Implementing our considered judgment

Paul Christiano Follow
Jan 18, 2015 · 14 min read

Suppose I had a very powerful prediction algorithm. How might I use this algorithm to build a smart machine that does what I want?

If I could implement a function J which returned the "right" answer to any question, I could build a machine which computed J("What should the machine do?") and did that. As far as I'm concerned, that would be good enough.

It's far from clear what the "right" answer is, but I'm going to describe an algorithm that tries to find it anyway.

(I'm actually happy to settle for a very good answer, rather than the best one. If we could build a very good machine—one which is much more productive than any combination of humans, say—then we could ask it to design a very, very good successor.)

The setup

I'll assume I have a function Predict( ) that takes as input a sequence of observations and predicts the next one.

I'll assume that Predict( ) predicts *very* well. In the future I'll talk about scaling down—using a mediocre predictor to get a mediocre version of the "right" answer—but for now I'll start with a strong assumption and end with a strong conclusion.

I'll assume that Predict( ) can be physically instantiated, and can make reasonable predictions about itself. (It's some work to formalize this, but there are no intrinsic problems with self-reference. If I ask Predict( ) to predict what Predict( ) won't predict, it will just output a uniform distribution.)

The notion of "observation" is going to become a bit confusing, so let's be clear: in the background is some (approximate, implicit) prior over binary sequences x. Predict(x[:n]) outputs a sample from the (approximate) posterior distribution of x[n] given the values of x[0], x[1],..., x[n-1].

I'll assume we have some source of randomness that is unpredictable to Predict( ). This is a very mild assumption. For example, we can use Predict's self-predictions to generate pseudorandomness.

Finally: I'm going to write a program to answer questions, but I'm actually going to have to interact with it each time it answers a question. This may look like a serious drawback, but it's actually quite mild. If I'm tired of making decisions, I can use Predict( ) to lighten the load. Every time it answers a question, it asks for my help with probability 1%. With probability 99%, it just predicts what I would have said if I had helped. (Similar tricks can lower my workload even further.)

The ideal

In this section, I'll describe a formalization of "considered judgment." In the next section, **The implementation**, I'll describe how to use Predict( ) to find our considered judgment.

The answer to any complicated question can depend on the answers to a host of related questions. The length of a flight depends on the weather, the competence of the crew, and the congestion at the destination. The competence of the crew depends on the competence of each member of the crew. The competence of one individual depends on a host of psychological considerations. And so on.

In order to find the right answer to any one question, it would be nice if I could first learn the right answer to each related question that I can think of.

To define my *considered judgment* about a question Q, suppose I am told Q and spend a few days trying to answer it. But in addition to all of the normal tools—reasoning, programming, experimentation, conversation—I also have access to a special oracle. I can give this oracle any question Q', and the oracle will immediately reply with my considered judgment about Q'. And what is my considered judgment about Q'? Well, it's whatever I would have output if we had performed exactly the same process, starting with Q' instead of Q.

Definition

First we have to choose a representation of questions and answers. Let's pick a particular computer and represent Q and A as files stored on this computer, each of size at most one gigabyte.

We also need to define the "oracle" which can tell me my own considered judgment. Let's say there is a special function J which I can use as a black box. J reads one file, corresponding to a question, and then immediately writes the answer.

In order to answer a question Q, I interact with my computer, on which Q is stored as a file. I can do whatever I like—write programs, call the function J, consult a friend, run an experiment, and so on. At the end of a few days, I write a new file A, which records my answer.

The outcome of this process depends both on the initial question Q, and on the behavior of the function J. (It also depends on me and my environment, but let's hold those fixed. Every time we talk about me deciding something, it's the same two days unfolding over and over again, just with a different question Q and a different function J.)

I'll write R[J](Q) for my answer to the question Q, where I am allowed to call the function J.

We can think of R[J] is an "improved" version of J. After all, to compute R[J] I can make any number of function calls to J, so in some sense it should be at least as good.

If in fact J = R[J], then we say that J *reflects my considered judgment*. In this case, further reflection on my views leaves them unchanged; I have reached a reflective equilibrium. There is always at least one (randomized) function J which has this property, by Brouwer's fixed point theorem.

A key feature of this definition is that we can actually compute R[J](Q) if we can compute J. Indeed, we just have to give me a few days to think about it.

Are our considered judgments good?

You might ask: is this a satisfactory notion of "the right answer"? Are the answers it produces actually any good?

Of course, the quality of the answers depends on how I choose to produce them. I am optimistic about my considered judgment because I think there is at least one good strategy, and that we can find it.

In this section, I'll describe my reasons for optimism. In the next section, I'll give some reasons for pessimism.

These are long sections. Once you get the idea, you might want to skip ahead to the section **The implementation** below.

Recursion. I've defined considered judgment as a fixed point of the map J → R[J]. Simple recursion is buried inside this fixed point computation as a special case.

For example, if I wanted to know "Does white win this game of chess?" I could consider each move for white in turn, and for each one compute J("Does white win in *this* position if black gets to play next?"). If white wins in any one of those positions then white wins the original game. If white loses in all of them, then black wins. If J makes correct judgments about each of these possible positions, then R[J] makes a correct judgment about the original one.

To answer each of these sub-questions, I would use a similar procedure. For checkmate positions, I would simply return the correct result. (I need to be careful about drawn games if I want to make sure that my recursion is well-founded, but this is not hard.) By induction, my considered judgment can be perfect about questions like "Does white have a winning strategy in chess?"

Most questions can't be quite so easily broken down into intermediate questions. But even for messy questions—"How could I best get to the moon?"—the ability to exhaustively consider *every* intermediate seems to be very powerful.

Brute force search. A special case of the above is the ability to consider *every* possibility. This reduces the problem of "finding the best X" to the problem of "reliably determining which X is better."

I can use a few techniques to do a brute force search:

- Break down the search space into two pieces, and then recursively find the best item from each one. Then compare them. These breakdowns could be "dumb," such as considering "Proposals that start with 'a'," "proposals that start with 'b'," etc., or they could be context-dependent such as breaking down proposals to get to the moon according to where the energy comes from.
- Break down the possible methodologies you could use into two classes, and then recursively find the best item produced by a methodology from either class. Then compare them. For

example, we could do a brute force search over ways to do a brute force search.

- After searching for a good X and finding our best guess, we can ask "What is the best X?" (Yes, this is the question that *we* were asked.) If the result is better than our current best guess, we can adopt it instead. This implies that if we produce the best X with any probability, it will be identified as the fixed point. As described in the next section, this is a dangerous approach.

I can also do a brute force search for possible flaws in a design or argument, or to evaluate the expected value of a random variable, or so on.

Search for methodologies. In addition to trying to answer a question directly, I can consult my considered judgment to find the best methodologies for answering a question; to find the best frameworks for aggregating evidence; to identify the most relevant lines of reasoning and experiments that I should explore; to find considerations I may have overlooked; to identify possible problems in a proposed methodology; and so on.

Long computations. By storing intermediate results in questions, my considered judgment can reach correct answers about very long-running computations. For example, for any computation C using at most a gigabyte of memory, I could answer "What is the result of computation C?" This can be defined by advancing the computation a single step to C' and then asking "What is the result of computation C'?"

Intellectual development. Suppose I were tasked with a question like "What physical theory best explains our observations of the world?" Using the techniques described so far, it looks like I could come up with answer as good as contemporary physics—by thinking through each consideration in unlimited detail, considering all possible theories, and so on.

Similarly, I would expect my considered judgment to reflect many future intellectual and methodological developments.

Robustness the hard way. Many of the techniques listed so far have possible issues with robustness. For example, if I perform a brute force search, I may find a solution which is very persuasive without being accurate. If I try to explore the game tree of chess without realizing that the game can loop, I may end up with a cycle,

which could be incorrectly assigned any value rather than correctly recognized as a draw.

However, I also have a lot of room to make my computations more robust:

- Every time I do anything I can record my proposed action and ask "Will taking this action introduce a possible error? How could I reduce that risk?"
- I can re-run checks and comparisons in different ways to make sure the results line up.
- I can sanitize my own answers by asking "Is there any reason that looking at this answer would lead me to make a mistake?"
- I can proceed extremely slow in general, splitting up work across more questions rather than trying to bite off a large issue in a single sitting.

Are our considered judgments bad?

On the other hand, there are some reasons to be skeptical of this definition:

Universality? The considered judgment of a 4-year-old about political economy would not be particularly reasonable. In fact, it probably wouldn't be any better than the 4-year-old's knee-jerk reaction. And if I had to come up with answers in a few seconds, rather than a few days, my judgment would be reasonable either.

It's reasonable to be skeptical of a proposed definition of "the right answer," if it wouldn't have helped a stupider version of you—we might reasonably expect a smarter version of ourselves to look back and say "It was a fine process, but they were just not smart enough to implement it."

But I think there is a good chance that our considered judgment is universal in a sense that the 4-year-old's is not. We know to ask questions like "How should I approach this question?" which a 4-year-old would not. And in a few days we have time to process the answers, though we wouldn't in a few seconds.

In simple domains, like mathematical questions, it seems pretty clear that there is such a universality property: the 4-year-old would play a

terrible game of chess even if they could consult their considered judgment, but we would probably play a perfect game.

If there is no universality phenomenon, then the fact that the 4-year-old gets the wrong answer strongly suggests that we will also get the wrong answer. But this might be OK anyway. It at least seems likely that our considered judgment is much wiser than we are.

Malignant failure modes. Suppose that my "considered judgment" about every question was a virus V, an answer which led me to answer whatever question I was currently considering with V. This is a fixed-point, but it doesn't seem like a good one.

Such an extreme outcome seems unlikely. For example, there are some questions that I can answer directly, and I probably wouldn't respond to them with a complicated virus V. It's also not clear whether there is any answer that has this malicious property.

That said, especially as we consider brute force searches or cyclical dependencies, it becomes increasingly plausible that we get unanticipated answers which have surprising effects. Adversarial effects could potentially spread throughout the network of related questions, even if it couldn't literally replicate itself or spread with 100% fidelity.

I am optimistic about this problem because we can painlessly go to heroic lengths to scrutinize our own reasoning at very high levels of detail. I expect there are levels of scrutiny sufficiently great that they could catch all plausible malignant failures. If this weren't possible, we would want to focus our attention on well-founded sets of queries, in order to avoid the implicit optimization power of a fixed point computation.

This problem feels interestingly analogous to computing in the presence of errors. In both cases, the critical question is whether an error-correcting step is itself so error-prone that it introduces more errors than it fixes. This depends on the reliability of the underlying process (and is another possible source of a universality phenomenon) but my intuition is that by proceeding carefully we can remove possible errors from our computations faster than they are introduced.

Experiments and big computations. My considered judgment about the question "Exactly how much does Mars weigh?" is not accurate unless I know enough information to figure out the answer

(or I can collect that information over a few days). Similarly for sufficiently complex computations, which I'll treat as another kind of empirical fact.

It's not clear this is a meaningful drawback. In the intended use case of a machine that consults our considered judgment to figure out "What should I do?", it's not clear that this is a drawback at all.

If we need to know some empirical fact and the machine asks "What should I do?" then our answer may be: "an experiment." But this is the right answer, regardless of what definitions we use. We could define our considered judgment so that it knew the results of all possible experiments. But since those results aren't available to the machine it would need to do the experiment anyway before it could actually act on the result.

Once we have any evidence that bears on an empirical question, then our considered judgment can use that judgment extremely efficiently.

The implementation

We are free to define terms however we like. But can we use Predict( ) to approximate this notion of "considered judgment"?

The first attempt

We'll maintain a log L of questions and answers (encoded in self-delimited binary). By definition, Predict(L) predicts the next item to be added to the log. We'll write Predict(L) for the prediction of the next question or answer rather than the next bit.

We start with a large enough log that Predict can build a good model of the world. Once it's made enough initial errors, it should be able to predict the next log entry as well as a human could. At this point, we start asking questions.

Define J(Q) to be Predict(L+Q), where + is concatenation of lists. That is, J(Q) imagines that Q was appended to the log, and predicts what comes next. We will always follow a question with its "correct" answer, so that J(Q) will try to predict the "correct" answer to Q.

If we want to compute $J(Q_0)$, then with 99% probability we make a single call to Predict. But with 1% probability, we should perform the following "training" process:

1. Compute J(Q0).
2. Generate a random question Q.
3. Determine R[J](Q). That is, I answer the question Q, consulting J and following the procedure described in the definition of considered judgment. Computing J requires only a single call to Predict, so this is feasible. Note that the log L is not changed during this process.
4. Set L = L + Q + R[J](Q).

If Predict( ) makes perfect predictions, then J(Q) must reflect my considered judgment: R[J](Q) always follows L+Q in the log, so J(Q) = Predict(L + Q) = R[J](Q), which is precisely the definition of considered judgment.

To ensure that Predict keeps making good predictions, we might want to continue appending sensor readings to the log, so that Predict understands its current environment and can make good predictions. (If we don't keep the log up to date, Predict should still make reasonable predictions, but it will no longer be aware of context. If my considered judgment depends on context that Predict doesn't know, then its predictions won't be so good.)

A problem

We hope that J(Q) reflects my considered judgment even if Predict( ) is sometimes wrong. Unfortunately, it doesn't.

The problem is that Predict is trained to answer random questions—only random questions+answers every appear in its log. A machine which answers 99.999% of random questions "correctly" need not do anything useful at all, since 99.999% of all possible questions are meaningless garbage.

When we ask Predict(L+Q') a *real* question Q', one that actually makes sense, it could be wrong 100% of the time, without having any serious effect on Predict( )'s overall performance.

The same problem might affect the human as well. J(Q') = Predict(L + Q') is supposed to predict what the human would output, if Q' was generated as a random question. But what does a human do if they flip 100,000 coins and find that they spell out a meaningful question about the methodology of science?

The human may (correctly) infer that something fishy is going on. Hopefully they would continue to behave sensibly, but it's hard to predict and essentially impossible to test. This is closely linked to problems that Wei Dai and Vladimir Slepnev have raised with an earlier definition of enlightened judgment.

The second attempt

In order to fix this problem, we would like to ensure that Predict is trained using the same kinds of questions that J has to answer. We can do this with a nice trick.

Let Q be our random "training" question, and let Q* be a random question that we ask in the course of answering Q. We want the distribution of Q to be the same as the distribution of Q*. We will do this by asking the predictor to sample Q from its predicted distribution over Q*, rather than generating Q uniformly at random.

To this end, before asking a question we either insert a 0 or a 1 into the log. 0 indicates that we will give the predictor a question, and then expect it to predict the answer. 1 indicates that we want the predictor to predict the next question, but we won't tell it the answer.

We define J(Q) = Predict(L + [0] + Q). If we want to compute J(Q_0), then with probability 99% we make a single call to Predict. But with probability 1%, we perform the following process:

1. Compute J(Q_0).
2. Let Q = Predict(L+[1]). That is, sample a question from the predicted distribution.
3. Determine R[J](Q) = R[Q' → Predict(L+[0]+Q')].
4. Let Q* be a random question that J answered in step 1 or step 3.
   With probability 1/2, set L = L + [0] + Q + R[J](Q).
   With probability 1/2, set L = L + [1] + Q*.

As before, if Predict does a good job then Predict(L+[0]+Q) will be R[J](Q), and so J implements our considered judgment.

But now, rather than being uniformly random, Q is distributed according to Predict(L+[1]). If Predict( ) is making good predictions, this is the distribution of Q*, a random question that was posed to J. So Predict( ) is being trained with exactly the same distribution of questions that J is being asked to answer.

Conclusion

Given a sufficiently powerful predictor Predict( ), we can implement a function J which reflects our considered judgment. If we trust the definition of considered judgment, then a machine that asks J("What should I do?") is almost as good as a machine that acts optimally.

With a more realistic predictor we will have to scale back our ambitions, and instead try to produce a the "most right" answer we can. Although this will require many new ideas, I am optimistic about scaling down to obtain reasonable behavior using significantly sub-human predictors.

I hope that the case of a powerful predictor helps illustrate what it might mean for a machine to do "the right thing," and that the concrete example can help move forward the theoretical discussion about AI safety.