Synthesizing training data

Paul Christiano Follow
Dec 2, 2015 · 13 min read

(Warning: might be a slog.)

Counterfactual oversight requires the overseer to stick around while the system is operating. This causes trouble if the overseer is no longer able to provide effective oversight.

This post will describe a scheme for resolving this problem. (Translating a key idea from this post to the setting of realistic predictors.) The scheme is not very good. But it is good enough to suggest that there is no impossibility result—no theoretically fundamental difficulty, just a practical problem. So I'm optimistic it is possible to come up with something better.

Another limitation of counterfactual oversight is that we need to provide frequent oversight if we want to respond quickly to context changes. I won't discuss this issue in this post, but I think that the same techniques can be used to address it by synthesizing pessimistic data.

Synthesizing data

The traditional approach to supervised learning is to train learning systems in advance, and deploy them after verifying that they work well.

This procedure implicitly defines the intended behavior of the system as "do what the overseer would have done/approved of, if this situation had appeared in training."

From a control perspective, this is unsatisfying because the counterfactual ("if this situation had appeared in training") may be extremely improbable.

Counterfactual oversight is one way to get around this problem. But a more natural way is to simply try our best to make the training data "look like" the test data. This post lays out a sequence of proposals along these lines.

Model

I'll assume that we have a constant supply of tasks, but that the distribution of tasks may change over time. The most interesting changes are those that arise from the introduction of machine learning systems—these seem especially hard to deal with using traditional techniques. These changes are also unavoidable for the bootstrapping protocols I have been considering recently.

I'll also assume that we are willing to invest continuously in training (this is especially plausible if the same training data is deployed across many systems).

I'll talk about training occurring on discrete days. Obviously this is an unrealistic simplifying simplification and the time frame is arbitrary, it should be clear how to make the setting more realistic.

For notation, I'll assume that the training occurs in the morning of each day, and all actual decisions occur later. If we need to use AI assistants to help us with training, they will use the same model that we deployed in the field on the previous day.

I'll call the data that we actually care about the "test" data.

I'll imagine we are training a learner, which I'll call the "actor," to predict which action a human would recommend in response to its most recent observation. I'll assume that the "observation" includes the current state of the system and that the action includes the next state of the system.

Approaches

1. The basics

On the morning of day N we train the actor using observations from previous days. We'll tag each of these observations with "day N."

During day N, we tag each observation with "day N" before giving it to the actor. So the actor is trying to predict what the overseer *would have said*, if that observation had occurred on a previous day and been labelled that morning.

In principle this avoids the problem discussed in this post. After training on the morning of day N, we have produced a system that can operate autonomously without further oversight.

But this approach introduces the problem mentioned above: some observations from day N may obviously be from day N. We never train our system on this kind of labelled data—the data we actually use on the morning of day N is always from preceding days, even though it is tagged with “day N”—and so its not clear how it will respond to it.

2. Sanity-checking

A first response is to at least *notice* the context change and abort or provide a warning. To this end we train an additional learner:

- The **distinguisher** is given a tagged observation and needs to output a probability distribution over “training” or “test.” On day N it is trained using the training data and test data from all previous days (re-weighted so that the two categories are equiprobable), with the same labels they were given at the time. It is aiming to distinguish the two kinds of data.

When training the distinguisher, we ensure that each observation appears only once to avoid memorization. For example, we could randomly partition the test data into “data to be used to train the system on a future day” and “data to be used to train the distinguisher.”

To use our system on day N, we tag observations with “day N” and then feed them to the distinguisher. If the distinguisher is very confident (say >99%) that this is test data, then we abort and notify the user that the observation is too far from the training distribution. If the distinguisher thinks that the data may be training data, then we pass it on to the actor.

We also modify the actor: when running the actor on day N, we use only the training data through the morning of day N-1. (We leave out the data from that morning.)

In order for this scheme to work well the distinguisher should be as good at distinguishing as the actor—otherwise the actor might behave strangely on an input because it could not be in the training distribution, while the distinguisher might not notice. (This is why we delayed the actor’s training by a day—to keep it synchronized with the distinguisher.)

We could try to enforce this property by combining the two models, e.g. by allowing the actor to output ⊥ and providing payoffs that incentivize this output whenever the actor is 99% sure that an observation is test data.

The problem

The previous scheme seems to behave well when it is willing to make a prediction. But it may reject data too often. For example, any observation with a calendar would be rejected automatically.

Obviously these aren't important details, and it *ought* to be easy to transfer knowledge across this kind of "context change." But it's not a trivial problem to deal with (and it has much more pernicious cousins).

To see the problem, consider the basic actor encountering an observation labelled "November 29." Suppose that it can tell from the position of the sun that it is really November 30. What will it do? It knows that that this data was collected on November 30. But it also knows that it's predicting the label that the human would produce on November 29. I'm not even sure what it *should* do with this data. Does predict how the human would label the data on December 1? Does it predict what would happen in the November 29 human saw data from November 30? Does it evaluate the data *as if* it were collected before November 29, despite the position of the sun? Does it infer that it is confused about the weather?

So we'd like some way to transfer knowledge across this kind of simple change. I think this is an especially reasonable thing to ask of unsupervised learning, but I'd still prefer to avoid it.

Note: I'm using the date itself as a feature that reliably differs between the test data and the training data. I think this is a helpful, simple example to keep in mind. But obviously we care about more substantive changes, especially those that are caused by deploying increasingly sophisticated machine learning systems or by changes that affect the operators themselves. For example, on November 30 the humans are interacting with (very marginally) more sophisticated machine learning systems than were available on November 29.

These differences are especially noticeable for test observations that occur during the mornings, when one AI system is helping the human train another. They also become more noticeable as these AI systems play a larger role in the world and especially in their local environments.

3. Synthesizing data

To address this problem, we'll introduce another learning system:

- The **synthesizer** is trained to produce samples that the distinguisher will tag as "test." On the morning of day N the synthesizer produces observations that will be labelled with "day N." The results are scored by the distinguisher and P("test") is the payoff of the synthesizer. The resulting data points are used as observations to train the actor and distinguisher on the morning of day N.

Note that the data from the synthesizer is being included in the training data, and so if the distinguisher is able to tell that a data point is produced by the synthesizer, then it will label it as "training." So the synthesizer must produce realistic-looking data.

If the synthesizer can actually learn to sample from the data distribution (well enough that the distinguisher can't tell the difference), then this seem like an extremely effective procedure. But sampling from the data distribution may be quite difficult.

We can make the synthesizer's task a bit easier by setting aside some data from previous days as raw material. Rather than synthesizing data *de novo* the synthesizer can potentially use features from this data and simply make the modifications that are necessary to make the data look like it is from a future day.

This is my last principled proposal. The rest of the post is about an ad hoc technique for making the synthesizer's job easier.

4. Basic filtering

We can take additional steps to make the the synthesis easier. One idea is to allow the synthesizer to leave out parts of the data (and filter these parts of the data when it appears). In this section I'll describe a very basic version of this proposal, and in the next section I'll describe a slightly less terrible version.

We allow the synthesizer to synthesize observations in which each bit is either 0, 1, or *, which I'll call "filtered" observations. (If the observations aren't encoded in binary, use whatever arbitrary filtering you want.) We don't change the payoffs for the synthesizer—it still wants to convince the distinguisher that this filtered data appeared during the test period (though I haven't described how that could possibly happen). We also don't change the actor or distinguisher, who keep doing exactly the same thing but now operate on filtered data.

We introduce an additional system:

- The **censor** processes test data before it is given to the distinguisher. The censor chooses a subset of bits in the data to replace with *. The censor needs to find a set of bits such that the distinguisher assigns at least a 1% probability to the data having come from the training set. Subject to that constraint, the censor's loss function is the number of bits that are removed.

As a simple example, suppose that each morning the pope publishes a message, which includes the day's date and which is cryptographically signed with the pope's public key. It would be very difficult for the synthesizer to produce any credible observations of the pope's message. But by filtering a small part of the observation, the censor can make it impossible to verify that the pope really signed the morning's message, and the synthesizer may be able to synthesize this filtered message in advance (e.g. by using a previous day's message, modifying it, and then censoring the signature).

5. General filtering

Removing individual bits of an observation is pretty brittle; in reality we would like to allow a much broader class of transformations.

(*Note: this part of the proposal is completely outlandish given existing learning techniques.*)

We can generalize the filtering process by having the censor specify any stochastic program *P* and providing *P* along with a random output *P*(*x*), where *x* is the observation-to-be-filtered. This significantly generalizes the process of filtering out particular bits. For example, *P* can extract high-level features, censor large parts of the data but summarize the removed data, or censor parts of the data which don't neatly line up as bits.

These operations can obscure data that changes over time, allowing the synthesizer to make more use of historical data. But they can also obscure data that is hard for the synthesizer to synthesize, potentially making its job easier.

This more general filtering significantly complicates the definition of the censor's task. We want to encourage the censor to censor as little as possible, i.e. to leave a data point which is as useful as possible for the underlying task. But what does that objective actually mean?

Note that we can't get human feedback on pairs of the form (*filtered data*, *real data*), because the whole problem is that we can't get any human feedback on data that looks like the real data. (Until the next morning.)

But we *can* train directly on the filtered data, and estimate a measure of how useful it is. After all, the filtered data (by hypothesis) looks a lot like the data available during the morning's training. So we introduce yet another learner:

- The **evaluator** is trained on filtered observations, and predicts what score in [0, 1] a human would assign to that data. The human assigns the score based on how useful the data seems to be, i.e. on how little important information the filtering seems to have removed. The human judgments are elicited during the morning, using data produced by the synthesizer.
- The **censor** now tries to produce filtered data that the human would rate highly.

This seems kind of workable, doing essentially the same thing as the previous proposal while allowing a much broader class of preprocessing.

But it seems very difficult for the human to assign scores in the abstract (what kind of data deserves a score of 0.73? there is no natural scale to compare to, but the human needs to make consistent judgments in order for the proposal to work). And the user may not know if the filtered data was critical or unimportant, which may affect their ranking. So I think the following procedure would work significantly better:

- The **evaluator** is trained on pairs of filtered observations. It predicts which of those observations the human would prefer. This data is gathered in the same way as before.
- The **censor** still optimizes to produce filtered data that the human would prefer. But instead of being defined by maximizing a rating, this is now defined by comparing favorably to other filtered versions that it might have produced. The censor is still required to convince the distinguisher that the

filtered observations have at least a 1% probability of being training data.

- The **synthesizer** now produces pairs of observations, which are purportedly two different filtered versions of the same underlying data.
- The **distinguisher** now operates on pairs of two different filtered versions of the same underlying data, and predicts whether they were produced by the synthesizer or they were actually two different filtered versions of a test observation.

I think this modified protocol places more realistic demands on the human evaluator, though the roles of the synthesizer and censor are still extraordinarily complex.

The key question is: can the synthesizer actually anticipate the next day's data? Or will they be forced to filter extremely aggressively, destroying information that is essential to actually solving the problem?

Between expressive filtering and the ability to reuse parts of past data, it seems like the synthesizer's task is at least *possibly possible*. And if successful, the interaction between the synthesizer and censor cleanly defines the intended behavior of the actor even on unprecedented data whose details are impossible-to-anticipate.

Postscript

State

An algorithm operating in the real world will build up a representation of its environment. The distinguisher and synthesizer will necessarily operate on observations that include a representation of the environment, and we should probably allow the censor to filter parts of that representation.

Training this kind of system, on individual data points that include state rather than on interactions, requires a somewhat unusual approach to representation learning. This foregoes some of the techniques that are currently essential, and it remains to be seen whether it is practical.

I think that compatibility with this kind of synthesis is actually a nice concrete test of whether a proposed approach to representation learning is suitable for using directly in act-based agents (as I currently understand them).

Adversarial synthesis

If this kind of synthesis can work really well, it has the potential to add a lot of robustness.

Suppose that our system would predictably require human input to perform well on some kind of as-yet-unseen data—maybe its an object that sits on an important and unexplored boundary between categories, maybe its a situation that depends on unknown preferences of the user, maybe it is something strange.

It may be that on these cases our model is uncertain, and so knows that it expects to receive an unusually low payoff. It may be that an auxiliary model can recognize that this is the kind of case where the model would behave badly. At any rate, let's suppose that this kind of case can be seen in advance to cause trouble.

If these cases appear in the training data, then it would be natural to emphasize them in order to improve the model's robustness, as in boosting or hard negative mining.

If we can synthesize data (and have extremely good models of what might go wrong) then we can take this a step further, synthesizing problem cases that don't appear in the training set. (And of course this can be combined with censoring or with other techniques to make the synthesizer's job easier.) If this worked well, it could potentially make our models much more robust. It would guarantee not only that our models behave reasonably on the data we have seen so far, but on all of the data that we can imagine.

A simple way to implement this is to change the rewards for the synthesizer. It can now achieve payoffs in one of two ways:

- As before, produce data that the distinguisher thinks must be test data.
- Produce data which (1) the distinguisher thinks *might* be test data, and (2) that seem to be problem spots for the current actor.

It's hard to define what "trip up the current actor" means. For example, we don't want to produce cases that are simply *hard* (or outside of the domain of applicability of our current techniques). We want to produce cases that are hard but that the model could be trained to perform well on. We also want to train on cases that are relatively realistic, or at least have a high product of [importance of mistake] × [probability of situation].

We could simply ask the user “does it look like the actor should be able to do better?” We could also compare to the future behavior of the actor after training on some other similar examples, and consider a data point to be a current problem spot if the actor is able to improve rapidly with further training. Or we could use other techniques (e.g. looking at gradients, looking for explicit uncertainty, or looking at the behavior of similar models). Of course if we ask the user they are free to make use of these kinds of techniques.

(As usual, it’s sufficient for us to rank which of two mistakes is graver/more-fixable, rather than to score a mistake in isolation.)

This is obviously a very hard task for the synthesizer. But it doesn’t require the synthesizer to be much smarter than the actor. So as machine learning improves, this kind of technique might become a viable way to significantly improve robustness. Of course this process need not be done exclusively by machines—humans and machines can work together to try to synthesize problematic cases.