Approval-directed algorithm learning

Paul Christiano Follow
Feb 21, 2016 · 8 min read

Suppose that I can train an AI system to handle "small" problems, and would like to build an AI that handles "big" problems.

One approach is to train an AI to implement a *process* for handling the big problem. Effectively, we break the big problem into small problems of the form "what should I do next?"

This has been the focus of some recent work in deep learning. In that context, the idea is to supply some external memory or computational aids, and to train a neural network to use those resources. The network learns a map (current controller state, observation) → (new controller state, action), where the action may read or write to an external memory or use a computational aid. (This framework includes attentional mechanisms, which are also getting a lot of attention.)

These systems are usually trained end-to-end: we have some intended functionality, we let the controller run, and we reward the controller whenever it successfully implements the intended functionality. If there is supervision, it is given by comparing the learned behavior to some algorithm that is known to achieve the task. (For example, rewarding the controller if it looks at the same part of the input that the intended algorithm does.)

I am interested in algorithm learning when we don't have a way to verify results, nor a preexisting implementation. I think that this setting is most relevant for AI control, and is also likely to be an important case for practical applications in the long-term.

Approval-directed computing

In order to train such a controller, we need a way to provide feedback on a transition from (current state, observation) → (next state, action).

The end-to-end approach is to actually execute the proposed transition, use the current version of the controller to continue execution from the resulting state, and use empirical or predicted value estimates as a training signal.

I'm interested in an alternative approach based on approval-maximization: a human reviewer assigns a score to a (state, observation) → (state, action) transition, and the system is trained to greedily maximize its score.

We score a decision without using any information about the results of that decision. Of course the reviewer can think about whether a decision will lead to the problem being solved correctly, and can use that information to help assign a score.

Desiderata

There are two related problems with applying approval-directed training to algorithm learning:

- The human reviewer can't make heads or tails of the internal state of the agent, and can't understand the consequences of any proposed actions.
- The actual quality of an action, such as writing to a memory location, depends critically on how the controller will behave in the future, and on the controller's implicit interpretation of program state.

For example, consider a human reviewer evaluating the action "Write the current input to memory location 1724." What are they supposed to make of this instruction? How are they supposed to evaluate it?

Ideally, we would like to ensure:

- the program state is comprehensible to a human, and
- the semantics of the program state are fixed, rather than depending on the rest of the controller's policy.

Of course we will eventually want to build rich and complex program states, so these fixed semantics need to be powerful enough that we can use them to build up more complex abstractions.

Natural language, with its usual (imprecise) semantics, is an extremely convenient representation language for this purpose, and so we will seek a computational model whose states are defined in terms of natural language.

Annotated functional programming

Terms

Our computation is built out of *terms*: a term is a short natural language expression (the *template*) with *n* “slots,” together with *n* pointers to other terms (the *arguments*). For example, we might have a term with template “the pair with first element _ and second element _.”

We represent terms by enclosing them in braces, and plugging the arguments into the template. For example, we would represent the term from the last paragraph as {the pair with first element {*x*} and second element {*y*}}, where *x* and *y* are the arguments.

The semantics of terms are straightforward; you’ve probably guessed them already. The term from the last paragraph is analogous to the pair (*x, y*) in Python.

Views

A single term, together with all of its arguments, and all of *their* arguments, and so on, may be very large. A small controller will not be able to directly manipulate very large terms.

Instead, our controller will take as input the *view* of a set of terms.

The view of a set of terms consists of the template of each term, with the slots filled by special tokens {1}, {2}, … Two slots in the view are filled with the same token if and only if the corresponding slots in the original terms held the same pointer.

For example, the view of

- {the tallest building in {San Jose}}
- {the next flight from {San Jose} to {the oldest airport in {New York}}}

is

- {the tallest building in {1}}
- {the next flight from {1} to {2}}

Constructors

Similarly, rather than having our controller output terms, our controller will output term constructors.

We can represent terms as rooted trees. To build the tree for *t*, we label the root with the template of *t*, and have one subtree for each argument of *t* (recursively constructed in the same way). The leafs are terms without arguments.

A term constructor is a similar tree. However, some leafs can be labelled with the special tokens {1}, {2}, … rather than with terms.

Given a state and a view of that state, we can instantiate a term constructor to obtain a term. To do so, we replace each leaf labelled with {*n*} by the corresponding pointer from the state.

In the example from the last section, the instantiation of

- {the distance between {ten miles North of {1}} and {2}}

is

- {the distance between {ten miles North of {San Jose}} and {the oldest airport in {New York}}.

Executing a program

The state of a program is given by:

- A single term representing the question-to-be-answered, such as {how many people live within {*x*} miles of {*y*}?}
- N registers, each of which holds a (query, response) pair. The query is a term constructor and the response is a term. The possible queries will be described later in this section.

N is pretty small, think 5–7.

To answer question Q, we prepare an initial state with Q as the question-to-be-answered and ({no query},{no response}) in each register. We then iteratively apply the controller to update this state, until it eventually produces a return value.

At each step, the controller receives the view of the state as input. The view of a state is simply the view of all of the terms and term constructors that define the state.

For example, the view of a state may look like:

- Question: {what is the length of {1}?}
- Register 1: {what is {1}?}, {the list with first element {2} and remaining elements in the list {3}}
- Register 2: {what is the length of {3}?}, {the length of {3} is {4}}
- Register 3: {no query}, {no response}
- Register 4: {what is {the number one} plus {4}?}, {the sum is {5}}
- Register 5: {no query}, {no response}

The controller then outputs an action for each register, which will change the query/response stored in that register.

The possible actions are as follows, together with the effect of executing that action in register *k*:

- **No-op**. Leave register *k* as is.
- **Clear**. The query in register *k* becomes {no query} and the response becomes {no response}.
- **Examine {*n*}**. {*n*} should be a token that occurs in the input view. The query in register *k* becomes {what is {*n*}?}. The value in register *k* becomes whatever was pointed to by {*n*}.
- **Return *a*.** We instantiate *a* to obtain a term A, which should be appropriate as an answer, i.e. as an assertion about or related to the question. The computation stops, and A is the return value. (This instruction should not be simultaneously executed in two registers, we can pick one randomly if there is a conflict.)
- **Query *q*.** We instantiate *q* to obtain a term Q, which should be appropriate as a question. The query in register *k* becomes Q. We then start a new recursive computation, answering question Q, and run the recursive computation until it returns a term A. The value in register *k* becomes A.

For example, in the state described above the controller might execute “Return {the length of {1} is {5}}.” Register 1 was produced by executing “View {1}.” And Register 2 was produced by executing “Query {what is the length of {3}?}”

Discussion

It is easy to see that this computational model is universal; we can directly translate programs written in a traditional functional language by replacing each function call with a **Query** and using the controller to implement the (finitely many cases of finitely many) pattern matches.

Comprehensibility

The chief novelty of this framework is that putting together correct-and-useful-looking steps will lead to a correct and efficiently computed answer. That is, the system eventually reaches a state where the given (query, response) pairs uniquely pin down the answer to the question-to-be-answered, under the operator's interpretation of those queries and responses. If all subqueries were answered correctly (under the operator's interpretation), and assuming the operator doesn't make any errors, then this answer will be correct (under the operator's interpretation).

For other computational models (e.g. a RAM machine), the notion of "correctness" is itself dependent on the controller's policy, and so there is no action-by-action way to train a controller to be correct.

This locality property is necessary if we want to apply approval-directed training. It may also make learning much easier.

How hard is learning

There is good news and bad news.

The good news is that this framework can concisely express very complex computations, in a framework that is very close to high-level functional languages. Other good news is that approval-directed training is much easier than reinforcement learning—if you have the data—since the learner doesn't have to deal with long-range dependencies.

The bad news is that each of the controller's steps is a map from 5–7 natural language sentences to 5–7 natural language sentences. This is a small enough domain that we can at least imagine training a controller with foreseeable techniques. But it is a hard problem, and the controller would need to learn a very complex policy (perhaps itself involving e.g. attentional mechanisms).

I don't think that the bad news is a show stopper, but it does restrict the applicability of this scheme.

Comparison with existing approaches

Approval-directed algorithm learning would not fit into the existing literature on algorithm learning (at least the recent work in deep learning). The fundamental goal of existing work is learning without any internal supervision, and in practice the focus has mostly been on simple algorithms that would be trivial for a human to specify directly.

Approval-directed algorithm learning is primarily interesting in domains where having humans implement *any* algorithm would be challenging and time-consuming. In these domains, we may be willing to spend a lot of human time, as long as it's less time than we'd need to actually write the algorithm. Of course, actually realizing these savings requires the controller to generalize well across different tasks or parts of tasks.

Human input

Requiring a bunch of human input may look really bad, compared to reinforcement learning systems that require no or minimal input.

The optimistic vision is for most of this supervision to itself be provided by learning algorithms. I'll talk about this much more in an upcoming post.