Learning Stochastic Inverses

Jacob Taylor, Ashwin Siripurapu, Andreas Stuhlmüller

December 14, 2012

1 Abstract

For this project, we worked with Andreas Stuhlmüller to develop and implement a new inference algorithm for the Church programming language. Church is a programming language that extends Scheme with random primitives. For more details, see the original Church paper [1]

Andreas developed the basic algorithm and we did most of the implementation work, also working with Andreas to extend the algorithm.

2 Execution Traces

In Church, the evaluation of an expression produces an execution trace. Intuitively, the execution trace is the full tree of evaluations that was necessary to evaluate the expression. An execution trace consists of the following:

- The expression that was evaluated
- The environment (mapping of variables to values) it was evaluated in
- The return value
- A list of sub-traces

The sub-traces item requires explanation. A typical Scheme/Church interpreter will have a recursive `eval` function. For example, to evaluate an application like `(+ 1 2)`, it is first necessary to evaluate the expression `+`, then the expression `1`, then the expression `2`. All of these evaluations have their own execution traces, which are included in the execution trace for the expression `(+ 1 2)`.

In addition to the function and the argument, sometimes additional expressions are evaluated during the evaluation of an application expression. For example, to evaluate the expression `(lambda (x) (* x 2)) 1)`, it is first necessary to evaluate `(lambda (x) (* x 2))`, then `1`, then `(* x 2)` in the environment `{'x':1]}`. Therefore, the sub-traces for execution trace for `(lambda (x) (* x 2)) 1)` will consist of the execution traces for these 3 expressions.

3 Sampling Traces

Our inference engine samples an execution trace, given an expression, its environment, and its return value. Intuitively, it "imagines" a sequence of evaluations that would have resulted in the provided return value. For example, if the expression is `(+ (gaussian) (gaussian))`, and the return value is `1.32`, then in the sampled execution trace, perhaps the first `(gaussian)` expression returned `0.96` and the second returned `0.36`.

Ideally, the sampling distribution $Q(trace|result)$ would exactly match the posterior $P(trace|result)$. However, due to our approximations, they will not be equal. Therefore, we attach an importance weight to each sample. The importance weight is equal to $\frac{P(trace, result)}{Q(trace|result)}$. For an ideal $Q = P$, this weight is always equal to $P(result)$. Therefore, if the variance of the importance weights approaches 0, this is a sign that Q is approximately correct.

Q can either be used as an independent Metropolis Hastings MCMC proposal distribution, or as an importance sampling distribution.

4 Learning the Inverse

To create Q, we train separate learners for each application expression (a parenthesized Church expression in which a function is applied to arguments). This learner will take as

input the return value of the expression and the values of any free variables used in the expression. As output, it produces a joint distribution over the non-constant items in the application, possibly including both the function and the function arguments. For example, consider the expression `(+ (gamma) (* (poisson) (gaussian)))`. Let us label the expression as `(+ (gamma)=A (* (poisson)=B (gaussian)=C)=D)=E`. We would train 2 learners:

1. $A, D|E$
2. $B, C|D$

To provide training data for these learners, we evaluate the top-level expression and collect execution traces many times. We will get many A, B, C, D, E tuples. Learner 1 will learn using (A, D, E) tuples, and learner 2 will learn using (B, C, D) tuples.

After training, we can sample an execution trace. Pseudocode that implements this sampling algorithm is shown in Algorithm 1.

Suppose we knew that $E = 5$. Then we use learner 1 to sample from the joint distribution $A, D|E$. Then we feed the sampled value of D into learner 2, producing a sample from the joint distribution $B, C|D$. It is easy to verify that, if the learners learn the correct distributions, then the distribution over execution traces sampled using this mechanism is equal to the true distribution $P(A, B, C, D|E)$.

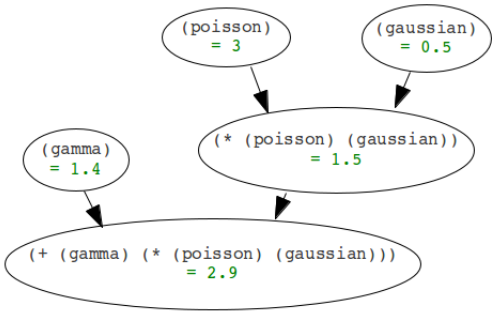

Figure 1: A sample execution trace. It can also be interpreted as a Bayesian network

For the actual learners, we use a k–nearest neighbors algorithm. If we want to learn the conditional distribution $Y|X$, first we get the k nearest (x, y) pairs to our sample x. Next, we find the distribution produced by selecting a random y value and "wiggling" any real numbers contained in it by a Gaussian distribution.

Throughout inference, we keep running counts of Q and P (in log space). Partway through inference, Q will be the probability of the random decisions that were made through sampling from the learned distribution. P will be the probability that applications of ERPs[1] whose arguments have been decided produce the results they do.

Both x and y may be structured Scheme objects, not only numbers. We decompose each object into its structure (everything about the object except real numbers contained in it) and its vector of real numbers.

5 Applications

We applied the system to a few problems to determine its strengths and weaknesses.

As a simple example, we applied the system to inferring object properties from luminance. We defined the reflectance to be a Gaussian variable and the illumination to be a sum of a gamma and Gaussian. The luminance is defined to be a product of reflectance and illumination, plus some noise. Inference proceeds in a standard way: use KNN to infer reflectance and illumination from the luminance, then infer the 2 variables making up reflectance. After 500 training runs, the standard deviation of log importance weights of sampled execution traces is 1.548, which is reasonable. We seem to be running up against the limitations of the Gaussian kernel in representing the true posterior.

Another application that we tried was a simple hidden Markov model, taken directly from the Wikipedia page on the topic [2]. Our Church code is reproduced below:

```
(define (sequence-from state n)
  (if (= n 1)
      (list (observe state))
      (cons (observe state)
        (sequence-from (transition state)
                       (- n 1)))))
```

[1] An *elementary random primitive*, or ERP, is a basic random function that can be used to construct other random functions. Examples in Church are `gaussian`, `gamma`, and `flip`. The requirements for an ERP are that it is possible to sample a value given the parameters, and also compute the probability of a value given parameters

Algorithm 1 Algorithm to sample an execution trace

$P \leftarrow 1$ ▷ The running probability of the trace according to the program.
$Q \leftarrow 1$ ▷ The running probability of the trace according to the sampler.
$Learners \leftarrow \{\}$ ▷ a mapping from expression to its learner
Run the program forwards multiple times to update $Learners$.
function SAMPLEEXECUTIONTRACE($expr, env, result$)
  if $expr$ is deterministic **then**
    $trace \leftarrow EvalAndGetTrace(expr, env)$
    Verify that $trace.result = result$
    return trace
  else[$expr$ is an application]
    $distr \leftarrow Learners[expr](result, env[GetFreeVariables(expr)])$
    ▷ In reality, only the non-deterministic items need to be sampled.
    $items \leftarrow Sample(distr)$
    $Q \leftarrow Q \cdot GetProbability(items, distr)$
    $subtraces \leftarrow []$
    for $i \leftarrow 0 \ldots Length(items) - 1$ **do**
      $subtrace \leftarrow SampleExecutionTrace(expr.subexpressions[i], env, items[i])$
      Append $subtrace$ to $subtraces$
    end for
    $function \leftarrow items[0]$
    $arguments \leftarrow items[1 \ldots Length(items) - 1]$
    if $function$ is a lambda function **then**
      $inner \leftarrow CreateLambdaEnvironment(function, arguments)$
      $subtrace \leftarrow SampleExecutionTrace(function.body, inner, result)$
      Append $subtrace$ to $subtraces$
    else if $function$ is an ERP **then**
      $P \leftarrow P \cdot ERPProbability(function, arguments, result)$
    else[$function$ is a deterministic primitive]
      Verify that $function(arguments) = result$
    end if
    return $MakeExecutionTrace(expr, env, result, subtraces)$
  end if
end function

```
(define (get-sequence n)
  (sequence-from (initial-state) n))
```

The `get-sequence` function samples an initial state (‘‘`rainy`’’ or ‘‘`sunny`’’) from a given distribution (not shown), and then generates a sequence of observations from that initial state by iteratively generating an observation and transitioning to a new state using a transition probability distribution (also not shown) `n` times.

Our learner now uses the Church inference algorithm to infer a likely sequence of underlying states for some observed output. In contrast to more traditional methods of HMM inference, such as the Viterbi algorithm, our program freely uses the structure of the program which encodes the HMM to generate multiple traces of the model, which are then used as training data. Moreover, whereas the Viterbi algorithm gives only a single most likely state sequence, our inference algorithm generates an entire probability distribution over state sequences, conditioned on the observation sequence that we want to explain.

The training points we compute are the output sequences of observations generated by the various runs of the program, and for each point, the training label is the underlying state sequence that was generated in the corresponding run. Now the inference algorithm can solve the machine learning problem of predicting a state sequence for an observation sequence given some training data using the techniques detailed above. Some generated training data is shown below:

Thirdly, we applied the system to perform parsing for a probabilistic context-free grammar (PCFG). Here is an excerpt of the model, showing the different productions for noun phrases:

```
(define (np)
  (if (flip 0.5)
    (list (noun))
    (if (flip 0.5)
      (cons (adjective) (np))
      (append (np) (cons ‘‘that’’ (vp))))))
```

The relevant learning problems are to determine which production a phrase came from , and to determine how appended phrases are split into sub-phrases. The first problem arises from the `(if (flip ...) ...)` calls; the second arises from the `append` calls. The `cons` and `list` functions do not require inference because they are fully invertible.

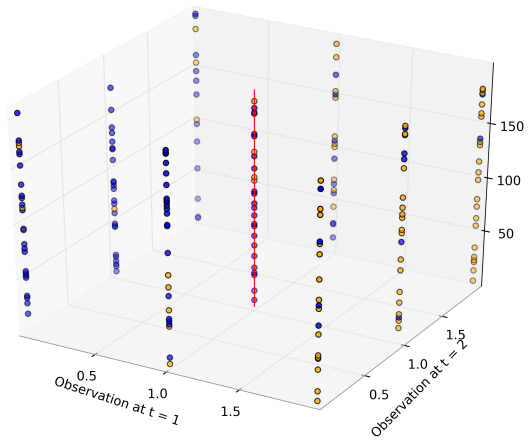

Figure 2: Example of training data for the HMM application. Points labeled ‘‘`sunny`’’ are colored orange, while points labeled ‘‘`rainy`’’ are colored blue. The red line depicts the observation sequence we want to explain.

The inference looks very similar to that of CYK parsing, except that, rather than consider every possible parse, it guesses the correct parse based on machine learning. The system sometimes returns valid parses, but sometimes returns invalid parses. This is to be expected: sometimes a production will be selected that cannot produce the given phrase, or the phrase will be split such that one or both parts cannot be formed by their respective non-terminals. Usually, for short sentences, the parse will be the unique correct parse. For example, the system correctly parses “buffalo buffalo buffalo” as `(S (NP (N buffalo)) (VP (V buffalo) (NP (N buffalo))))`. Long sentences are unlikely to be parsed correctly; they often assign words to parts of speech they cannot come from, causing P to be 0.

6 Problems and Solutions

One problem with the algorithm, as presented, is that the inferred arguments of the function are not necessarily consistent with the return value of the function. For example, in the expression `(+ (gaussian) (gaussian))`, if the return value is 3, the algorithm could guess that the first argument is 1.2 and the second is 1.9. Obviously, a trace in

which `(+ 1.2 1.9)` evaluates to 3 is invalid.

Initially, we dealt with this problem by modifying the execution trace so that the return value of the expression is 3.1 instead. Once the return values of calls to elementary random primitives are decided, the consequences of these decisions are propagated forwards so that the overall trace is consistent.

Unfortunately, this approach makes it difficult to update the counts for P and Q. Once we "decide" what the arguments to a function are (and multiply Q accordingly), these arguments might themselves change through forward propagation. For example, in the expression `(gaussian (+ (beta) (gamma)))`, once the sum has been "decided" the sum might change if it turns out that the sum of `(beta)` and `(gamma)` does not match this "decided" sum. This means that it is possible to get the same execution trace through 2 different initial decisions of what the sum is. This makes our estimate of Q by repeated multiplication of probabilities of random decisions invalid.

Andreas is currently working on a way to handle the importance weight in this case. In the meantime, it would be good to ensure that the proposed arguments always match the return value. So far, this has not been difficult. It requires the notion of a *pivot value*. A pivot value is a value that, along with the return value, can be used to recover the arguments. For example, for the `append` function, the pivot value is the length of the first argument, or the "splitting point". If we know that `(append x y) = (1 2 3)` and `(length x) = 2`, then we can recover the arguments: `x = (1 2)` and `y = (3)`. We can have the learner predict the pivot value (splitting point) rather than directly predicting the values of `x` and `y`. A pivot value is also possible for addition, subtraction, multiplication, and division; in all cases, the pivot is the first argument, and the second argument can be recovered as a function of the first argument and the result. Different logic applies when one or more of the arguments is a deterministic expression, but it is straightforward.

A second problem is that, sometimes, the queried result is not a possible return value of the expression. For example, perhaps it was earlier assumed that the expression `(list (get-word))` returned `(''a b'')`. In this case it is not possible to assign any sensible values to the `(get-word)` argument, because `(get-word)` returns a single word and the queried result consists of two words. As a partial resolution, we re–evaluated the expression and changed the result to equal the result of that evaluation. This does not resolve the importance weight correctly, so it is always best to avoid this situation entirely.

A final problem, which appears very difficult to resolve, is that the performance of the inference depends strongly on how the program is written. As an example, consider a few ways to encode an if expression, assuming we have a strict version of the `if` combinator, which we call `ifstrict`. One translation of `(if <cond> <then> <else>)` is `(ifstrict <cond> (lambda () <then>) (lambda () <else>)`. For this expression, inference proceeds from first guessing which function, `(lambda () <then>)` or `(lambda () <else>)`, was called based on the return value; then guessing what the value of `<cond>` is based on which function was called. This is definitely not a good inference strategy, because it requires inferring functions (which are possibly closures). On the other hand, if the expression is translated as `((lambda (c) ((ifstrict c ( lambda () <then>) (lambda () <else> )))) <cond>)`, then inference proceeds by predicting the value of the condition based on the return value. This inference strategy is superior because it only requires predicting a single boolean value.

The dependence of the inference on how the program is written leads to problems with practical applications. While the system can sample an execution trace, implementing Church's query operation is difficult because it is hard to specify exactly how to extract the queried value from the execution trace. If the expression for the observation can be written as `(observe (hypothesis))`, then it is easy to extract the hypothesis from the execution trace; however, the program will ordinarily not be written this way, because this might prevent efficient inference. Specifying the program in a way so that inference is efficient and it is also easy to extract the hypothesis from the execution trace is difficult.

References

[1] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum *Church: a language for generative models.* `http://www.stanford.edu/~ngoodman/papers/churchUAI08_rev2.pdf`

[2] Wikipedia, *Hidden Markov model.* `http://en.wikipedia.org/wiki/Hidden_Markov_model`