and AI. Problems that involve searching immense trees or assigning values to weight matrices. These problems can be as big as the 170 billion parameters of a large language model, but we can still tackle them.

What we cannot do is "brute forcing" them, since they are too big for that, but we know how to use all sorts of tricks to tackle them. E.g., using overparameterization to make "good enough solutions" so abundant some can be found by gradient descent.

They are, so to say, the "bread and butter" of computer science work.

Tackling formal problems On the contrary, formal problems generate search spaces so big that the space of combinatorial problems becomes a set of measure zero in them. In the previous case, we mentioned a search space of $n^{170\vec{billion}}$ as immense but doable. Using a formal language we can express numbers like: that number raised to itself applied itself number of times, etc. And if that wasn't bad enough, we have to consider the halting problem. If we allow code to do endless conditional looping or endless conditional recursion, waiting for a program is unfeasible. What we are doing is some form of reinforcement learning and endlessly running programs return no information.

The sad truth is we don't know how to explore the space of formal problems. It is useless beyond theoretical implications like Solomonoff induction.

Forward running code Therefore, to make learning code feasible, we have to run a large amount of "candidate programs" and that requires some restriction. In our initial work (Basaldúa 2020a) programs were sequences of opcodes. Conditionals were only possible through inhibitory mechanisms in a more biologically plausible way.

Now, running code (imperative code) is still a sequence of opcodes, but it is generated from declarative code by a process called **interrogation** (see section 3). We use "forward running code" to name the whole idea. You can see it like walking down a tree, different decisions generate different code deterministically but, whatever code is generated, jumping backwards is not allowed. Recursion is possible through field inheritance, but has a cost and will rarely be deep.

Opcodes may be kernel bytecode or **Bop** statements. Bop is short for **Bebop**, the language of Jazz.

Bop statements are one of:

- Constants
- Pure functions (from objects to objects)
- Methods: Functions with access to two **fields**. Implementation-wise a **field** is an abstract parent of both formal fields and semspaces described below. There is no loss of generality from limiting it to two, since methods can fork fields building arbitrary binary trees of fields.

With this setup, learning to code is feasible. It is a tree search. The search can be very big, depending on the problem, the domain and how much information the **reward** and **target** (see below) provide. It is a reinforcement learning problem and can be addressed with RL or DRL methods such as alphaZero (Silver et al. 2017).

3. Objects, concepts and symbols

An established definition of "symbol" would be: *A mark used as a conventional representation of an object, function, or process.*

Our definition is:

Definition 1 *A **symbol** is a token that is unique in some **field** and represents a concept.*

Our approach is –if you need to use that word– symbolic[3], in the sense that our knowledge representation is a sequence of symbols. Symbols are indices to concepts. For now, concepts include Bop statements. Therefore, they are not limited to a form of logic, probabilistic reasoning or algebra. They represent code which is general and includes all that and much more.

We have to introduce a key idea here: they cannot just represent objects. We need to represent **concepts as something other than objects**.

In modern AI, embeddings are used to represent concepts, images, sentences, etc. This allows using algebra in high dimensional spaces to define relations between them, even across different classes (e.g., mapping images to texts). While this has lead to unprecedented advances, it cannot possibly define concepts that are unlimited by nature. You cannot fully represent *Germany* as a tensor. You could have an object-Germany in a board game that is a fixed size data structure, but the real *Germany* is *a concept*. As knowledge representation in an agent, it will always be: subjective, incomplete and forever updating.

So, let's start by defining "object".

Definition 2 *An **object** is a tuple of tensors.*

[3] And, of course, connectionist. It is connections all the way down. And why not, neural, it uses deep learning where appropriate.

We use the term ”object” to mean any fixed-size data structure, typically a tuple that can combine different types of data, including single values.

We need dynamically allocating structures to represent concepts. We start defining a concept by what it is not.

Definition 3 *(Preliminary definition) A **concept** is a dynamic knowledge representation that cannot be represented as an object. Only the subjective state of a concept at a given moment in time can be serialized as an object.*

This preliminary definition highlights the subjectivity and temporality of the state of a concept, which is the only thing an agent can aspire to have.

Usual candidates for such dynamically allocating structures are graphs/hypergraphs or informal ones like Wikipedia pages in natural language with a structure. We will describe concepts by how they work, rather than how they are stored, but they are trees of sequences of symbols that can be read as natural language with some peculiarities. First we need a few more things.

We have already mentioned that in our approach everything runs as code. To explain how, we first need to introduce the container that maps symbols into Bop statements.

Definition 4 *A **formal field** is a container with the definitions of OpCodes. OpCodes are symbols representing kernel functions and Bop statements.*

Formal fields can be dynamically created by multiple inheritance.

The definition of formal field has been simplified since the original implementations (Basaldúa 2020a). The original definition included parts related with how to search code. Now it is only defining how to run it.

Definition 5 *A **semantic space** (semspace for short) is a container of concepts.*

Semspaces map symbols to concepts and are also dynamically created by multiple inheritance. Concepts are code, but mostly declarative code. Semspaces and formal fields are similar, they have a common parent, but their function and also the knowledge they contain is different. An agent’s knowledge is stored in semspaces.

Code is text

The whole point of this framework is building software that reliably communicates and leverages knowledge written in natural language. The mapping of arbitrary code into arbitrary symbols allows for the knowledge representation to be human readable and look like natural language. We say ”look like” because it still is some formal language that is a subset of true natural language, but is understandable by humans and can be edited as language.

```
Mary went to the kitchen yesterday.
This afternoon John went to the kitchen.
Mary journeyed to the hallway this evening.
Yesterday John traveled to the garden.
  ..                                Where was John before the kitchen?
```

Figure 1: Facebook’s bAbI engine example

In our PoC with Facebook’s bAbI engine (Weston et al. 2015) –it is a text-adventure type world that generates scenarios like the example shown in Figure 1– we have successfully managed to compute correct answers based exactly on the original text mapped into code.

Of course, since the code is arbitrary human written code, it is just proof that the knowledge representation can be human friendly. Also, once we have some structure, we can learn new words and situations very easily.

How concepts work

As mentioned already, a concept is a data structure, identified by a symbol that is unique within a semspace, that contains declarative code (and possibly some imperative code that simplifies its operation). It supports both finding what we want inside it and identifying it in a **context**. A context is a semspace created for the purpose of some computation.

Concept interrogation: The chromosome analogy This analogy is based on chromosomes being the storage of genes. Each chromosome contains a thousand genes that can produce ten times that number of different proteins and the machinery finds the specific bits. We –personally– have no idea how that biomolecular marvel works, but find it inspirational for the much easier task of building code from the ”genes” stored in a concept.

Definition 6 ***Interrogation** is running declarative code as imperative. It is finding inside a concept an answer to a question.*

Interrogation is the mechanism for expressing parts of concepts as imperative code. For now, interrogation is done through pattern matching. Bear in mind that we are just defining the framework. Finding the best ways to do each part is a long term project.

Concept abstracting Concepts are made of other concepts that are made of other concepts, ... The recursion cannot possibly be infinite since each time it implies a different semspace. As we recurse, we are traveling back to ancestors of the semspace up to, possibly, the last ancestor.

Nevertheless, exploring that tree becomes gigantic soon and is fortunately rarely a good idea. What we mean by abstracting a concept is somehow replacing it by a placeholder. There is no actual ”replacement”, it is just a possible branch in a tree search that will be taken most frequently. The ”placeholder” is a terminal abstract concept that has no lower level concepts (is part of a formal field). Concepts have code that defines how they abstract in a given context.

Concept blending Concepts compose with other concepts. Not just like code does, by executing functions over the result of other functions, but to form new concepts that inherit some of their code. E.g., The word ”antifunny” should be understood even if it has not been seen before.

Understanding language requires grammatical parsing of concepts. This is called **concept blending** and is a recursive process that identifies structure based on structures of structures, a little bit like protein folding. And, again, a tree search.

4. Putting it all together with examples

Different implementations

In the present section, we detail the implementation that has been constructed as a proof of concept with the aim of gaining a deeper understanding of the best way to create concepts, fields, and semspaces.

We are putting everything together to show that what we already described is enough to tackle complex intelligence engineering architectures. We also show how learning-wise a tree search is all we need. Once a model is trained, the contextual knowledge on each decision node is enough to provide valid decisions with little computation.

It is not our intention at this point to provide a canonical experimental section accompanied by a comparative analysis with another existing implementation since existing approaches are not directly comparable.

While we have made significant progress in the implementation, several aspects require further refinement and optimization. We have already described the application to the ARC challenge and published the complete implementation in (Basaldúa 2020b).

What are the opcodes So far, we have just mentioned that opcodes are primitives (constants, functions and methods) or compositions of opcodes that we call snippets. Snippets can also be constants, functions and methods, but they are defined from primitives and other snippets. The notion of *item* in the original paper is no longer necessary. In the original implementation items could be evaluated as an intermediate result. Intermediate reward is now part of the searching algorithm, not the code.

Primitives can be anything. In the ARC implementation they are short snippets of numpy code, typically from 3 to 5 lines, that are utilized for extracting or modifying images. E.g., *pic_fork_on_v_axis_as_pics* takes an image locates a vertical axis and returns both sides of the axis as a tuple of pictures or fails. A complex deterministic algorithm like a vocoder, which converts a sound wave into a tuple of *(pitch, envelope and aperiodic components)* is just a primitive, a pure function. A model that requires training can also be a single primitive, a method. Being a method gives it access to some field to store the model's parameters. One can construct both types of algorithms, using fundamental building blocks such as a basic set of linear algebra tools. In our proof of concept using bAbI, we implemented basic operations on sets as primitives. All bAbI tasks use a vocabulary of just 141 words. With a set of 19 primitives we can solve all the questions asked by bAbI and some new questions about the scenarios generated. We also created scenarios by combining different bAbI tasks.

Composing tree search by example: Speech to text

To show how more conventional machine learning models can be implemented and benefit from lifelong learning, we start describing a very lightweight speech to text pipeline. For now, we have this and an end-to-end solution still combining some components that are not implemented as Bop code. The currently implemented vocoder is a refactoring of World (Morise, Yokomori, and Ozawa 2016) as a fixed memory C++ function. It can express complex human voice nuances including singing. This design was SOTA (State Of The Art) few years ago and is around two orders of magnitude more computationally efficient than current approaches, from around a hundred times real time on CPU to requiring a GPU to be just above real time. Combining both the efficiency and SOTA performance is a mid-term goal for our Jazz platform.

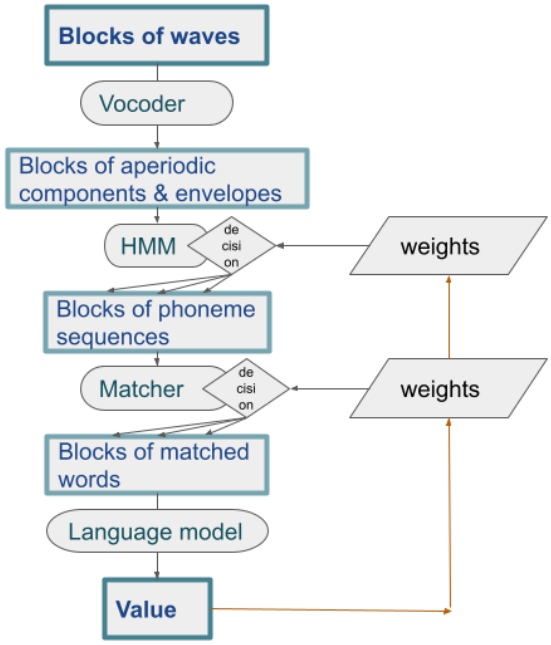

Figure 2: A speech to text pipeline
Steps labeled *decision* correspond to multiple choice (softmax) evaluations. A *run down* the tree starts at the top and branches at each possible decision until it reaches the final evaluation. A *run up* propagates the result of the evaluation back to the weights present at each decision branch. The vocoder does not have trainable parameters, the Hidden Markov Model (HMM) and the matcher do. The matcher compares with phonetic versions of words created by a phonemizer.

Note that this implementation has some advantages compared with a "classical" fit/predict model.

- By converting a classification problem into a decision problem we can approach both learning and predicting as a tree search. Furthermore, combining MCTS with strong domain specific priors we can backpropagate evaluation combined with intermediate knowledge.
- The same framework that optimizes code by trying to match concepts to an input can be used to build a pipeline that is not learned as code, since it does not make use of concepts. It is still a pipeline that learns and predicts at the same time.
- It can be used as a trained model that does not learn further by just running down the tree and picking "best so far".

- It can also be used as a model that further improves. When the likelihood of the final solution according to the language model is low, we can invest in computation trying to improve it. We can also "give up" when that computation is not further improving.
- Learning happens in a context. Through inheritance we can control control the life cycle of the trained models to fit individuals or groups.

So far, we have not used concepts. This pipeline could be stored in a concept called *"listen"*. Different sound sources could also be concepts and the whole process could be triggered by text as we see in the next example.

An example with concepts: Text as code

In this example we show how text as code works in our first implementation. We combine text generated by two different bAbI tasks *"Single Supporting Fact"* and *"Compound Coreference"*. Then, we introduce an new person, Jabari, not in bAbI generator's vocabulary and a question that could not be generated by the bAbI generator.

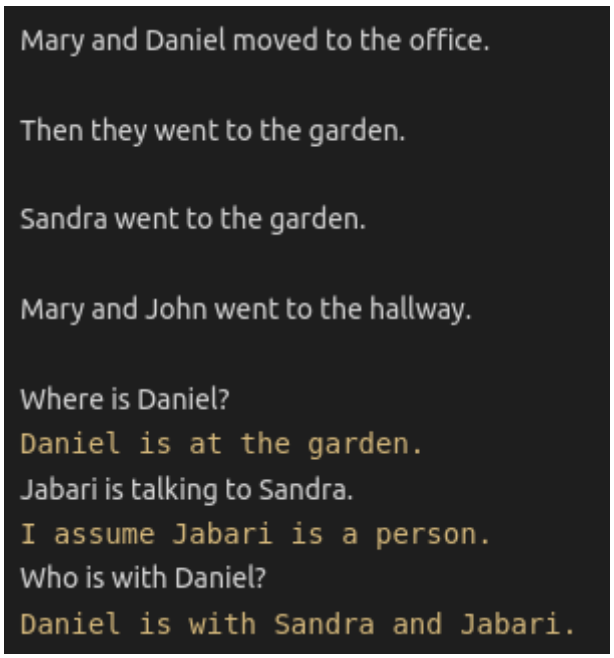

Figure 3: Short bAbI example dialog with answers
Output generated by the model is displayed in orange.

Note that generating correct answers, which can be done by gofai-like ad hoc programming, is not the most important here. What counts is having a minimalist system that grows by just doing tree search with minimal branching, one algorithm for everything. This is possible since what is creating and storing new code are the concepts.

We start with a semspace named *sys.agent* that contains the 141 words bAbI generates as symbols to concepts of which only ten in the current example are primitives: *(person, location, to move to, to be at, to be with, then, and, they, where, who)*. The rest is defined in terms of primitives. bAbI includes many synonyms. Using just 19 primitives, all dialogs generated by the 20 tasks can be answered correctly. Our initial semspace named *sys.context* inherits *sys.agent* and *sys.interaction*, a concept to store I/O.

Concept blending Concept blending is matching symbols coming from a tokenizer to symbols that are keys in a semspace. This requires finding appropriate solutions to multiplicity or non existence, including modifying how it was tokenized. We consider the findings from *link grammar* (Sleator and Temperley 1995) and its community influential to our approach. Mainly: deterministic rules about word links can be used to completely classify grammatically correct English from incorrect. We do not use link grammar for blending to make it more robust to incorrect input. Text does not always come in correctly spelled and punctuated complete sentences. Our approach starts local and composes possibly complex sentences from smaller parts. Patterns are learnable rather than written by experts. Link grammar is still a valuable source of local patterns we are planning to use.

Concepts branch different parts of their content through pattern matching. Matches evaluate to a real number rather than (true, false) to represent likelihood. Patterns like *[person/to move to/location]* are part of the concept *to move to*. The different forms like "moved to" or "went to" forward to *to move to*. This forwarding will in general carry extra information, like verb tenses, plurals, etc.

In the sentence "Jabari is talking to Sandra." we see concept blending finding a possible match between the unknown word Jabari and the concept *person*. It outputs some template message to inform that it created a new concept that abstracts as a person. That could also have been a question asking the user for clarification.

Like the rest, concept blending is a tree search with a local evaluation (the patterns) that can be trained from global evaluation (updating pattern weights from the final value).

Concept abstraction A possible approach to blending could be using POS (Part Of Speech) for the patterns. That produces syntactically correct matches, but they may be meaningless. We could also introspect the candidate matches to see if somewhere inside them we find relevant information. As the system grows, we have a lot of information. In our example, in "Mary and John went to the hallway." as blending links Mary, we already know that she is a person, but also she is currently at the garden and was at the office before that. That extra information does not help with blending. Having to search through possibly a lot of information for matching could easily become inefficient.

Abstraction is a mechanism that defines categories (aka. placeholders) used in blending that Mary fits. In this case "Mary is a person." is not just a fact, but a special kind of relation meaning "Mary matches the placeholder person.". She can also be: a proper noun, a female, a doctor or whatever at the same time. Abstraction is also used for disambiguating polysemy since concepts can have many candidate patterns that express different parts of it. This multiplicity is handled by blending via tree search.

Abstraction is not only used in blending, it provides information about Mary, since the categories are, of course, concepts.

Like everything else, abstraction can be learned and so can new patterns possibly with new categories.

Concept interrogation Each possible blending match generates code. In our example, sentences are very simple. The imperative primitives *to move to, to be with, where is, who is with* do not appear together in the same sentence. Each sentence is fully evaluated by creating a snippet that runs and is evaluated. The sentences with *to move to, to be with* silently connect both the person and the location through a declarative *is at*. The coordinating conjunction *and* builds a noun phrase. The pronoun *they* matches it solving the coreference using *sys.interaction* to see previous input. The adverb *then* is implemented to provide a sense of time in other tasks. In this task, the location would be updated also without it. The sentences with *where is, who is with* print the answer using a template checking the locations that have been previously connected to persons.

In general, the complexity of a sentence is not limited. If we had written *"Mary and Daniel moved to the office and later they went to the garden."* as one sentence, blending would start locally and match the parts. The difference is: we are parsing a bigger tree rather than two small ones. At the top level we will have two clauses linked by a coordinating conjunction. Only one of them can be imperative, that is a requirement to make it work. If both were imperative, they could be executed out of sequence. The declarative code will also run. It is like in a lazy evaluation of a function, everything that is a dependency runs in the order in which the coordination requires results, not necessarily the order in which the words are written.

The platform is efficient at handling branching. Different alternatives generating different code and results have a part that is common and a part that is different. Re-using results avoiding having to recompute them can be done through caching. The implementation of the class *core* simplifies it.

5. A pragmatic approach to engineering intelligence

Definition 7 ***Intelligence*** *is conversion into form. Form (as in formal) is unambiguous definition. In our case, just another word for code.*

Defining intelligence sounds like invading many other academic fields. The reason why we do it is because converting non formal (data and natural language) into formal (model and code) is the key ingredient of AI. We could invent a new word for it like "formalizing". But, it happens to be exactly:

- What IQ tests measure: The capacity of finding a model that explains some given examples and applying it to new ones.
- What is meant by: "Intelligence is building models."
- by: "Intelligence is finding patterns."
- or by: "Intelligence is problem solving.".

We already have a word for that, **intelligence**, we don't need a new one.

That has been said long time ago by McCarthy: *"The intelligence, if any, of the advice taker will not be embodied in the immediate deduction routine. This intelligence will be embodied in the procedures which choose the lists of premises to which the immediate deduction routine is to be applied."* (McCarthy 1960)

We can pretend defining intelligence is "one hundred Nobel prizes away" or we can just do it so that we can engineer it.

It is important to exactly point out **where** it is: In the step of building code (or fitting models) to the data to obtain a formal model which we can run on new data and evaluate.

The next definition, also possibly controversial, was already argued in the original formal fields paper:

Definition 8 ***Understanding*** *is being able to successfully run and evaluate code in order to optimize it.*

Basically, the only thing computers natively understand is function optimization. When we convert chess into function optimization, they "understand" chess as a result of optimizing the function that plays chess. And they provide evidence of understanding by the actions they take, which is enough for an engineer, maybe not for a philosopher.

It makes sense to have a specific definition for computers that is different than for humans (although some say: "This doesn't run." as a synonym of "I don't understand it."). Agents need to use a precisely defined "I don't understand." in dialogs with humans. The question may need clarification, maybe it is just misspelled, etc. Originally, this was called "understanding within a field". The field part is just one way of doing it. It is not really necessary and makes it less clear.

Armed with these definitions, we can describe what agents are doing:

Agents are searching candidate knowledge representations of data to fit some function (see section 5) and **from those they understand** (i.e. those which run and can be evaluated) they optimize and find out the result of the best representation. All this subject to resource limitations. The results are possibly used in the context of another computation.

6. Agency and learning

Agency

What we build, rather than models that have **fit/predict** modes are **lifelong** agents. Lifelong has nothing to do with living computation, self reproduction or anything like that, it is just about running for unlimited time without exhausting its resource allocation. It uses resources to do computation and gets more as a reward for useful output. The agent is persisted, built from scratch or the good parts of other agents, can be stopped and edited/fixed/updated.

It has:

- An I/O pipeline to communicate **data**, **targets** and **output**. A target can be a constant that represents some objective like the "observed Y" in supervised learning or the "correct solution to an example" in the ARC (Chollet

2019) challenge. It is code, so it can be a constant or a function.

- An event callback that **provides computation** as a result of data coming in or just time passing
- Knowledge representations: **Semspaces**
- A computational **budget**, a **cost function** (a proxy that estimates the computational cost in advance which is used for practical reasons instead of the actual cost) and a **reward** for its output in the same units that can possibly be negative.
- A **metric** that is a proxy function of the expected reward and has no cost to evaluate on intermediate results.

Learning

Now that we have precisely defined what part intelligence is, and that our agents are lifelong, we see that learning does not play the same role as in models that are fit/predict.

In AI learning is –and this in not controversial, unlike how the word is often used for humans– not acquiring more knowledge, but improving at a task. Acquiring more knowledge is so trivial for software it doesn't even need a name (try inserting rows in a database).

We want to highlight that once an agent is proficient at a set of tasks, learning becomes less important. The most useful models in production –say Google translate– do not learn at all and are occasionally replaced by better models learned offline. In this case, an agent that applies intelligence converting questions made by humans over a domain, say geography, is converting them into form (using intelligence to interpret the question in natural language, clarifying it if necessary) and answering them. It is useful without learning (becoming more efficient at the same or a new task). It can also update its geographical knowledge without that being learning in the computer sense.

Differentiable learning vs. lifelong learning End-to-end differentiable learning (aka. deep learning):

- Is the most efficient way known to learn huge datasets, especially from scratch.
- Is the most efficient way known to fit gigantic models.
- Is best at most problems, especially those problems in which a structure in the form of a lower dimensional manifold exists.

Why not use DL all the way? The answer is, that comes at a price of obscurity and inefficiency that we do not want to pay. We want to build agents that are faster, use less resources and can be deployed with little or no deep learning.

Experience There are many applications in which an agent is not expected to rediscover Quantum Field Theory from raw data of a particle accelerator. Just being honest about its own limitations and helpful when it can be, without exploring infinite trees is enough. Many decisions in walking the tree can just be "in this situation, that was useful". Occasionally, explore new paths but mostly just follow successful ones. We call data about situations and success rates **experience**. As in Monte-Carlo Tree Search, the algorithm is "always-ready", meaning it has an answer, possibly not very good, and it can spend resources improving it. Experience is a principle rather than an algorithm: Having a shallow always-ready answer for most tree search decisions based of previous success rates in some context.

7. Present and future goals

Our main goal for this architecture, the reason we created The Tangle and what we use to battle-test our implementations is making sense of human-written text. The next goal is becoming a reliable natural language interface for software services.

Dreaming about near future applications to advance towards deeper human interaction, we could give some definitions.

Definition 9 ***Self awareness*** *is having a concept of self in one's semspace. That is as natural for an agent as playing chess against itself is for a chess engine. It will be just as good/bad, smart/stupid about itself as about anything else.*

Definition 10 ***Introspection*** *is being able to make sense of one's code. And in this case, it is superhuman just out of the box by design.*

Definition 11 ***A model of mind*** *is having good concepts of other agents and humans. Again, like with self awareness it will be as good/bad as anything else.*

Definition 12 ***Self experience*** *is having access to one's past history of interactions (experience as in record, not as in perceptual experience). Again, superhuman out of the box.*

None of that is hard to build with what we already have. We are not aware of any other architecture in which this is as simple. Some authors expect it to "just emerge." We don't want to make an argument about that, but we do want to highlight that engineering it is not the same as hand-coding it. Engineering it makes it reliable, understandable, and editable, but it is still the agent's task to learn and evolve it, each agent in its own subjective way.

8. Conclusion

We have presented an architecture built on top of ideas that have been researched by our team in recent years. It operates as a tree search of tree searches (which is still a tree search) to tackle ambitious present and near future challenges.

The architecture is open to different search algorithms. It is work in progress to find out which one is best, but we have already shared experimental results with algorithm details and code in the ARC challenge (Basaldúa 2020b), The Tangle and TLSS.

We also discuss the unique position of our approach to tackle reliable human interaction in natural language.

9. Acknowledgments

We wish to thank BBVA for releasing Jazz version 0.1.7 as open source software in 2017. Jazz is a highly efficient data processing platform that is currently being refactored as a server implementing formal fields in both research and industrialized applications.

We wish to thank BBVA AI Factory for supporting our research including The Tangle and this paper.

References

Basaldúa, J. 2020a. Formal Fields: A Framework to Automate Code Generation Across Domains. *arXiv preprint arXiv:2007.14075*.

Basaldúa, J. 2020b. JazzARC, PoC on Code Generation using Formal Fields on the ARC Challenge. https://github.com/kaalam/JazzARC.

Chollet, F. 2019. The Abstraction and Reasoning Corpus (ARC). https://github.com/fchollet/ARC.

McCarthy, J. 1960. *Programs with common sense*. RLE and MIT computation center.

Morise, M.; Yokomori, F.; and Ozawa, K. 2016. WORLD: a vocoder-based high-quality speech synthesis system for real-time applications. *IEICE TRANSACTIONS on Information and Systems*, 99(7): 1877–1884.

Schmidhuber, J. 1990. *Making the world differentiable: on using self supervised fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments*, volume 126. Inst. für Informatik.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Sleator, D. D.; and Temperley, D. 1995. Parsing English with a link grammar. *arXiv preprint cmp-lg/9508004*.

Von Neumann, J. 1958. *The computer and the brain*. Yale university press.

Weston, J.; Bordes, A.; Chopra, S.; Rush, A. M.; Van Merriënboer, B.; Joulin, A.; and Mikolov, T. 2015. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*.

Winograd, T. 1980. What does it mean to understand language? *Cognitive science*, 4(3): 209–241.