# Automated assistants

Paul Christiano Follow
Jan 22, 2015 · 13 min read

In my last post, I described a possible scheme for using a powerful predictor to build a smart machine that does what you want it to. In that post, I assumed that the predictor was **very** powerful. In this post, I'll talk about "scaling down" these results to a more realistic predictor.

## The starting proposal

Here is the proposal from my last post, with a more pragmatic—and less detailed—description. As before, we assume that we have access to a predictor P.

We'll describe an assistant that accepts spoken commands (e.g. "What is the population of Brazil?", "Let Alice know that I'll be late" or "Make my blog look nice.").

When you give the assistant a command Q, you may be asked to provide training data. During normal operation, this happens with small probability (say 1%) for each command. To gather training data:

1. You are given a random command Q*. Half of the time, Q* = Q. The other half of the time, Q* is P's guess for a random command that you will either execute or output in step 2.
2. You decide how you would want the assistant to execute Q*, and enter a sequence of commands F(Q*) = [Q1, Q2, ..., Qk]. In order to determine what sequence of commands is appropriate, you might have your assistant execute some additional commands Q'. (But you definitely won't be asked to generate more training data.)

In order to execute the command Q, the assistant does the following:

1. If Q is a primitive command, such as "Display X" or "Send message M," then it is performed.

2. Otherwise, the P guesses what the value F(Q) = [Q1, Q2, …, Qk] would have been, if you had been asked to enter training data and Q* = Q had been chosen. It then executes each instruction Q1, Q2, …, Qk.

Note that the same training data, and hence the same function F, is used for Q itself, for each of Q's steps Q1, Q2, …, Qk, and for any other commands Q' that you invoke while you figure out how to implement Q*. (This is why we randomized Q*—so that it covers each of these cases with some probability. This randomization is actually doing a lot of work.)

My last post argued that—at least if P is a very good predictor—this is an attractive candidate for a robustly beneficial AI.

## Example

Suppose I gave the command "Which direction is home?" I'd like the assistant to display an arrow pointing in the direction of my home and to tell me which compass direction it is.

With probability 1%, I am first asked to produce training data. In this case, the system speaks a random command for me to demonstrate.

If it happens to be "Which direction is home?" (this happens half of the time) then I say, "Display a map that includes my current location and my home." The assistant brings up a map. (Rather, it does whatever I would do if I had instead been given the command "Display a map that includes my current location and my home." Hopefully I would correctly display the map!) I determine that my home is two miles Northwest. I then enter the sequence ["Display an arrow pointing Northwest," "Say 'Northwest' "].

In order to execute the query, P predicts how I would implement "Which direction is home?" If all goes well, then P predicts that I would output ["Display an arrow pointing Northwest," "Say 'Northwest' "]. It then uses the same procedure to execute these two commands, i.e. it predicts how I would implement each of them and does that.

This is a pretty lightweight example, but it is the same as executing a more complicated task such as "Arrange a trip to Boston." With a more creative user, it can even apply to commands they couldn't execute on their own, like "How could I most efficiently learn chemistry?"

## The problem

This proposal involves the predictor learning a *very* complicated mapping.

In the extremely simple example above, even if I just ask it "Which direction is home?" over and over again, it must learn the mapping from (States of the world → which direction to my house). In order to answer this question, it needs to consult its compass and GPS and combine the results in an intelligent way.

As the domain becomes more complicated, the demands on P become even more extreme. Imagine a predictor which can learn to predict the design of a machine based on a description of its function. If it worked very generally, such a predictor would need to be a resourceful agent that can use its computational resources strategically and that can carry out complicated cognitive processes. This isn't what we would normally call a "predictor," and it already raise concerns with robustness and control.

This post asks: can we build a similarly safe system using a more realistic predictor?

# Scaling down

In this section, I'll describe two techniques for using weaker predictors:

1. Learning processes rather than results
2. Going meta

In fact [2] is a generalization of [1], but [1] is such an important special case that it seems worth discussing anyway.

## Learning processes rather than results

We can think of your behavior as a loop:

1. You perceive a situation.
2. You react to that situation.
3. Your action changes the situation.

For example, if you are asked "Will it rain tomorrow?" you might respond by consulting the GPS to find your location. Once you see the

latitude and longitude, you may send an appropriate query to weather.com. Once you get the result, you may extract the probability of rain and display it on the screen.

Each of these actions changes your situation, and once you perceive the new situation you can quickly find the next action.

Rather than teaching a machine the map **Command → Implementation**, we could try to teach it the map **Context → Response**. We could then generate complex behaviors by iterating this map.

Here is a simple implementation of this idea. A *context* consists of the command which is currently being executed, as well as everything the user has input or observed while implementing the command.

A *response* consists of a command to execute in a given context, or a command to supply as the next element of the output sequence.

Instead of training P to directly execute a command, we train it to guess our response to each context. We gather training data exactly as in the original proposal, but rather than collecting a single data point of (Command→Implementation) we get a series of data points of (Context→Response), one for each command you execute while implementing Q*.

We can then implement the main loop described above:

1. You perceive a situation // P is given a context.
2. You react to that situation // P guesses your response to that context.
3. Your actions change the situation // P's response is executed by the assistant. The response itself, as well as any outputs produced by the assistant while answering, are appended to the context.

This process is then iterated until P produces an output response.

For example, to execute "Will it rain tomorrow?", P learns to respond by querying the GPS. The assistant executes this instruction by displaying the coordinates, which are then added to the context. P responds to this new context by querying weather.com for those coordinates. This again results in an output, which modifies the context. Finally, P responds to the new context by outputting "Display

25%” or whatever the answer may be. Each of these is a very simple mapping, which can be learned using existing technology.

In this setup, P learns a policy to control a more complicated computational system. It is similar to training a neural Turing machine by trying to teach the controller by example. The difference is that P’s instructions are implemented recursively by the assistant, rather than being restricted to primitive operations.

Teaching by example can potentially lead to fast learning (and facilitates this recursive breakdown), but collecting the training data is expensive. In many contexts, it is much more efficient to generate training data automatically by experimenting with policies until you find one that achieves a given goal. For example, it is much cheaper to learn to play backgammon by playing millions of games against the computer and learning which work well, than by hiring human experts to produce training data. This is a significant problem, which can be addressed using the ideas from the next section.

Learning process rather than results also forces the assistant to use the same computational process as the human. If the predictor is weaker than a human this is fine, but if the predictor is more powerful then we will be handicapping it. This is another significant problem, which will also be addressed in the next section.

If P is a weak predictor, then we can simplify its problem by “exposing” some of our intermediate cognitive steps. For example, we might normally answer “Is 3 x 8 > 5 x 5?” by carrying out the multiplication in our head. We could instead spell out our reasoning, by asking the assistant “What is 3 x 8?”, “What is 5 x 5?”, and “Is 24 > 25?”. This makes the predictor’s job much easier: the map from “Is 3 x 8 > 5 x 5?” → “What is 3 x 8?” is much more straightforward than the map “Is 3 x 8 > 5 x 5?” → No.

## Going meta

At the core of our assistant is the predictor P. In practice P might be a neural network trained to reproduce the user’s judgments, or some other relatively simple model. The original proposal is asking P to solve a problem which is *much* too challenging. There are two problems:

1. If we try to each P processes rather than results, then its job gets easier. But it’s still probably too hard. For example, we’d like P to make really efficient use of observations of the user, to anticipate

changes in behavior, and to carry out whatever complex processing the user performs in their head.

2. Even if P is up to this task, then it's just going to be following the user's lead, which is unsatisfying. We would like P to use its capabilities to their full extent, short-cutting any computations it can and searching for more efficient alternatives.

We can address both of these problems by *going meta*.

The original assistant executes Q by asking P to predict the user's implementation [Q1, Q2, …, Qk] and then executing each step. Instead, we could execute Q by asking *the assistant* to predict the user's implementation, i.e. by executing the command "Predict how the user would implement Q." This command is executed by P predicting how the user would predict how the user would implement Q. I'll call this process metaprediction.

In fact we could execute "Predict how the user would implement Q" by going more meta, and asking the assistant to predict how the user would implement "Predict how the user would implement Q." Obviously this needs to eventually bottom out with a call to P. For now I'll set aside the question of how to choose when to go more meta.

When going meta, the predictor can use much more sophisticated procedures to predict the user's behavior. It can do inference in a causal model of the user and their environment, it can reason from analogous situations, it can apply heuristics like "the human wants to achieve X," it can use ensemble methods to combine different predictors, and so on.

Indeed, even the idea of "learning processes" is an example of going meta. One way to predict how the user would implement Q is to actually follow the same steps the user would. So when P predicts how the user would answer "Predict how the user would implement Q," it may learn to answer ["Use P to predict how the user will respond to the current context," "Execute the resulting instruction," "Modify the context appropriately," "execute Q in this new context."]

Once this is a learned behavior, it can be improved. The assistant can be taught to optimize computational processes without compromising the results; it can be taught to intelligently decide when it needs to follow the human's process and when it can just

predict the result directly; it can be taught to learn processes on its own or from other kinds of data, and so on.

Most of the process of programming an AI might be replaced by teaching P how to implement new prediction strategies. Any idea which allows an AI to predict, plan, or perceive, can be translated into a procedure for predicting a human's outputs. (Assuming the human can themselves produce the desired output with enough deliberation.) In the degenerate case, we can write functional programs by teaching P to match patterns.

As the assistant becomes better at predicting the user's decisions, it becomes easier to teach it new prediction strategies. Getting the process off the ground may require some challenging bootstrapping.

# Challenges

In this section I'll talk about a few problems with the above system, and how you might address them.

## Remaining "in the counterfactual loop"

You are involved with only a tiny fraction of your assistant's decisions —if you give 100 commands, you only need to implement 1, while your assistant make make thousands, millions, or billions of low-level decisions.

Nevertheless, this architecture requires you to remain "in the counterfactual loop." The right decision for the assistant is defined by what you would recommend, *if* you were asked.

In some sense this is an advantage: if your system is working well, then it won't do anything that you wouldn't. But it could also be a crippling disadvantage: you need to be able to reason about every decision the assistant makes. You might want your assistant to use cognitive strategies that you don't understand, exploit knowledge you don't have, use conventions you don't know, and reason about domains you've never seen. Is this possible?

The key idea is that you don't have to actually know about these decisions. You just have to be able to answer questions about them if asked.

If you are happy to trust a black box you don't understand, you could answer questions just by pointing to a written record. For example, if

you had to implement "Predict how the user would implement Q," you might just look up a default answer in a database and report that. For most users, these responses might be automatic, so that the user is only ever consulted about questions they are likely to care about.

A few users may want to actually supervise the entire reasoning process, either to improve the safety or performance of their own queries, or to help develop the system further.

In these cases, the user needs to be able to understand everything the assistant is doing. But they only need to actually understand something when asked how to implement it—they can learn about it when asked. At that point they can read documentation that describes an algorithm, read comments that describe the assistant's current internal state, receive a tutorial on the format used for some data, be briefed on relevant parts of the assistant's history, or so on. Even if this would be an impossibly difficult task without the assistant's help, with help it might be easy.

This ensures that the assistant is self-documenting—it never does anything that the user can't understand. When the system adopts a new data format or cognitive strategy, explaining the change to the user (counterfactually) is a key part of implementing it. If P is a good enough predictor, then explaining the change to the users is the *only* part of implementing it (once the documentation changes, and the users' counterfactual behavior changes, P's predictions can change immediately). This self-documentation won't be misleading unless the user intentionally creates misleading documentation.

I don't know how much this requirement handicaps the system. Self-documentation is impossible until the system is smart enough to intelligibly describe its own behavior. Once it can describe its own behavior sufficiently well, I would guess that the handicap is minimal.

## Imitation vs. maximization

We might train the predictor P in two different ways:

1. **Imitate**. Provide pairs (Context, Response), and train P to predict the response given the context. This is the proposal described above.

2. **Satisfy.** Evaluate pairs (Context, Response), and train P to produce a response which will score highly given the context.

This would require modifying the training procedure described above.

If the space of responses is small, the two approaches are essentially equivalent. In general, they diverge substantially.

We only need to make this choice at the lowest level—for P itself. When we use metaprediction, all that matter is how we implement "Predict how we would implement Q." We don't need to make a literal prediction. We can "predict" however best suits our needs, in light of any possible concerns with either [1] or [2].

But we do need to make a choice for P itself.

If P is weak, than [1] can be a dangerous option. A weak predictor can't tell what we'll actually do, and a rational "best guess" may be very unattractive. We would prefer to train our system directly to output reasonable answers. (Not all incorrect predictions are equally good.)

If P is strong, then [2] can be a dangerous option. A strong predictor might find outcomes which we would assign a high rating for bad reasons (and these might be the very highest rated options!). Malicious responses might fool us into thinking they are good, or might interfere with our ability to rate them accurately.

[2] is suitable for weak predictors and [1] is suitable for strong predictors. But in reality there is no such dichotomy. Most predictors will be strong in some respects and weak in others, and we won't understand exactly what they do or don't know. So it would be great to capture the benefits of both approaches. I don't know how to do this.

My guess is that either approach [1] or [2] can work, because it only needs to be applied to P. If we want to use [2], we can work with predictors too simple to be ingeniously malicious, and we can use the assistant's help to avoid being fooled by a carefully crafted answer. If we want to use [1], we can dumb down our decisions, and use computational procedures that are robust to the kinds of misunderstandings P might make.

# Conclusion

Computing considered judgments looks far-fetched, but it may be possible to scale down the idea to a more practical framework.

Most of these ingredients can already be implemented. Others can be prototyped using a human's predictions in the place of P.

As usual, much more theoretical work is needed to understand when these ideas would be safe, and especially when they would be as effective as AI systems based on rational agency. But for a change of pace, experimental work may also be able to provide some insight into the feasibility of this framework.