Thoughts on reward engineering

Paul Christiano Follow
Nov 8, 2016 · 13 min read

Suppose that I would like to train an RL agent to help me get what I want.

If my preferences could be represented by an easily-evaluated utility function, then I could just use my utility function as the agent's reward function. But in the real world that's not what human preferences look like.

So if we actually want to turn our preferences into a reward function suitable for training an RL agent, we have to do some work.

This post is about the straightforward parts of reward engineering. I'm going to deliberately ignore what seem to me to be the hardest parts of the problem. Getting the straightforward parts out of the way seems useful for talking more clearly about the hard parts (and you never know what questions may turn out to be surprisingly subtle).

The setting

To simplify things even further, for now I'll focus on the special case where our agent is taking a single action a. All of the difficulties that arise in the single-shot case also arise in the sequential case, but the sequential case also has its own set of additional complications that deserve their own post.

Throughout the post I will imagine myself in the position of an "overseer" who is trying to specify a reward function $R(a)$ for an agent. You can imagine the overseer as the user themselves, or (more realistically) as a team of engineer and/or researchers who are implementing a reward function intended to expresses the user's preferences.

I'll often talk about the overseer computing $R(a)$ themselves. This is at odds with the usual situation in RL, where the overseer implements a very fast function for computing $R(a)$ in general ("1 for a win, 0 for a draw, -1 for a loss"). Computing $R(a)$ for a particular action a is strictly easier than producing a fast general

implementation, so in some sense this is just another simplification. I talk about why it might not be a crazy simplification in section 6.

Contents

1. Long time horizons

A single decision may have very long-term effects. For example, even if I only care about maximizing human happiness, I may instrumentally want my agent to help advance basic science that will one day improve cancer treatment.

In principle this could fall out of an RL task with "human happiness" as the reward, so we might think that neglecting long-term effects is just a shortcoming of the single-shot problem. But even in theory there is no way that an RL agent can learn to handle arbitrarily long-term dependencies (imagine training an RL agent to handle 40 year time horizons), and so focusing on the sequential RL problem doesn't address this issue.

I think that the only real approach is to choose a reward function that reflects the overseer's expectations about long-term consequences—

i.e., the overseer's task involves both making predictions about what will happen, and value judgments about how good it will be. This makes the reward function more complex and in some sense limits the competence of the learner by the competence of the reward function, but it's not clear what other options we have.

Before computing the reward function R(a), we are free to execute the action a and observe its short-term consequences. Any data that could be used in our training process can just as well be provided as an input to the overseer, who can use the auxiliary input to help predict the long-term consequences of an action.

2. Inconsistency and unreliability

A human judge has no hope of making globally consistent judgments about which of two outcomes are preferred—the best we can hope for is for their judgments to be right in sufficiently obvious cases, and to be some kind of noisy proxy when things get complicated. Actually outputting a numerical reward—implementing some utility function for our preferences—is even more hopelessly difficult.

Another way of seeing the difficult is to suppose that the overseer's judgment is a noisy and potential biased evaluation of the quality of the underlying action. If both R(a) and R(a') are both big numbers with a lot of noise, but the two actions are actually quite similar, then the difference will be dominated by noise. Imagine an overseer trying to estimate the impact of drinking a cup of coffee on Alice's life by estimating her happiness in a year conditioned on drinking the coffee, estimating happiness conditioned on not drinking the coffee, and then subtracting the estimates.

We can partially address this difficulty by allowing the overseer to make *comparisons* instead of assessing absolute value. That is, rather than directly implementing a reward function, we can allow the overseer to implement an antisymmetric comparison function C(a, a'): which of two actions a and a' is a better in context? This function can take real values specifying *how much* one action is better than another, and should by antisymmetric.

In the noisy-judgments model, we are hoping that the noise or bias of a comparison C(a, a') depends on the actual magnitude of the difference between the actions, rather than on the absolute quality of each action. This hopefully means that the total error/bias does not drown out the actual signal.

We can then define the decision problem as a zero-sum game: two agents propose different actions a and a', and receive rewards C(a, a') and C(a', a). At the equilibrium of this game, we can at least rest assured that the agent doesn't do anything that is *unambiguously worse* than another option it could think of. In general, this seems to give us sensible guarantees when the overseer's preferences are not completely consistent.

One subtlety is that in order to evaluate the comparison C(a, a'), we may want to observe the short-term consequences of taking action a or action a'. But in many environments it will only be possible to take one action. So after looking at both actions we will need to choose at most one to actually execute (e.g. we need to estimate how good drinking coffee was, after observing the short-term consequences of drinking coffee but without observing the short-term consequences of not drinking coffee). This will generally increase the variance of C, since we will need to use our best guess about the action which we didn't actually execute. But of course this is a source of variance that RL algorithms already need to contend with.

3. Normative uncertainty

The agent is uncertain not only about its environment, but also about the overseer (and hence the reward function). We need to somehow specify how the agent should behave in light of this uncertainty. Structurally, this is identical to the philosophical problem of managing normative uncertainty.

One approach is to pick a fixed yardstick to measure with. For example, our yardstick could be "adding a dollar to the user's bank account." We can then measure C(a, a') as a multiple of this yardstick: "how many dollars would we have to add to the user's bank account to make them indifferent between taking action a and action a'?" If the user has diminishing returns to money, it would be a bit more precise to ask: "what chance of replacing a with a' is worth adding a dollar to the user's bank account?" The comparison C(a, a') is then the inverse of this probability.

This is exactly analogous to the usual construction of a utility function. In the case of utility functions, our choice of yardstick is totally unimportant—different possible utility functions differ by a scalar, and so give rise to the same preferences. In the case of normative uncertainty that is no longer the case, because we are

specifying how to *aggregate* the preferences of different possible versions of the overseer.

I think it's important to be aware that different choices of yardstick result in different behavior. But hopefully this isn't an *important* difference, and we can get sensible behavior for a wide range of possible choices of yardstick—if we find a situation where different yardsticks give very different behaviors, then we need to think carefully about how we are applying RL.

For many yardsticks it is possible to run into pathological situations. For example, suppose that the overseer might decide that dollars are worthless. They would then radically increase the value of all of the agent's decisions, measured in dollars. So an agent deciding what to do would effectively care much more about worlds where the overseer decided that dollars are worthless.

So it seems best to choose a yardstick whose value is relatively stable across possible worlds. To this effect we could use a broader basket of goods, like 1 minute of the user's time + 0.1% of the day's income + *etc*. It may be best for the overseer to use common sense about how important a decision is relative to some kind of idealized influence in the world, rather than sticking to any precisely defined basket.

It is also desirable to use a yardstick which is simple, and preferably which minimizes the overseer's uncertainty. Ideally by standardizing on a single yardstick throughout an entire project, we could end up with definitions that are very broad and robust, while being very well-understood by the overseer.

Note that if the same agent is being trained to work for many users, then this yardstick is also specifying how the agent will weigh the interests of different users—for example, whose accents will it prefer to spend modeling capacity on understanding? This is something to be mindful of in cases where it matters, and it can provide intuitions about how to handle the normative uncertainty case as well. I feel that economic reasoning is useful for arriving at sensible conclusions in these situations, but there are other reasonable perspectives.

4. Widely varying reward

Some tasks may have widely varying rewards—sometimes the user would only pay 1¢ to move the decision one way or the other, and sometimes they would pay $10,000.

If small-stakes and large-stakes decisions occur comparably frequently, then we can essentially ignore the small-stakes decisions. That will happen automatically with a traditional optimization algorithm—after we normalize the rewards so that the "big" rewards don't totally destroy our model, the "small" rewards will be so small that they have no effect.

Things get more tricky when small-stakes decisions are much more common than the large-stakes decisions. For example, if the importance of decisions is power-law distributed with an exponent of 1, then decisions of all scales are in some sense equally important, and a good algorithm needs to do well on all of them. This may sound like a very special case, but I think it is actually quite natural for there to be several scales that are all comparably important *in total*.

In these cases, I think we should do importance sampling—we oversample the high-stakes decisions during training, and scale the rewards down by the same amount, so that the contribution to the total reward is correct. This ensures that the scale of rewards is basically the same across all episodes, and lets us apply a traditional optimization algorithm.

Further problems arise when there are some *very* high-stakes situations that occur very rarely. In some sense this just means the learning problem is actually very hard—we are going to have to learn from few samples. Treating different scales as the same problem (using importance sampling) may help if there is substantial transfer between different scales, but it can't address the whole problem.

For very rare+high-stakes decisions it is especially likely that we will want to use simulations to avoid making any obvious mistakes or missing any obvious opportunities. Learning with catastrophes is an instantiation of this setting, where the high-stakes settings have only downside and no upside. I don't think we really know how to cope with rare high-stakes decisions; there are likely to be some fundamental limits on how well we can do, but I expect we'll be able to improve a lot over the current state of the art.

5. Sparse reward

In many problems, "almost all" possible actions are equally terrible. For example, if I want my agent to write an email, almost all possible strings are just going to be nonsense.

One approach to this problem is to adjust the reward function to make it easier to satisfy—to provide a "trail of breadcrumbs" leading to high reward behaviors. I think this basic idea is important, but that changing the reward function isn't the right way to implement it (at least conceptually).

Instead we could treat the problem statement as given, but view auxiliary reward functions as a kind of "hint" that we might provide to help the algorithm figure out what to do. Early in the optimization we might mostly optimize this hint, but as optimization proceeds we should anneal towards the actual reward function.

Typical examples of proxy reward functions include "partial credit" for behaviors that look promising; artificially high discount rates and careful reward shaping; and adjusting rewards so that small victories have an effect on learning even though they don't actually matter. All of these play a central role in practical RL.

A proxy reward function is just one of many possible hints. Providing demonstrations of successful behavior is another important kind of hint. Again, I don't think that this should be taken as a change to the reward function, but rather as side information to help achieve high reward. In the long run, we will hopefully design learning algorithms that automatically learn how to use general auxiliary information.

6. Complex reward

A reward function that intends to capture all of our preferences may need to be very complicated. If a reward function is implicitly estimating the expected consequences of an action, then it needs to be even more complicated. And for powerful learners, I expect that reward functions will need to be learned rather than implemented directly.

It is tempting to substitute a simple proxy for a complicated real reward function. This may be important for getting the optimization to work, but it is problematic to change the definition of the problem.

Instead, I hope that it will be possible to provide these simple proxies as hints to the learner, and then to use semi-supervised RL to optimize the real hard-to-compute reward function. This may allow us to perform optimization even when the reward function is many times more expensive to evaluate than the agent itself; for example, it might allow a human overseer to compute the rewards for a fast RL

agent on a case by case basis, rather than being forced to design a fast-to-compute proxy.

Even if we are willing to spend much longer computing the reward function than the agent itself, we still won't be able to find a reward function that perfectly captures our preferences. But it may be just as good to choose a reward function that captures our preferences "for all that the agent can tell," i.e. such that the conditioned on two outcomes receiving the same expected reward the agent cannot predict which of them we would prefer. This seems much more realistic, once we are willing to have a reward function with much higher computational complexity than the agent.

Conclusion

In reinforcement learning we often take the reward function as given. In real life, we are only given our preferences—in an implicit, hard-to-access form—and need to engineer a reward function that will lead to good behavior. This presents a bunch of problems. In this post I discussed six problems which I think are relatively straightforward. (Straightforward from the reward-engineering perspective—the associated RL tasks may be very hard!)

Understanding these straightforward problems is important if we want to think clearly about very powerful RL agents. But I expect that most of our time will go into thinking about harder problems, for which we don't yet have any workable approach. These harder problems may expose more fundamental limits of RL, that will require substantially new techniques to address.

Appendix: harder problems

Informed oversight

The process that produces a decision may encode important information about the quality of that decision, and if we throw out that information then a very bad decision may nevertheless receive a high reward. For example, if we want an agent to produce original art, we may not be able to evaluate a painting without understanding the process that produced it (and, in particular, what other artwork it was inspired by).

In some cases, recovering information about the process seems feasible. For example, if the agent is using attention or an external

memory, the overseer can check what art the agent looked at. But in other cases it seems much more challenging: if the agent's weights have memorized properties of particular artwork, we would need new techniques in order to make that information available to the overseer.

It is not clear whether there is any general way to give the overseer enough insight into the agent's thought process to avoid problems like this; there may be some inherent limitations on delegation. Either way, it would be nice to understand the situation much better.

Capability amplification

By folding our preferences and long-term predictions into the reward function, we are in some sense limiting the agent to be no stronger than the overseer. If we start with a human, then RL will only ever produce human-level reasoning about long-term consequences or about "what is good."

So we need some other mechanism for producing better long-term reasoning. I think that the most promising approach is to treat "reasoning well" as a short-term problem, for which humans can provide adequate oversight, and then to run that reasoning process for longer in order to produce more powerful reasoning.

Reward hacking / security

If our reward function has any problems, a powerful RL agent may tend to find actions a that exploit those problems in order to receive a high reward. In order to get good consequences, our reward function needs to be sufficiently secure that the agent can't attack it in this way.

The first line of defense against an attack is to avoid immediately executing the agent's action. Instead, the overseer can inspect that action and try to ensure that it isn't intended as an attack. This leaves attacks which the overseer cannot recognize as attacks, or which do damage even when the overseer looks at them.

If the techniques from the previous sections actually allow the overseer to evaluate the agent's actions, then they can also probably allow the overseer can detect attacks. Security during evaluation itself is an additional question though.

The main cause for hope is if the overseer can (1) be smarter than the agent which is trying to attack it, and (2) have access to some

information about the agent's thought process. Hopefully (2) allows the overseer to overcome the disadvantage of the "position of the interior"—if the agent picks a particular attack vector, the overseer can "watch them thinking" and then devote its energies to trying to detect or defend against that particular attack.