1 CONSTRAINED INFERENCE - ZENNA TAVARES

We address the problem of making good proposals in probabilstic inference. In particular we propose a formalism for constructively incorporating constraints into generative models, and suggest two strategies of implementation.

1.1 INTRODUCTION

Probabilstic inference has established itself as a means of reasoning in domains subject to uncertainty. Although the rules of conditional probability state how to update our beliefs in hypotheses conditioned on evidence, it tells us only declaratively, and leaves us with no guidance as to how hypotheses should be constructed in the first place. In anticipation of the distinction emphasised in this essay, we describe conditional probability and in particular Bayes' Rule as the *logic* of a program, while the the *control* or *procedure* remains a topic of continued research.

Inference typically refers to finding the expectation of some function with respect to a probability distribution, computing the posterior distribution, or drawing samples to approximate it. Numerous approaches exist, but our focus is on Monte Carlo methods whose strengths lie in their applicability to high dimensional distributions, where direct sampling becomes intractable. Inference by means of random perturbations to hypotheses is often inefficient however; we would prefer to not waste resources proposing candidate hypotheses which violate sensible constraints. In other words, we would like to make good guesses.

The objective of this study is to explore conceptual and practical methods of making good guesses by incorporating constraints *constructively* into generative models. We frame construction in constrast to testing; to construct is (ideally) to not test. We build upon recent interpretations of probabilstic inference in computational terms, in particular the QUERY operation, which performs universal inference in the probabilstic programming language Church [1]. We suggest a new constructive perspective CONSTRAIN, which given a generative model manifested as a stochastic *prior program* P, and a conditioning predicate C aims to construct a new program P^* which samples only values which adhere to our condition.

Universal inference [1] is analagous to notions of universality in computation. Informally it refers to the ability to perform conditional simulation of any computable probabilstic generative process.

The feasibility of this aim is vulnerable to skepticism; how can we construct only constrained proposals without testing to see if they satisfy our constraints? It is unlikely this objective can be achieved in full generality, in some cases we have no alternative than to generate and test. Furthermore we shall see that relaxations of the above maxim may lead to more pragmatic implementation strategies; to construct is to test less perhaps, or to test only partially and incrementally. Precisely what distuiginshes one constraint from another in terms of the extent to which it can be implemented constructively, is a topic of great interest and speculated on in our conclusion. Yet, we can motivate the plausiibilty of constructive inference by appealling to common experience: consider the ease with one can compose a paragraph that rhymes, generate a polynomial expression, or draw an acyclic graph. Each of these examples entails a logical constraint which we satisfy without hinderance. While it is risky to draw conclusions about the computational hardness of a problem based on our subjective difficulty when solving it, each of the above examples has a known efficient algorithm.

First we will summarise QUERY, a computational theory of probabilstic inference. Then we formalise the key concept of this paper, CONSTRAIN, in terms of QUERY. Two approaches to implementing CONSTRAIN are then described in detail.

1.1.1 *Query*

QUERY formalises probabilistic inference as a *program* in a corresponding model of computation. Developed first as a primitive function in the probabilstic programming language Church [1], QUERY has been described in terms of probabilstic generalisations of the λ-calculus and Turing machine, both classical models of computation.

In its original formulation, QUERY is a higher order function which accepts as input a stochastic expression to be evaluated and a set of predicate conditions. In lisp (clojure) notation:

Notation: defn defines a function. The following term is the function name and terms inside square brackets [] are argument names. Ĺetís used to assign names to values, conceptually similar to assignment of values to variables in imperative languages.

```
(defn query [exp pred]          ; Define function
  (let ((val (eval exp))        ; sample from model, call it val
    (if (pred val)              ; If val satisfies conditions
        val                     ; Then.. return it
        (query exp pred)))))    ; Otherwise try again
```

In [3] a formulation in terms of a probalistic turing machine (PTM) is given. A Turing machine [4] is a mathematical abstraction of a machine, which may read, write and seek access on a finite collection of infinitely long binary tapes. Prior to execution, its input is loaded onto one or more of its tapes, and the output is the content of its tapes after the machine halts. A probabilistic Turing machine (PTM) is a Turing machine equipped with a tape consisting of a sequence of independent random bits, which is accessible to the Turing machine as a read only randomness source.

QUERY is a PTM which takes two inputs, a prior program P and a conditioning predicate C. Both P and C are themselves encodingings of PTMs that take no input. QUERY generates a sample from P. Then, if C is satisfied this sample is outputted, otherwise the process is repeated.

It should be of little surprise that both these formulations are equivalent, as they both perfom the function of drawing samples from a prior distribution conditioned on C, using rejection. While rejection sampling provides an simple and intuitive understanding of the meaning of QUERY, it is of course grossly inefficient for the majority of non-trivial problems.. Much research in inference is in looking for tractable approximations and alternatives.

1.1.2 *Constrain*

The semantics of CONSTRAIN can be readily understood in terms of QUERY. Expanding on the lisp definition of QUERY given above, CONSTRAIN is a higher order function expecting a prior program P, and taking *constraint* C^*. In the first of the approaches outlined in the next chapter cases C and C^* will be identical, while in the second the set of constraints is a subset of the set of conditions. They are differentiated in terms of hardness; constraints are logical and must be true, i.e., $P(x|\neg C^*) = 0, \forall x$. CONSTRAIN simply returns a function of no arguments which is QUERY with all its arguments evaluated.

```
(defn constrain [exp pred]         ; Define a function
  (fn [] (query expr pred)))       ; Return a 0-ary function
```

While QUERY returns a sample given a model and condition, CONSTRAIN returns a function of no arguments which calls QUERY. In deterministic programs, a function with all its arguments evaluated is simply a value which evaluates to itself, and little is typically gained from taking a functional perspective. In stochastic programs however the output of CONSTRAIN implictly defines a conditional distribution. This alone differs little from QUERY, as we have only described the semantics. We differentiate with the objective that that C^* is conditioned on constructively; we wish to refrain from applying it as a predicate to fully formed samples, i.e., testing, and instead exploit its semantics to find a more efficient means. The rest of this study is devoted into proposed means of doing this.

2 CONSTRAINED GENERATIVE MODELS

There are a number of ways to explore this problem, varying principally in dependence on domain specific knowledge. We will outline two methods, which draw largely from the domains of program semantics and automated reasoning about programs.

2.1 TRANSFORM REJECTIVE GENERATIVE MODEL

Transformational Programming is a prominent method used in automated program development. A formal, declarative specification of a program is *refined* into a complete program by applying a sequence of correctness-preserving transformations. We can appropriate this framework for our purposes; first constrain a naive generative through rejection sampling, then transform into a semantically equivalent, but more efficient program.

From a stochasic program P and constraint C, we construct a new program R_P^C with rejection sampling semantics. R_P^C executes P to sample from its prior and returns the sample C is satisfied, otherwise a further attempt is made. R_P^C is what is returned from application of CONSTRAIN to P and C, and can also be viewed as the partial-evaluation of the following lisp function:

Partial evaluation of a program means to take some subset of its arguments, and compile a new program with this subset fixed (under closure) and no longer arguments.

```
(defn R [P C]
  (let [sample (P)]
  (if (true? (C sample))
      sample
      (R P C))))
```

Our next objective is to perform a series of transformations to improve the efficiency R_P^C. By constraining this set of transformations to be semantic preserving - any new program is *equivalent* to the original program, our program will describe the same distribution. r It does not follow immediately that a transformed program will be constructive, this depends entirely on the transformations applied. The follow example illustrates one method, which may fall short of constructivism but could improve upon rejection sampling.

Consider a naive generative model which generates polygons by sampling points uniformly over some two dimensional interval. Clearly, the majority of generated polygons will not be convex, most will not even be simple. We wish our polygons to be convex and so specify this as some computable predicate C. R_P^C will construct a fully formed polygon, then reject it if it is not convex. One obvious transformation will result in a new program which applies the convexity test to partially constructed proposals, exploting the fact that if some part of a polygon is not convex, neither will the whole polygon ever be. Clearly the effiency gains will depend on the complexity of the test, the size of the polygons being generated and the frequency with which it is applied to partial solutions. Additionally this kind of transformation cannot be applied unconditionally; there instances of constraints which may fail partial solutions but permit whole solutions compsoed of these failing parts.

2.1.1 *Domain General and Specific Transformations*

Three well known program transformations are *finite differencing, partial evaluation and dominated convergence*. Partial evaluation [2] is perhaps the mot practical transformation commonly used, and forms the basis of CONSTRAIN. It is based on the idea in which a highly parameterised program is concretised by simplification , when some subet of the parameters are fixed. It uses a highly generic strategy which traces a portion of the computation for which expressions can be evaluated. As a result, partial evaluation must be implemented in full accordance with language semantics

Transformations such as these are deductive in the sense that every one has an associated proof that it is semantic preserving. These proofs are typically domain independent; they apply to any program and exploit the denotations, i.e., the meaning of the program syntax to make valid changes. Clearly however, many semantic preserving transformations will depend on domain specific proofs. A program transformation for the convex polygon example above could depend upon geometric proofs for instance. Automatically generating domain specific transformations then must defer to the problem of automated theorem proving. This is indeed an active area of research in program transformation.

2.2 ERROR CORRECTING EVALUATION

Constraining generative models without knowledge speific to the domain X seems implausible; how can one generate only convex polygons without any understanding of convexity or geometry? Yet the complexities of representing, incorporating and devising algorithms to exploit domain specific knowledge is best avoided. Black box optimisation methods succeed in obviating domain specific requirements, but at the cost of discarding useful information about the structure of the problem and causes of error in proposals. This approach posits that when considered in accordance to the semantics of program in which they are represented, P, C^* and x are sufficiently meaningful to sidestep any need to inject domain specific knowledge.

The constraint predicate C^* itself contains a wealth of information to aid in both transforming particular samples to force them to adhere to constraints, and transforming the generative model itself. The central idea is of *transparency*: to treat neither constraint C^* nor sample x as monolithic and impenetrable entities. They are viewed instead as structured objects, composed of parts which individually and in concert have a rigorously defined meaning. Failure to satisfy a constraint is not the fault indiscriminately of the whole object then, it can be *blamed* on subcomponents. This idea is easily explained by example, consider the following predicate on a real valued pair of values x_1 and x_2:

```
(defn C [x1 x2]
  (if (and (pos? x1) (neg? x2))   ; If x1 is +ve and x2 -ve
      true                        ; return true
      false))                     ; else return false
```

Given a sample $(x_1, x_2) = (2.0, 3.0)$ this predicate will return false. An understanding of the program will lead a reasoner to blame this failure on the fact that x_2 is not negative, and furthermore conclude that any change to x_2 to make it negative, such as $x_2 = -0.5$ will satisfy the constraint. Hence in order to fix this particular sample such that it satisfies the constraint, we have found a local constraint that needs to be satisfied, namely x_2 must be negative. We have attributed blame to part of the sample. We can extend this idea to the generative model:

```
(defn gen-pair []
  (list (rand 0 10) (rand -10 10)))
```

Now we can ask see what caused x_2 to be positive, from the generative model we can clearly see that the evaluation of (rand -10 10), which returns a uniform value between -10 and 10, is to blame. Tt is the cause unsatisfiability at the level of the generative model. In order to correct this, a local constraint can be applied on the second argument of list (the cause of the value of x_2). That is, a change to (rand -10 10) that will only evaluate to negative values is to be found.

2.2.1 *Blame attribution and counterfactual reasoning*

The concepts of *blame attribution*, *counter factual reasoning*, *local consistency* and *causal chaining* can be seen in the example, and are central to the proposed method. Informally, the approach can be stated as follows:

1. Evaluate the program until failure (if the constraint is satisfied then we are done), and attribute blame to parts of the program that caused the failure.

A cause of failure could be the evaluation of an expression or the execution of an imperative statement. There may be multiple causes, and independencies between them.

2. Consider counterfactuals.

   Alternative worlds where a particular cause of failure does not cause failure are then considered. Specifically we seek a local constraint on the cause closest to failure of the entire predicate, that describes the change required result in satisfiability.

3. Chain causes.

   A causal chain from the cause of failure back to our object of interest is made. This object is either the sample if our goal is to fix one particular sample, or the generative model itself.

4. Solve constraints The output of this algorithm is a set of local constraints. They are local in the sense they apply to specific parts of the program. We may then wish to solve these constraints to produce a fixed sample or generative model. Constraints will typically have numerous solutions, we may then wish to pick a single one, or in some other way describe a new generative model over the solutions of the constraints.

In this light, the approach can be viewed as a special kind of evaluation. The objective of this error correcting evaluation is not to return a value, but instead to correct either the sample or the generative model by fixing local problems. In order to do this we must evaluate both forwards to the point of failure, and backwards to infer the causes.

2.2.2 *Program Sematnics*

This method relies on understanding the meaning of a program, its denotational semantics. To illustrate let us consider a slight variation of the previous predicate:

```
(defn C2 [x1 x2]
  (if (foo (pos? x1) (neg? x2))
      true
      false))
```

We have substituted the *and* for *foo*, a function of unknown semantics. As a result, the program has become meaningless to us. We can no longer reason about its behaviour or infer what the cause of failure will be since foo could be any arbitrarily complex or simple function of two arguments. Neither then could we expect any automated reasoner to be able to perform this task.

Fortunately, programming languages are formal objects and their meaning can be specified exactly. Denotational semantics attempts to construct mathematical objects, denotations, that describe the meaning of the language. Semantics should be compositional; denotations of a program should be composed of denotations of its subcomponents.

Several practical langauges such as Haskell and ML have complete denotational semantics, and are good candidates for use by the proposed method.

2.2.3 *Formal specification of method*

Try to define our approach with respet to denotational semantics

2.2.4 *Practical proposal for constrained goal*

In order Let's focus on a small imperative language. Our domain X is composed of input vectors of booleans or real valued numbers. Our predicates and our generative models are non-recursive. no looping allowed.

2.2.5 *TO TALK ABOUT*

How this relates to dynamic execution

Concrete/symbolic evaluation

How this is a kind of constraint programming

3 CASE STUDY

3.0.6 *issues*

Order of constraints Discussion has been limited to a single constraint C, since multiple constraints can be found conjoiing each individual constraint together. There may be practical implications of the order. There may also be means to recompile the constraint into one which is easier to reason about.

Partial evaluation

Hardness 0

3.1 CONCLUSION

Here the probabilstic inference framework QUERY was given a constructive perspective with CONSTRAIN. We proposed two methods for implementing CONSTRAIN based on program transformations and local error inference. We suspect there will be many difficulties with such an implementation, but preliminary evidence suggests evencrude approaches to generating more constrained proposals can have dramatic effects on inference performance.

BIBLIOGRAPHY

[1] N Goodman, V Mansinghka, D Roy, K Bonawitz *Church: a language for generative models* 2012.

[2] RA Paige, O Danvy *Automatic program development: A tribute to Robert Paige* 2008:

[3] CE Freer, DM Roy, JB Tenenbaum *Towards common-sense reasoning via conditional simulation: legacies of Turing in Artificial Intelligence* 2012 Arxiv:

[4] A Turing *On computable numbers, with an application to the Entscheidungsproblem* 1936: Proceedings of the London mathematical society.