Using Reinforcement Learning for Probabilistic Program Inference

Avi Pfeffer
Charles River Analytics
apfeffer@cra.com

1. INTRODUCTION

Inference in probabilistic programming often involves choosing between different methods. For example, one could use different algorithms to compute a conditional probability, or one could sample variables in different orders. Researchers have taken a variety of approaches to handle the array of choices. Mansinghka [1] advocates meta-programming, in which a user guides the solution interactively. Alternatively, we [2] have presented an approach that decomposes inference problems into small subproblems and optimizes each separately. In general, the problem of optimizing inference falls into the general area of programming by optimization [3].

In this abstract, we explore the use of reinforcement learning (RL) in a novel way to optimize inference. In this approach, we automatically adjust how inference is performed based on seeing how various approaches are performing. A given inference task might involve many choices. Each of these choices is optimized by a separate RL. In this way, we get a network of interacting learners for an inference problem. We first describe our general approach and then describe three particular kinds of strategies.

2. APPROACH

We represent the result of a computation as a (possibly infinite) stream of successive approximations to an answer. When an exact answer can be reached in a finite number of steps, the stream will be finite, otherwise the stream will converge to the answer. We use Haskell's laziness to implement this idea.

If there are alternative ways to perform a computation, we explore between the multiple streams to produce a single output stream that contains selected values of each of the input streams. We use a *choosing strategy* to choose which stream to get the next value from at each time. In other situations, we have to combine multiple input streams to perform a computation. We use a *combining strategy* for this. Finally, we might be given a stream of streams where we have to enumerate all the elements of all the streams in a single stream. We use *merging strategies* for this.

3. CHOOSING STRATEGIES

In this case, we have a finite number of streams, all representing the answer to the same computation, and have to choose values from each to produce the output stream. This is naturally cast as a multi-armed bandit problem. For our experiments, we used Tokic and Palm's value-difference based exploration [4] as the RL algorithm. This algorithm intelligently adjusts the exploration rate over time based on observed rewards, using a softmax distribution to choose between actions when exploring. We are exploring the use of other reinforcement algorithms as well.

An example application is choosing between multiple methods that converge to a solution, where some might converge faster than others but it is unknown which is better. Since the variance of successive approximation is an indication of how poorly a stream is converging, one possibility is to use the negative variance of a stream as the reward for an item. Specifically, if the number of items we have already taken from the stream is n, the previous estimate from the stream is x, the previous reward is r and the new estimate is x', the new reward is $\frac{rn-(x-x')^2}{n+1}$. Like the choice of RL algorithm, we are exploring alternative reward functions.

For the output estimate of the choosing strategy, we use the last element extracted from the stream with the most elements extracted. This is for two reasons. First, the RL method will tend to extract more elements from the better streams, and second, the further we go down a stream, the better we expect its estimate to be. An alternative is to take a weighted average of the stream estimates, but we found this to work less well.

Another example of a choosing strategy is conditioning a variable that depends on other variables. For example, suppose we have a definition $z = x + y$, and some evidence specified by a predicate ϕ on z. One approach is to generate values of x and y, compute z, and check that it satisfies ϕ. An alternative is to generate x and then

generate y conditioned on the predicate $\lambda y.\phi(x+y)$. We could similarly generate y first and condition x. The advantage of this is that we might be able to propagate the evidence higher in the network, resulting in fewer rejections. But how do we know whether to propagate the condition to x or y? We can use a choosing strategy.

We found the runtime overhead from RL to be negligible compared to the strategies themselves. Figure 1 shows results for computing a probability query on a four-variable mixture model. Three strategies were considered: a sampling strategy, a support enumeration strategy described in a companion paper, and a choosing strategy over the two. The horizontal axis shows the size of streams generated for each of the strategies, while the vertical axis shows the log error of the RL strategy compared to the worst and best strategies. Results are averaged over 20 runs. Even with as few as 100 items in the stream, the RL strategy identifies the best stream 85% of the time. This goes up to 90% with 1000 items; however, it failed to go up to 100% with even more items.

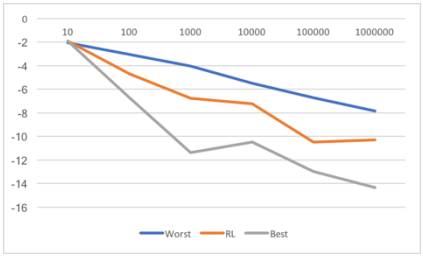

Figure 1: Ratio of error of choosing strategy to error of worse strategy

There are a couple of deficiencies in our approach. First, the Tokic and Palm algorithm has a number of parameters to set. While we found the results so far to be somewhat robust to parameter variations, we could not find parameters that made it work perfectly, and perhaps a different RL algorithm is needed. Second, our current implementation does not take into account the time taken by the different strategies. This can be fixed by timing the different strategies and incorporating it into the reward.

4. COMBINING STRATEGIES

Consider computing the density of a value `v` of a variable x defined by `if y then z else w`. The density is computed by `(density y True) * (density z v) + (1 - density y True) * (density w v)`. If the three densities are streams, we need to combine elements of each of the three streams to produce the result. The further we go down each stream, the better the estimate of that density will be. However, we do not know in advance which stream contributes the most to the result, and we also don't know how much proceeding down a stream improves its estimate.

We use a combining strategy to handle this case. This is similar to a choosing strategy, except that now the reward for a value in a stream depends not only on that stream but also on the other streams, and their contribution to the overall estimated result. Since we are trying to improve a single combined estimate, rather than select among a number of several estimates, and since proceeding along each of the input streams generally improves the output, we want to choose items that most change the current estimate. Therefore, we use the absolute difference between the new estimate resulting from an item and the previous estimate as the reward. We have implemented this strategy and obtained successful results with mixture models.

5. MERGING STRATEGIES

When we have a variable that depends on a variable with infinite support, the density of the child is the sum of contributions of infinitely many variables (see the companion paper submitted to this workshop, on a support-based method for answering queries, for details). Each of these contributions may itself be an infinite stream, and we need to merge this stream of streams into a single stream.

One simple, unintelligent strategy is to produce triangles. We proceed in iterations. In iteration n, we take the next element of the first n streams, in reverse order. This ensures that all elements of all streams will eventually be taken while staying somewhat balanced.

A more intelligent strategy treats this as an infinite-armed bandit problem. At any point in time, the first n streams are active. We always look ahead one step in each of the active streams. At each step, we choose the best element from the active streams. Whenever we choose the first element of the nth stream, we make the $(n+1)$th stream active and look ahead to its first element. While this method expands the streams with the biggest contribution, it might not be guaranteed to take all elements of all streams. We continue to search for a better approach. We have implemented both the triangle method and this method. Both work but we do not have definitive results.

6. CONCLUSION

We have introduced the idea of using RL as a general approach to choosing between inference algorithms. We have shown how reinforcement learners can be combined in a network. Much work still needs to be done, however. While we have articulated the general principle, with reinforcement learning the devil is often in the details. While we have found different configurations to work for different problems, we haven't yet found a general configuration that works without user tweaking. This is a

vital step if we are to use RL in the inner loop of probabilistic inference.

REFERENCES

[1] A. Lu, "Venture: an extensible platform for probabilistic meta-programming," MIT Master's Thesis, 2016.

[2] A. Pfeffer, B. Ruttenberg, and W. Kretschmer, "Structured Factored Inference: A Framework for Automated Reasoning in Probabilistic Programming Languages," *ArXiv160603298 Cs*, Jun. 2016.

[3] H. H. Hoos, "Programming by optimization," *Commun. ACM*, vol. 55, no. 2, pp. 70–80, 2012.

[4] M. Tokic and G. Palm, "Value-difference based exploration: adaptive control between epsilon-greedy and softmax," *KI 2011 Adv. Artif. Intell.*, pp. 335–346,