A learning system for mathematical intelligence

Jiayi Wei

October 29, 2017

> "Mathematics is a game played according to certain simple rules with meaningless marks on paper."
>
> – David Hilbert

What is the essence of mathematical intelligence? Is it possible to construct an artificial intelligence agent that can *learn* how to do maths? Assuming this agent works within some logical calculus and starts with only minimal amount of mathematical knowledge, and like how a teacher teaches a student, we give it increasingly harder mathematical problems to solve and provide solutions when necessary. Could it gradually acquire more knowledge and become better at solving new problems by imitating the solutions we provided and learning from its own problem-solving experience?

A concrete example

To get some sense of what kind of challenges we could be facing when building such a learning system, let's consider some concrete scenarios. Suppose for now that we are teaching a human student instead of an AI, and we want to teach him how to write formal proofs in an interactive proof assistant like Isabelle. First, we show him the definition of an Integer List (iList): A list is either an empty list (called "Nil") or an integer followed by another list. Then, we also introduce the concat operation (which prepends one list to another) by giving him a recursive function definition. Below is the corresponding code in Isabelle.

```
datatype iList = Nil | Cons int iList
fun concat :: "iList ⇒ iList ⇒ iList" where
  "concat Nil ys = ys" |
  "concat (Cons x xs) ys = Cons x (concat xs ys)"
```

Now, we give the student this problem: prove that concatenating any list with Nil is still the same list, i.e. prove this proposition:

```
lemma concat_nil: "concat xs Nil = xs"
```

Of course, we shouldn't expect a student with zero experience of this proof assistant to solve this problem. In Isabelle (and in many other proof assistants), we can rewrite proof goals by giving different instructions to the system, and these instructions are often called "tactics". Suppose so far, this student has seen the usage of tactic "simp" and tactic "induction _", so naturally, he would tackle this problem by first trying these two tactics.

Basically, tactic simp is used to simplify the current proof goal by rewriting it using existing definitions and proved lemmas. Since neither "concat xs Nil" nor "xs" can be simplified, directly applying this tactic wouldn't make any progress. Now, the only option left is to apply proof by induction. Because tactic "induction _" requires an induction variable, and here we only have one variable xs in our goal, the student would try "induction xs", and this would rewrite the goal into the following two subgoals:

```
1. concat Nil Nil = Nil
2. ∀x xs. concat xs Nil = xs ==> concat (Cons x xs) Nil = Cons x xs
```

The first goal can be solved directly by using the first case in concat's definition, so a simp tactic would handle this. The second goal is not hard, too; we just need to rewrite the induction conclusion using the induction hypothesis, and then apply concat's definition. Thus, this goal can also be handled by a simp. Actually, to prove this lemma in Isabelle, you just need to write down the following code:

```
apply(induction xs) apply(simp) apply(simp) done
```

An AI student

Now let's think about how could an "AI student" deal with this problem? Traditionally, we have developed many kinds of proof searching techniques that work by enumerating all possible combinations of a given set of tactics. But here we are talking about a more intelligent system that can actually improve its ability of using new tactics and writing proofs.

So how does the human student know the existence of those two tactics? Well, maybe he saw his teacher use them to solve other similar problems, or maybe he was simply told that he should use those two tactics. The point is, if our system is going to be adaptive to new mathematical domains, it should be able to have an extendable knowledge base. Also, this knowledge base should not only store a set of exact components like tactics, definitions, and lemmas, but also associate "experience" with them, which usually means we should also find some statistical/probabilistic representation to store implicit knowledges created by solving past problems.

In order to solve similar new problems using similar solutions, the agent should utilize this statistical representations to correlate new problems with existing components and solutions. For instance, if the system just solved three problems about lists using induction and it's facing a new list problem, it should probably also try induction. As an example on this kind of correlation learning, there was a recent work on program synthesis which trained networks to predict the kinds of functions that might be useful when trying to recreate the outputs for a given set of inputs (see DeepCoder).

Figure 2: Neural network predicts the probability of each function appearing in the source code.

In oder to turn unfamiliar problems into solvable subproblems (i.e., learn by analogy), the agent should also utilize the behavioral connections between components encoded in this statistical representation. For example, after seen enough problems about sin and cos functions, the agent should realize there is some duality between these two functions. So after learning how to solve one trigonometric problem, it should also be able to solve the dual version of that problem. This kind of behavioral connections are common in NLP models (see for example Mikolov et al., 2013):

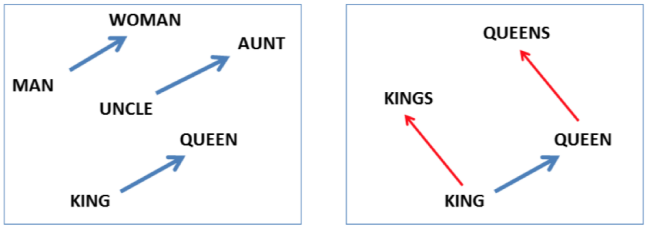


Figure 2: Left panel shows vector offsets for three word pairs illustrating the gender relation. Right panel shows a different projection, and the singular/plural relation for two words. In high-dimensional space, multiple relations can be embedded for a single word.

A harder problem

Since the concrete example we just saw is too simple, let's consider more complex scenarios and see what new challenges would arise. So back to the teaching scenario, suppose now that we also introduce the definition of the "rev" function (which reverses the appearance oder of each element in a list):

```
fun rev :: "iList ⇒ iList" where
  "rev Nil = Nil"|
  "rev (Cons x xs) = concat (rev xs) (Cons x Nil)"
```

And we want the student to prove this lemma about rev and concat:

```
lemma rev_concat:"rev (concat xs ys) = concat (rev ys) (rev xs)"
```

As before, directly applying simp would not make any progress. Now the question is, since there are two variables (xs and ys), which one to induct on? In this simple case, the student can simply try both of them and see which one works. But in general, induction proof generation is a very challenging problem and can introduce infinite branch points into the proof search space (see Bandy et al., 1999). But as mentioned above, if we can associate experience with each tactics, the agent may learn from experience and use each tactics in ways that lead to higher success rate.

Now assume the student has applied "induction xs", and he needs to solve the following 2 subgoals:

```
1. rev (concat Nil ys) = concat (rev ys) (rev Nil)
2. ∀x xs. rev (concat xs ys) = concat (rev ys) (rev xs) ==>
   rev (concat (Cons x xs) ys) = concat (rev ys) (rev (Cons x xs))
```

For the first goal, a simp will rewrite it into "rev ys = concat (rev ys) Nil", and this is exactly the lemma "concat_nil", which has just been proved. So adding this lemma into simp's simplification rules will solve this goal, and this can be done in Isabelle via:

```
apply(simp add: concat_nil)
```

For the second goal, apply simp will leave us with this new goal:

```
concat (concat (rev ys) (rev xs)) (Cons x Nil) =
concat (rev ys) (concat (rev xs) (Cons x Nil))
```

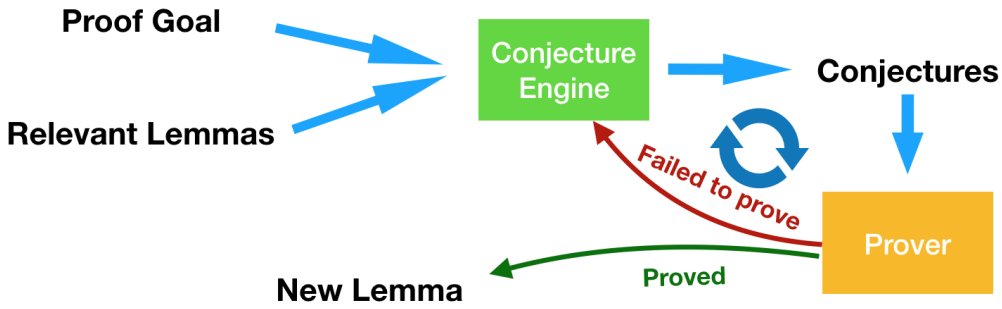
Directly prove this goal is hard and induction on either xs or ys wouldn't work because the resulting induction hypotheses are too weak for proving the induction conclusions. Instead, if we first prove the associativity of concat as a lemma, this goal becomes just a special case of that lemma and can hence be solved via a simp. It turns out that this more general lemma is also easier to prove (just induction followed by two simp):

```
lemma concat_assoc: "concat (concat xs ys) zs = concat xs (concat ys zs)"
```

With this lemma proved, the student can finish the previous proof by "apply(simp add: concat_assoc)".

Although this is a simple example, it demonstrates an important fact which can make writing proofs challenging: Proving some facts may require finding more general facts and proving them first. And this is often a challenging task. To find those more general facts, we need a conjecture generation engine. Moreover, this conjecture generation engine should take a goal or at least some relevance metric as input, since there are potentially infinite number of facts and we are only interested in those that are relevant. Also, as we usually make conjectures based on our past observations and experiences (inductive inference), the conjecture should also take some relevant facts as input. Concrete motivating examples are usually very helpful because they can act as efficient constraints over the candidate space and prune search branches dramatically. For example, say we want to prove "a+5=5+a" and have already proved "1+a=a+1", a reasonable conjecture would be "a+b=b+a".

So in our proof automation workflow, we can have a conjecture engine take both the proof goal and some relevant special case lemmas as input and output a sequence of conjectures, and then try to prove (or disprove) those that are useful for solving the proof goal. This workflow is demonstrated in the following graph:



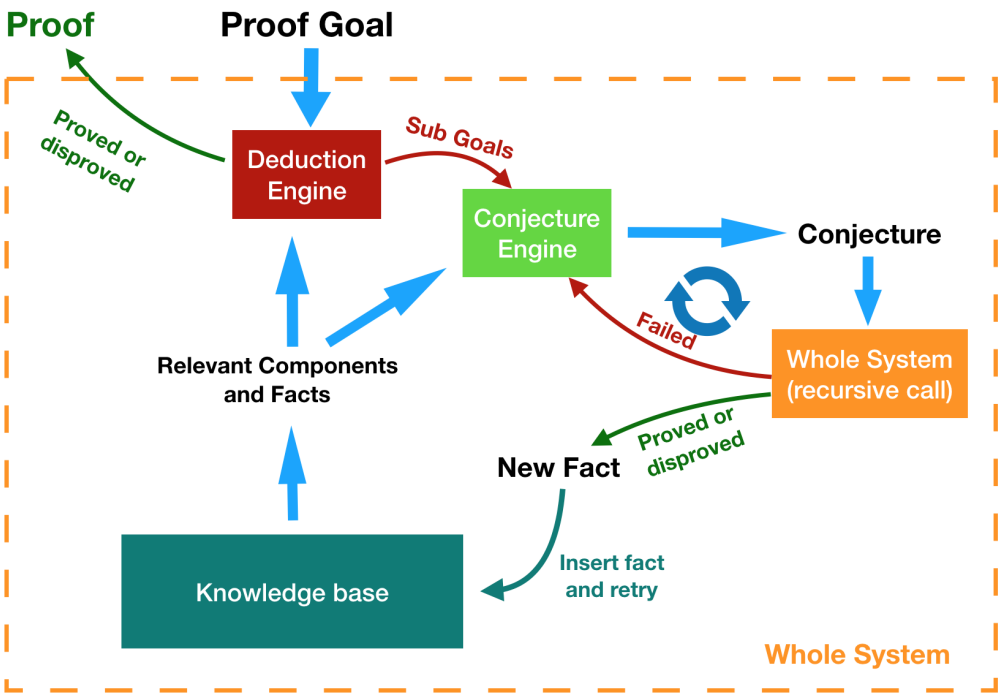
One possible way to implement this conjecture engine is by using an extendable set of conjecture construction tactics. For instance, in the previous example, to construct the associativity conjecture, we can use an expression relaxation tactic which replaces complex expressions with simpler ones (replace "Cons x Nil" with "zs" in the associativity example) . And after introducing those tactics to the agent, its only task is just learning "when and how to use which tactics".

Learning cycles

We can generalize the conjecture generation workflow and build a system that can learn from problem solving. As mentioned above, this system would have an extendable knowledge base that stores new definitions, lemmas, and tactics. And the knowledge base also maintains some statistical knowledge representation to associate past experience with those components. When we need to introduce new components, we just directly insert them into this knowledge base so that our system would know the existence of those components and can start to accumulate experience of using them.

The learn-by-doing process consists of a sequence of problem solving tasks. In each task, a teacher gives the system a proof goal which may be true or not, and the system then tries to prove or disprove this goal and will eventually return a proof (if succeeded) or a failure (if timed out). When a failure is returned, the teacher can optionally choose to provide an example solution, and the system would then

try to solve this problem again by mimicking the given solution. After solving each task, the system not only stores those newly proved facts into the knowledge base, but also optimizes corresponding implicit knowledge representations to make its deduction and conjecture engines more efficient. The graph below demonstrates how the system solves a given task.



The road map (unfinished)

To concretize the such a system, the following questions may need to be answered:

- How to design the knowledge representation?
  - For explicit knowledges, how to recognize important facts and discard/forget unimportant ones?
  - For implicit knowledges about logical formulas, which features to use?
- How to search for solutions?
  - Monte Carlo Search guided by implicit knowledges?
- How to design an improvable conjecture generation engine?
- How to learn from experience?
  - How to design an efficient reinforcement learning algorithm?

connection with programming languages and formal methods

A general mathematical intelligence system can be very helpful for those PL problems like proof synthesis, program synthesis, and program analysis etc. Many of the current PL solutions to these problems often don't use persistent memory, and as a result, they can't learn from the past and are thus unable to scale to harder problems.

connection with machine learning

In many machine learning applications, there are usually a fixed set of objects/rules and new components cannot be added. Having the ability to reason about an extendable set of mathematical objects and discover new logical facts, our new learning system can be more flexible and universally usable, and hence has the potential to handle problems in domains which are beyond the reach of current machine learning approaches.

Limitations

In our proposed system, all learning is essentially about decision making, i.e. when (based on features of the goals) to use which component (i.e. definitions, lemmas, proof/conjecture tactics, etc.) in what way (e.g. which tactic arguments to use). But all those definitions are introduced manually by us. When facing a challenging problem, a mathematician can decompose the task by creating new definitions/notions and using them as reusable reasoning procedures. For example, complex numbers were initially introduced to study roots of cubic polynomials (see history). Hence, a powerful extension to this system could be adding the ability to automatically create and evaluate new definitions. And Developing such an ability might bring us closer to the very essence of mathematical intelligence and creativity.

> "The true sign of intelligence is not knowledge but imagination."
>
> – Albert Einstein