Probabilistic Programming

Frank Wood
fwood@robots.ox.ac.uk

NIPS TUTORIAL 2015
My Objectives

Get you to

• Know what probabilistic program is and how it’s different to a normal program.

• Roughly understand how to write a probabilistic program and have the resources to get started if you want to.

• Understand the literature at a very high level.

• Know one way to roll your own state-of-the-art universal probabilistic programming system.
What is probabilistic programming?
Intuition

Parameters → Program → Output

Parameters → Program → Observations

$p(x|y)$

$p(y|x)p(x)$

y

CS
Probabilistic Programming
Statistics
A Probabilistic Program

“Probabilistic programs are usual functional or imperative programs with two added constructs:

(1) the ability to draw values at random from distributions, and

(2) the ability to condition values of variables in a program via observations.”

Gordon, Henzinger, Nori, and Rajamani
Goals of the Field
Increase Programmer Productivity

Lines of Matlab/Java Code

Lines of Anglican Code

HPYP, [Teh 2006]

Object Tracking, [Neiswanger et al 2014]

Automata Induction [Pfau et al 2010]

Collapsed LDA

DP Conjugate Mixture
Commodify Inference

Models / Simulators

Language Representation / Abstraction Layer

Inference engines
New Kinds of Models

\[ p(x|y) = \frac{p(y|x)p(x)}{p(y)} \]

![Diagram showing the relationship between x and y](image)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>program source code</td>
<td>program output</td>
</tr>
<tr>
<td>scene description</td>
<td>image</td>
</tr>
<tr>
<td>policy and world</td>
<td>rewards</td>
</tr>
<tr>
<td>cognitive process</td>
<td>behavior</td>
</tr>
<tr>
<td>simulation</td>
<td>constraint</td>
</tr>
</tbody>
</table>
History
<table>
<thead>
<tr>
<th>Year</th>
<th>PL</th>
<th>AI</th>
<th>ML</th>
<th>STATS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>Figaro</td>
<td>HANSAI</td>
<td>probDL</td>
<td>webPPL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Haskell, Scheme,…</td>
</tr>
<tr>
<td>2000</td>
<td>IBAL</td>
<td></td>
<td>probDL</td>
<td>Venture, Probabilistic-C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Anglican</td>
</tr>
<tr>
<td>1990</td>
<td></td>
<td></td>
<td>probDL</td>
<td>LibBi</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STAN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JAGS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JAGS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WinBUGS</td>
</tr>
<tr>
<td>Simula</td>
<td>Prolog</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Success Stories

Graphical Models

BUGS

STAN

Factor Graphs

Factorie

Infer.NET
As an aside, note that if we were to do prediction we could simply specify another model on the value of observations and the Anglican (let \[ \{ \text{ys} \} \sim \text{dnorm}(\text{x}, 1/c) \]) for each node and attempt to pattern match it to an “e cient” Gibbs operator for such transformations of complex models to graphical models in which computationally efficient inference can be performed, but we digress. More on this later.

To start, every variable name to the left of a \( \sim \) denotes a conditional distribution, in this case of \( x \) given \( c \):

\[
\text{model} \{ \\
\quad x \sim \text{dnorm}(a, 1/b) \\
\quad \text{for } (i \text{ in } 1:N) \{ \\
\qquad y[i] \sim \text{dnorm}(x, 1/c) \\
\quad \} \\
\}
\]

- Language restrictions
  - Bounded loops
  - No branching
- Model class
  - Finite graphical models
- Inference - sampling
  - Gibbs

STAN: Finite Dimensional Differentiable Distributions

parameters {
    real xs[T];
}

model {
    xs[1] ~ normal(0.0, 1.0);
    for (t in 2:T)
        xs[t] ~ normal(a * xs[t - 1], q);
    for (t in 1:T)
        ys[t] ~ normal(xs[t], 1.0);
}

• Language restrictions
  • Bounded loops
  • No discrete random variables*
• Model class
  • Finite dimensional differentiable distributions
• Inference - sampling
  • Hamiltonian Monte Carlo
    • Reverse-mode automatic differentiation
  • Black box variational inference, etc.

Factorie and Infer.NET

- Language restrictions
  - Finite compositions of factors
- Model class
  - Finite factor graphs
- Inference - message passing, etc.

Inference - message passing

\[ p_{\pi_3} = \mathcal{N}(\mu_3; \sigma^2_3) \]

Finite factor graphs

\[ \text{Finite factor graphs} \]

Inference - message passing, etc.

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

Finite factor graphs

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]

\[ \text{Finite compositions of factors} \]

\[ \text{Model class} \]

\[ \text{Language restrictions} \]

\[ \text{Inference - message passing, etc.} \]
Modeling language desiderata

- Unrestricted
  - “Open-universe” / infinite dim. parameter spaces
  - Mixed variable types

- Unfettered access to existing libraries

- Easily extensible

- Will come at a cost
  - Inference is going to be harder
  - More ways to shoot yourself in the foot
Languages and Systems

PL | AI | ML | STATS
---|----|----|------
2010 | Figaro | ProbLog | webPPL
2000 | HANSAI | KMP | Probabilistic-C
1990 | IBAL | Blog | Probabilistic-ML, Haskell, Scheme, …

Discrete RV’s Only

Simula | Prolog

Bounded Recursion
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>program source code</td>
<td>program output</td>
</tr>
<tr>
<td>scene description</td>
<td>image</td>
</tr>
<tr>
<td>policy and world</td>
<td>rewards</td>
</tr>
<tr>
<td>cognitive process</td>
<td>behavior</td>
</tr>
<tr>
<td>simulation</td>
<td>constraint</td>
</tr>
</tbody>
</table>
distractors. Each source of text violates the underlying assumptions of our probabilistic graphics.

We experimented with enabling enumerative (griddy) Gibbs sampling for uniform discrete variables.  

Letter Gaussian spatial blur bandwidths.

The stochastic likelihood model is a multivariate Gaussian whose mean is the blurry rendering; formally, before applying the stochastic likelihood model on the blurred original and rendered images. The renderer rasterizes each letter independently, applies a spatial blur to each image, composites the latent scene.

We developed a probabilistic graphics program for reading short snippets of degraded text consisting of a bank of variables for each glyph, including whether a potential set to favor small bandwidths). To make hard classification decisions, we use the sample.

The main text for quantitative details, and supplemental material for the full corpus.

Figure 2: Four input images from our CAPTCHA corpus, along with the final results and conver-

Figure 3: A probabilistic graphics program for scene perception. CVPR (2015).

Kulkarni, Kohli, Tenenbaum, Mansinghka

We now turn to our final experiment: controlling a simulated snake robot in a maze. This domain combines elements of navigation—except that low-level actions are now continuous—with an additional constraint: the robot must follow a set path to find the goal. This can be viewed as a form of intra-task transfer, where the agent learns to navigate the maze while being guided by a shaping reward.

Figure 2: Results on the memory maze. Top left: with a shaping reward throughout the entire maze, using options slightly degrades performance. Top right: when the shaping reward is removed, the options-based policy is still able to find the goal, achieving a higher return. The explanation is shown in the lower-right pane, where the option-based policy reliably reaches the goal, while the other policy stagnates.

For every combination of internal state and external observation, we choose a successor internal state from a Dirichlet-Multinomial. Given the size of the search space, for our first experiment, we use a hierarchical Dirichlet Process prior. For every combination of internal states, we choose a successor internal state from a Dirichlet-Multinomial.

The state of the internal, mental actions (denoted by \(e\)) and observations (denoted by \(a\)) are severely aliased, reflecting only the presence or absence of the corresponding motor primitives. However, we do not know how long the motor primitives should be, or how many there are, or which ones should guide the navigation. A similar story could be told about two similar domains, finding the goal. This can be viewed as a form of intra-task transfer, where the agent learns to navigate the maze while being guided by a shaping reward.

To simultaneously capture state and a policy, we use a finite-state controller (FSC). Each state in the FSC represents a policy over the integers \(\mathbb{Z}\), where \(GEM\) is the standard stick breaking construction of a generative process. To allow for a potentially unbounded number of internal states, but to favor a small number of states a priori, we choose a posterior internal state from a Dirichlet process. For ease of exposition, we will consider these to be deterministic actions from a compound Dirichlet-Multinomial.
Reasoning about reasoning

Want to meet up but phones are dead…

I prefer the pub.
Where will Noah go?
Simulate Noah:
Noah prefers pub
but will go wherever Andreas is
Simulate Noah simulating Andreas:
…
-> both go to pub

Stuhlmüller, and Goodman.
"Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs."
Program Induction

\[ \hat{y} \sim p(\cdot | x) \]

\[ x \sim p(x) \]

\[ y \sim p(y) \]

---

Perov and Wood.
"Learning Probabilistic Programs."
Constrained Stochastic Simulation

Stable Static Structures

Procedural Graphics

---


Universal
Probabilistic Programming
Modeling Language
Introduction to Anglican/Church/Venture/WebPPL…
A Language Family Tree

- **Church**
- **WebChurch (Bher)**
- **WebPPL**
  - **lisp javascript**
  - **clojure**
  - **c**

**Interpreted** languages:
- **Anglican**
- **VentureScript**

**Compiled** languages:
- **Inspiration**
  - Modeling language
Syntax: Anglican \approx Clojure \approx Church \approx Lisp

- Notation: Prefix vs. infix

```clojure
;; Add two numbers
(+ 1 1)

;; Subtract: "10 \(-\) 3"
(- 10 3)

;; (10 \(*\) (2.1 + 4.3) \(/\) 2)
(/ (* 10 (+ 2.1 4.3)) 2)
```
• Notation: Prefix vs. infix

;;; Add two numbers
(+ 1 1)

;;; Subtract: "10 - 3"
(- 10 3)

;;; (10 * (2.1 + 4.3) / 2)
(/ (* 10 (+ 2.1 4.3)) 2)

• Branching

;;; outputs 4
(+ (if (< 4 5) 1 2) 3)
Functions

• Functions are first class

;; evaluates to 32
((fn [x y] (+ (* x 3) y))
  10
  2)
Functions

• Functions are first class

;; evaluates to 32
((fn [x y] (+ (* x 3) y))
  10
  2)

• Local bindings

;; let is syntactic "sugar" for the same
(let [x 10
      y 2]
  (+ (* x 3) y))
Higher-Order

• map

```clojure
(map (fn [x y] (+ x (* 2 y)))
     [1 2 3] ; these are values x1, x2, x3
     [10 9 8]) ; these are values y1, y2, y3
```

• reduce

```clojure
(reduce + 0 [1 2 3 4])
```

;; does (+ (+ (+ 0 1) 2) ... ;; and evaluates to 10
(defquery gaussian-model [data]
  (let [x (sample (normal 1 (sqrt 5)))
        sigma (sqrt 2)]
    (map (fn [y] (observe (normal x sigma) y)) data)
    (predict :x x)))

(x ~ Normal(1, \sqrt{5})
  y_i|x ~ Normal(x, \sqrt{2})

(def dataset [9 8])

(def posterior
  ((conditional gaussian-model
      :pgibbs
      :number-of-particles 1000) dataset))

(y_1 = 9, y_2 = 8
  x|y \sim Normal(7.25, 0.91)

(def posterior-samples
  (repeatedly 20000 #(sample posterior)))
Graphical Model

(defquery gaussian-model [data]
  (let [x (sample (normal 1 (sqrt 5)))
        sigma (sqrt 2)]
    (map (fn [y] (observe (normal x sigma) y)) data)
    (predict :x x)))

(def dataset [9 8])

(def posterior
  ((conditional gaussian-model
     :pgibbs
     :number-of-particles 1000) dataset))

(def posterior-samples
  (repeatedly 20000 #(sample posterior)))

\[ x \sim \text{Normal}(1, \sqrt{5}) \]
\[ y_i | x \sim \text{Normal}(x, \sqrt{2}) \]

\[ y_1 = 9, \ y_2 = 8 \]

\[ x | y \sim \text{Normal}(7.25, 0.91) \]
Graphical Model

(defquery gaussian-model [data]
  (let [x (sample (normal 1 (sqrt 5)))
        sigma (sqrt 2)]
    (map (fn [y] (observe (normal x sigma) y)) data)
    (predict :x x)))

(def dataset [9 8])

(def posterior
  ((conditional gaussian-model
      :pgibbs
      :number-of-particles 1000) dataset))

(def posterior-samples
  (repeatedly 20000 #(sample posterior)))

\[ x \sim \text{Normal}(1, \sqrt{5}) \]
\[ y_i | x \sim \text{Normal}(x, \sqrt{2}) \]
\[ y_1 = 9, y_2 = 8 \]
\[ x | y \sim \text{Normal}(7.25, 0.91) \]
(defquery gaussian-model [data]
  (let [x (sample (normal 1 (sqrt 5)))
       sigma (sqrt 2)]
    (map (fn [y] (observe (normal x sigma) y)) data)
    (predict :x x)))

(def dataset [9 8])

(def posterior
  ((conditional gaussian-model
      :pgibbs
      :number-of-particles 1000) dataset))

(def posterior-samples
  (repeatedly 20000 #(sample posterior)))
Anglican: Syntax ≈ Clojure, Semantics ≠ Clojure

```clojure
(defquery gaussian-model [data]
  (let [x (sample (normal 1 (sqrt 5)))
        sigma (sqrt 2)]
    (map (fn [y] (observe (normal x sigma) y)) data)
    (predict :x x)))
```

```clojure
(def dataset [9 8])
(def posterior ((conditional gaussian-model
                   :pgibbs :number-of-particles 1000) dataset))
```

```clojure
(def posterior-samples
  (repeatedly 20000 #(sample posterior)))
```

$x \sim \text{Normal}(1, \sqrt{5})$

$y_i | x \sim \text{Normal}(x, \sqrt{2})$

$y_1 = 9, y_2 = 8$

$x | y \sim \text{Normal}(7.25, 0.91)$
Bayes Net

(defquery sprinkler-bayes-net
  (let [cloudy (sample (flip 0.5))
        raining (sample (if cloudy
                         (flip 0.8)
                         (flip 0.2)))
        sprinkler-dist (if cloudy
                         (flip 0.1)
                         (flip 0.5))
        sprinkler true
        _ (observe sprinkler-dist sprinkler)
        wetgrass-dist (cond (and (= sprinkler true)
                              (= raining true))
                        (flip 0.99)
                        (and (= sprinkler false)
                             (= raining false))
                        (flip 0.00)
                        (or (= sprinkler true)
                            (= raining true))
                        (flip 0.90))
        wetgrass true
        _ (observe wetgrass-dist wetgrass)]

  (predict :cloudy cloudy)
  (predict :raining raining)
  (predict :wetgrass wetgrass)))
(defquery hmm
  (let [init-dist (discrete [1 1 1])
        trans-dist (fn [s]
                      (cond
                        (= s 0) (discrete [0 1 1])
                        (= s 1) (discrete [0 0 1])
                        (= s 2) (dirac 2)))
        obs-dist (fn [s] (normal s 1))
        y-1 1
        y-2 1
        x-0 (sample init-dist)
        x-1 (sample (trans-dist x-0))
        x-2 (sample (trans-dist x-1))]
    (observe (obs-dist x-1) y-1)
    (observe (obs-dist x-2) y-2)
    (predict :x-0 x-0)
    (predict :x-1 x-1)
    (predict :x-2 x-2)))
(defquery hmm
  [ys init-dist trans-dists obs-dists]
  (predict
    :x
    (reduce
      (fn [xs y]
        (let [x (sample (get trans-dists (peek xs)))]
          (observe (get obs-dists x) y)
          (conj xs x))
        [(sample init-dist)]
        ys)))

\[\begin{align*}
x_0 & \rightarrow x_1 \\
x_1 & \rightarrow x_2 \\
x_2 & \rightarrow x_3 \\
\end{align*}\]
\[\begin{align*}
y_1 & \rightarrow y_2 \\
y_2 & \rightarrow y_3 \\
\end{align*}\]
An Unbounded Recursion

(defquery geometric [p]
"geometric distribution"
(let [dist (flip p)
    samp (loop [n 0]
      (if (sample dist)
        n
        (recur (+ n 1))))]
  (predict :x samp)))
Deterministic Simulation

```
(defquery arrange-bumpers []
  (let [bumper-positions []]
    ;; code to simulate the world
    world (create-world bumper-positions)
    end-world (simulate-world world)
    balls (:balls end-world)
    ;; how many balls entered the box?
    num-balls-in-box (balls-in-box end-world)]

  (predict :balls balls)
  (predict :bumper-positions bumper-positions)))
```

goal: “world” that puts ~20% of balls in box…
(defquery arrange-bumpers []
  (let [number-of-bumpers (sample (poisson 20))
        bumpydist (uniform-continuous 0 10)
        bumpxdist (uniform-continuous -5 14)
        bumper-positions (repeatedly
                          number-of-bumpers
                          #(vector (sample bumpxdist)
                                   (sample bumpydist)))]

;; code to simulate the world
world (create-world bumper-positions)
end-world (simulate-world world)
balls (:balls end-world)

;; how many balls entered the box?
num-balls-in-box (balls-in-box end-world)]

(predict :balls balls)
(predict :bumper-positions bumper-positions)))
Constrained Stochastic Simulation

(defquery arrange-bumpers []
(let [number-of-bumpers (sample (poisson 20))
  bumpydist (uniform-continuous 0 10)
  bumpxdist (uniform-continuous -5 14)
  bumper-positions (repeatedly
    number-of-bumpers
    #(vector (sample bumpxdist)
               (sample bumpydist)))
  world (create-world bumper-positions)
  end-world (simulate-world world)
  balls (:balls end-world)
  num-balls-in-box (balls-in-box end-world)
  obs-dist (normal 4 0.1)]
  (observe obs-dist num-balls-in-box)
  (predict :balls balls)
  (predict :bumper-positions bumper-positions)))
A Hard Inference Problem

(defquery md5-inverse [L md5str]
  "conditional distribution of strings
  that map to the same MD5 hashed string"
  (let [mesg (sample (string-generative-model L))]
    (observe (dirac md5str) (md5 mesg))
    (predict :message mesg))))
An Inference Framework For Universal Probabilistic Programming Languages
The Gist

- Explore as many “traces” as possible, intelligently
  - Each trace contains all random choices made during the execution of a generative model
- Compute trace “goodness” (probability) as side-effect
- Combine weighted traces probabilistically coherently
- Report projection of posterior over traces

If it's going to be "hard," let's at least make it fast

First generation - interpreted
Second generation - compiled
(let [t-1 3
      x-1 (sample (discrete (repeat t-1 1)))]
  (if (not= x-1 1)
      (let [t-2 (+ x-1 7)
             x-2 (sample (poisson t-2))]))
Goodness of Trace

(let [t-1 3
  x-1 (sample (discrete (repeat t-1 1)))]
(if (not= x-1 1)
  (let [t-2 (+ x-1 7)
    x-2 (sample (poisson t-2))]
    (observe (gaussian x-2 0.0001) 1))))
Trace

• Sequence of $N$ **observe**’s

$$\{(g_i, \phi_i, y_i)\}_{i=1}^N$$

• Sequence of $M$ **sample**’s

$$\{(f_j, \theta_j)\}_{j=1}^M$$

• Sequence of $M$ sampled values

$$\{x_j\}_{j=1}^M$$

• Conditioned on these sampled values the entire computation is *deterministic*
Trace Probability

• Defined as (up to a normalization constant)

\[\gamma(x) \triangleq p(x, y) = \prod_{i=1}^{N} g_i(y_i|\phi_i) \prod_{j=1}^{M} f_j(x_j|\theta_j)\]

• Hides true dependency structure

\[\gamma(x) = p(x, y) = \prod_{i=1}^{N} \tilde{g}_i(x_{n_i}) \left(y_i \left| \tilde{\phi}_i(x_{n_i}) \right. \right) \prod_{j=1}^{M} \tilde{f}_j(x_{j-1}) \left(x_j \left| \tilde{\theta}_j(x_{j-1}) \right. \right)\]
Inference Goal

• Posterior over traces

\[ \pi(x) \triangleq p(x|y) = \frac{\gamma(x)}{Z} \quad \text{and} \quad Z = p(y) = \int \gamma(x)dx \]

• Output

\[ \mathbb{E}[z] = \mathbb{E}[Q(x)] = \int Q(x)\pi(x)dx = \frac{1}{Z} \int Q(x)\frac{\gamma(x)}{q(x)}q(x)dx \]
Three Base Algorithms

- Likelihood Weighting
- Sequential Monte Carlo
- Metropolis Hastings
Likelihood Weighting

• Run $K$ independent copies of program simulating from the prior

$$q(x^k) = \prod_{j=1}^{M^k} f_j(x^k_j | \theta^k_j)$$

• Accumulate unnormalized weights (likelihoods)

$$w(x^k) = \frac{\gamma(x^k)}{q(x^k)} = \prod_{i=1}^{N^k} g_i^k(y_i^k | \phi_i^k)$$

• Use in approximate (Monte Carlo) integration

$$W^k = \frac{w(x^k)}{\sum_{\ell=1}^{K} w(x^\ell)}$$

$$\hat{E}_\pi[Q(x)] = \sum_{k=1}^{K} W^k Q(x^k)$$
Likelihood Weighting Schematic

\[ z^1, w^1 \]

\[ z^2, w^2 \]

\[ \vdots \]

\[ z^K, w^K \]
Sequential Monte Carlo

- Notation
  \[ \tilde{x}_{1:n} = \tilde{x}_1 \times \cdots \times \tilde{x}_n \]

- Incrementalized joint
  \[ \gamma_n(\tilde{x}_{1:n}) = \prod_{n=1}^{N} g(y_n|\tilde{x}_{1:n})p(\tilde{x}_n|\tilde{x}_{1:n-1}) \]

- Incrementalized target
  \[ \pi_n(\tilde{x}_{1:n}) = \frac{1}{Z_n} \gamma_n(\tilde{x}_{1:n}) \]
SMC for Probabilistic Programming

Want samples from

$$\pi_n(\tilde{x}_{1:n}) \propto p(y_n|\tilde{x}_{1:n})p(\tilde{x}_n|\tilde{x}_{1:n-1})\pi_{n-1}(\tilde{x}_{1:n-1})$$

Have a sample-based approximation to

$$\hat{\pi}_{n-1}(\tilde{x}_{1:n-1}) \triangleq \sum_{k=1}^{K} W^k_{n-1} \delta^{k}_{\tilde{x}_{1:n-1}}(\tilde{x}_{1:n-1})$$

Sample from

$$\tilde{x}_{1:n-1}^{a_{n-1}} \sim \hat{\pi}_{n-1}(\tilde{x}_{1:n-1})$$
$$\tilde{x}_{n}\tilde{x}_{1:n-1}^{a_{n-1}} \sim p(\tilde{x}_{n}|\tilde{x}_{1:n-1}^{a_{n-1}})$$

$$\tilde{x}_{1:n} = \tilde{x}_{1:n-1}^{a_{n-1}} \times \tilde{x}_{n}$$

Importance weight by

$$w(\tilde{x}_{1:n}^k) = p(y_n|\tilde{x}_{1:n}^k) = g_n^k(y_n|\tilde{x}_{1:n}^k)$$

$$W^k_n \triangleq \frac{w(\tilde{x}_{1:n}^k)}{\sum_{k'=1}^{K} w(\tilde{x}_{1:n}^{k'})}$$
SMC for Probabilistic Programming

Want samples from

$$\pi_n(\tilde{x}_{1:n}) \propto p(y_n | \tilde{x}_{1:n}) p(\tilde{x}_n | \tilde{x}_{1:n-1}) \pi_{n-1}(\tilde{x}_{1:n-1})$$

Have a sample-based approximation to

$$\hat{\pi}_{n-1}(\tilde{x}_{1:n-1}) \triangleq \sum_{k=1}^{K} W_{n-1}^{k} \delta_{\tilde{x}_{1:n-1}^{k}}(\tilde{x}_{1:n-1})$$

Sample from

$$\tilde{x}_{1:n-1}^{a_{n-1}} \sim \hat{\pi}_{n-1}(\tilde{x}_{1:n-1}) \quad \tilde{x}_{n}^{a_{n-1}} | \tilde{x}_{1:n-1}^{a_{n-1}} \sim p(\tilde{x}_{n} | \tilde{x}_{1:n-1}^{a_{n-1}})$$

$$\tilde{x}_{1:n}^{k} = \tilde{x}_{1:n-1}^{a_{n-1}} \times \tilde{x}_{n}^{k}$$

Importance weight by

$$w(\tilde{x}_{1:n}^{k}) = p(y_n | \tilde{x}_{1:n}^{k}) = g_n(y_n | \tilde{x}_{1:n}^{k})$$

$$W_{n}^{k} \triangleq \frac{w(\tilde{x}_{1:n}^{k})}{\sum_{k'=1}^{K} w(\tilde{x}_{1:n}^{k'})}$$
SMC for Probabilistic Programming

Want samples from

$$\pi_n(\tilde{x}_{1:n}) \propto p(y_n|\tilde{x}_{1:n})p(\tilde{x}_n|\tilde{x}_{1:n-1})\pi_{n-1}(\tilde{x}_{1:n-1})$$

Have a sample-based approximation to

$$\hat{\pi}_{n-1}(\tilde{x}_{1:n-1}) \triangleq \sum_{k=1}^{K} W_{n-1}^k \delta_{\tilde{x}_{1:n-1}^k}(\tilde{x}_{1:n-1})$$

Sample from

$$\tilde{x}_{1:n-1}^{a_{n-1}} \sim \hat{\pi}_{n-1}(\tilde{x}_{1:n-1})$$

$$\tilde{x}_n^{a_{n-1}} \sim p(\tilde{x}_n|\tilde{x}_{1:n-1}^{a_{n-1}})$$

$$\tilde{x}_{1:n}^k = \tilde{x}_{1:n-1}^{a_{n-1}} \times \tilde{x}_n^k$$

Importance weight by

$$w(\tilde{x}_{1:n}^k) = p(y_n|\tilde{x}_{1:n}^k) = g_n^k(y_n|\tilde{x}_{1:n}^k)$$

$$W_n^k \triangleq \frac{w(\tilde{x}_{1:n}^k)}{\sum_{k'=1}^{K} w(\tilde{x}_{1:n}^{k'})}$$
SMC for Probabilistic Programming

Want samples from

\[ \pi_n(\tilde{X}_{1:n}) \propto p(y_n|\tilde{X}_{1:n})p(\tilde{X}_n|\tilde{X}_{1:n-1})\pi_{n-1}(\tilde{X}_{1:n-1}) \]

Have a sample-based approximation to

\[ \hat{\pi}_{n-1}(\tilde{X}_{1:n-1}) \triangleq \sum_{k=1}^{K} W_{n-1}^k \delta_{\tilde{X}_{1:n-1}^k}(\tilde{X}_{1:n-1}) \]

Sample from

\[ \tilde{X}_{1:n-1}^{a_{n-1}} \sim \hat{\pi}_{n-1}(\tilde{X}_{1:n-1}) \]

\[ \tilde{X}_n^{a_{n-1}} \sim p(\tilde{X}_n|\tilde{X}_{1:n-1}^{a_{n-1}}) \]

\[ \tilde{X}_{1:n}^k = \tilde{X}_{1:n-1}^{a_{n-1}} \times \tilde{X}_n^k \]

Importance weight by

\[ w(\tilde{X}_{1:n}^k) = p(y_n|\tilde{X}_{1:n}^k) = g_n(y_n|\tilde{X}_{1:n}^k) \]

\[ W_n^k \triangleq \frac{w(\tilde{X}_{1:n}^k)}{\sum_{k'=1}^{K} w(\tilde{X}_{1:n}^{k'})} \]
SMC Schematic

Intuitively:
- run
- wait/weight
- continue

Threads

observe

continuations
Metropolis Hastings = “Single Site” MCMC = LMH

Posterior distribution of execution traces is proportional to trace score with observed values plugged in

\[ \gamma(x) \triangleq p(x, y) = \prod_{i=1}^{N} g_i(y_i | \phi_i) \prod_{j=1}^{M} f_j(x_j | \theta_j) \]

\[ \pi(x) \triangleq p(x|y) = \frac{\gamma(x)}{Z} \]

Metropolis-Hastings acceptance rule

\[ \alpha = \min \left( 1, \frac{\pi(x')q(x'|x')}{\pi(x)q(x|x')} \right) \]

Need proposal
LMH Proposal

\[ q(x'|x^s) = \frac{1}{M^s} \kappa(x'_\ell|x^s_\ell) \prod_{j=\ell+1}^{M'} f'_j(x'_j|\theta'_j) \]

**Number of samples in original trace**

**Probability of new part of proposed execution trace**
LMH Acceptance Ratio

“Single site update” = sample from the prior = run program forward

\[ \kappa(x'_m | x_m) = f_m(x'_m | \theta_m), \theta_m = \theta'_m \]

MH acceptance ratio

Number of sample statements in original trace

\[ \alpha = \min \left( 1, \frac{\gamma(x')M \prod_{j=m}^{M} f_j(x_j | \theta_j)}{\gamma(x)M' \prod_{j=m}^{M'} f'_j(x'_j | \theta'_j)} \right) \]

Number of sample statements in new trace

Probability of original trace continuation restarting proposal trace at m\(^{th}\) sample

Probability of proposal trace continuation restarting original trace at m\(^{th}\) sample
LMH Schematic
Implementation Strategy

• Interpreted
  • Interpreter tracks side effects and directs control flow for inference

• Compiled
  • Leverages existing compiler infrastructure
  • Can only exert control over flow from within function calls
    • e.g. sample, observe, predict

Wingate, Stuhlmüller, Goodman “Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation” AISTATS 2011
Paige and Wood “A Compilation Target for Probabilistic Programming Languages” ICML 2014
Probabilistic C

Intuitively
- run
- wait/weight
- fork

Processes

new processes

observe

Paige and Wood “A Compilation Target for Probabilistic Programming Languages” ICML 2014
Compilation

- LW
  - sample: inject random values

- LMH
  - catalog all random choices and compare traces by running new future

- SMC - run multiple independent futures without corrupting past
  - Must have control over the “rest of the computation”

- No longer control execution, can only interrupt and exert control at key points
  - Start, Sample, Observe, Predict, Terminate
Continuations

• A *continuation* is a function that encapsulates the “rest of the computation”

• A Continuation Passing Style (CPS) transformation rewrites programs so
  • no function ever returns
  • every function takes an extra argument, a function called the *continuation*

• Standard programming language technique

• No limitations

Fischer, Kiselyov, and Shan “Purely functional lazy non-deterministic programming” ACM Sigplan 2009
Goodman and Stuhlmüller http://dippl.org/ 2014
Tolpin https://bitbucket.org/probprog/anglican/ 2014
Example CPS Transformation

;;; Standard Clojure:
(println (+ (* 2 3) 4))

;;; CPS transformed:
(*& 2 3 (fn [x] (+& x 4 println)))

;;; CPS-transformed "primitives"
(defn +& [a b k] (k (+ a b)))
(defn *& [a b k] (k (* a b)))
CPS Explicitly Linearizes Execution

• Compiling to a pure language with lexical scoping ensures
  
  A. variables needed in subsequent computation are bound in the environment
  
  B. can’t be modified by multiple calls to the continuation function

Let’s try a new example: the Pythagorean theorem, which we use to compute the hypotenuse of a right triangle.

```clojure
(defn pythag&
  "compute sqrt(x^2 + y^2)"
  [x y k]
  (square& x
    (fn [xx]
      (square& y
        (fn [yy]
          (+& xx yy
            (fn [xxyy]
              (sqrt& xxyy k))))))))
```

\[
xx = x^2 \\
yy = y^2 \\
xxyy = xx + yy \\
\cdot = \sqrt{xxyy}
\]
Anglican Programs

(defquery flip-example [outcome]
  (let [p (sample (uniform-continuous 0 1))]
    (observe (flip p) outcome)
    (predict :p p)))

(defquery flip-example [outcome]
  (let [u (uniform-continuous 0 1)
        p (sample u)
        dist (flip p)]
    (observe dist outcome)
    (predict :p p)))
Are “Compiled” to Native CPS-Clojure

(defn flip-query& [outcome k1]
  (uniform-continuous& 0 1) (let [u (uniform-continuous 0 1)]
    (fn [dist1]
      (sample& dist1) p (sample u)
        (fn [p] (fn [p k2]
          (flip& p) dist (flip p))
            (fn [dist2]
              (observe& dist2 outcome) (observe dist outcome)
                (fn []
                  (predict& :p p k2))))))(predict :p p)
  p k1))))))

;; CPS-ed distribution constructors
(defn uniform-continuous& [a b k]
  (k (uniform-continuous a b)))

(defn flip& [p k]
  (k (flip p)))

---

Clojure

Anglican “linearized”
Are “Compiled” to Native CPS-Clojure

(defn flip-query& [outcome k1]
  (uniform-continuous& 0 1) (let [u (uniform-continuous 0 1)]
    (fn [dist1]
      (sample& dist1 p (sample u)
        (fn [p] ((fn [p k2]
                    (flip& p dist (flip p))
                (fn [dist2]
                    (observe& dist2 outcome (observe dist outcome)
                      (fn []
                        (predict& :p p k2)))))))))))

;; CPS-ed distribution constructors
(defn uniform-continuous& [a b k]
  (k (uniform-continuous a b)))

(defn flip& [p k]
  (k (flip p)))

Clojure

Anglican “linearized”
Explicit Functional Form for “Rest of Program”

```
(defn flip-query & [outcome k1]
  (uniform-continuous & 0 1
   (fn [dist1]
     (sample & dist1
       (fn [p] ((fn [p k2]
         (flip & p
           (fn [dist2]
             (observe & dist2 outcome
               (fn []
                 (predict & :p p k2)))))))
             p k1)))))))
```
(defn flip-query& [outcome k1]
    (uniform-continuous& 0 1
        (fn [dist1]
            (sample& dist1
                (fn [p] ((fn [p k2]
                    (flip& p
                        (fn [dist2]
                            (observe& dist2 outcome
                                (fn []
                                    (predict& :p p k2))))))))))))

And then talk about implementation, from transformation compilation to prob.-c to Anglican style with brooks' prob.-c and how to get "the same" via CPS transformation and compilation to a pure functional language – this works because of datastructures.

We can show Stanford's stuff on C3 as a family that decends from trans. comp. to our family of inference methods that decend, roughly, from SMC.

We should talk about why functional datastructures are required for valid statistical reasoning. And why various languages have continuations natively and why some don't and what that means.

Pure functional data structures.

composibility (tuan anh's request hmm, Noah style models),

why semantics matters and point out that higher order probabilistic languages don't have defined semantics

program transformations (by example) and hakuru as something that could arise (advanced)

11. Advanced Applications

FW Hit on Picture and perception.
Controllable

```
(defn flip-query [outcome k1]
  (uniform-continuous 0 1
   (fn [dist1]
    (sample dist1
      (fn [p] ((fn [p k2]
              (flip p
                (fn [dist2]
                  (observe dist2 outcome
                    (fn []
                      (predict :p p k2))))))))
      p k1))))

(defn weighted-sample [query &]
  (init-backend!
   (query &
     terminate
     :predicts @backend
     :log-weight @backend)
     (weighted-sample flip-query-true &)
     10.6)

And then talk about implementation, from transformation compilation to prob.-c to Anglican style with brooks' prob.-c and how to get "the same" via CPS transformation and compilation to a pure functional language – this works because of datastructures. Mention massive parallelism automatically.

We can show Stanford's stuff on C3 as a family that descends from trans. comp. to our family of inference methods that descend, roughly, from SMC.

We should talk about why functional datastructures are required for valid statistical reasoning. And why various languages have continuations natively and why some don't and what that means.

Pure functional data structures.

composibility (tuan anh's request hmm, Noah style models),

why semantics matters and point out that higher order probabilistic languages don't have defined semantics

program transformations (by example) and hakuru as something that could arise (advanced)

11. Advanced Applications

Hit on Picture and perception.

webPPL CPS compiles to pure functional Javascript
Inference “Backend”

(defn sample& [dist k]
    ;; [ ALGORITHM-SPECIFIC IMPLEMENTATION HERE ]
    ;; Pass the sampled value to the continuation
    (k (sample dist)))

(defn observe& [dist value k]
    (println "log-weight =" (observe dist value))
    ;; [ ALGORITHM-SPECIFIC IMPLEMENTATION HERE ]
    ;; Call continuation with no arguments
    (k))

(defn predict& [label value k]
    ;; [ ALGORITHM-SPECIFIC IMPLEMENTATION HERE ]
    (k label value))
Common Framework

Pure compiled deterministic computation

start $P$

continue $P$

continue $P$

terminate $P$

“Backend”

sample $(f, \theta, k)$

observe $(g, \phi, y, k)$

predict $(z, k)$

terminate
Likelihood Weighting “Backend”

(defn sample& [dist k]
  ;; Call the continuation with a sampled value
  (k (sample dist)))

(defn observe& [dist value k]
  ;; Compute and record the log weight
  ;; Call the continuation with no arguments
  (add-log-weight! (observe dist value))
  (k))

(defn predict& [label value k]
  ;; Store predict, and call continuation
  (store! label value)
  (k))
Likelihood Weighting Example

Let's try a new example: the Pythagorean theorem, which we use to compute the hypotenuse of a right triangle.

1. Define sqrt& as a new "primitive" function

\[
\text{defn} \quad \text{sqrt}\;[a]\;[k] = (k \text{Math/sqrt}\;a)
\]

2. Define square& using *&

\[
\text{defn} \quad \text{square}\;[a]\;[k] = (*\;a\;a)
\]

3. For example:

\[
\text{(square 5 \text{println})} \quad 25
\]

\[
\text{(sqrt 9 \text{println})} \quad 3
\]

4. Define pythag& "compute sqrt(x^2 + y^2)"

\[
\text{defn} \quad \text{pythag}\;[x]\;[y]\;[k] = \text{square}\;x\;[\text{fn} \;[xx] = \text{square}\;y\;[\text{fn} \;[yy] = (+\;xx\;yy\;[\text{fn} \;[xxyy] = \text{sqrt}\;xxyy\;k])])}
\]

5. Test it:

\[
\text{(pythag 3 4 \text{println})} \quad 5
\]

\[
\text{(pythag 5 12 \text{println})} \quad 13
\]

Note that the continuations we define within the pythag& function have state, in their closure! We cannot write, for example, the function

\[
(\text{fn} \;[yy] = (+\;xx\;yy\;k))
\]

since it requires a value xx, a variable which is available due to being in scope at the time the function is called, rather than passed in as an argument. Immutability in Clojure/Anglican allows us to "get away with" calling these continuation functions repeatedly, anyway, since no subsequent executions of the program will modify these variables.

10.5.2 A Example of a Probabilistic Model

\[
\text{(defquery} \;\text{flip-example} \;[\text{outcome}] = \text{(let} \;[p (\text{sample} \;\text{uniform-continuous} \;0 \;1)])}
\]

\[
(\text{observe} \;\text{flip} \;p \;\text{outcome})
\]

\[
(\text{predict} \;:p \;p)
\]

\[
(\text{defquery} \;\text{flip-example} \;[\text{outcome}]
\]

\[
(\text{let} \;[p (\text{sample} \;\text{uniform-continuous} \;0 \;1)])
\]

\[
(\text{observe} \;\text{flip} \;p \;\text{outcome})
\]

\[
(\text{predict} \;:p \;p)
\]
(defn sample& [dist k]
    ;; Call the continuation with a sampled value
    (k (sample dist)))

(defn observe& [dist value k]
    ;; Block and wait for K calls to reach observe&
    ;; Compute weights
    ;; Use weights to subselect continuations to call
    ;; Call K sampled continuations (often multiple times)
    )

(defn predict& [label value k]
    ;; Store predict, and call continuation
    (store! label value)
    (k))
Particle Markov Chain Monte Carlo

Andrieu, Doucet, Holenstein “Particle Markov chain Monte Carlo methods.” JRSSB 2010

• Iterable SMC
  - PIMH : “particle independent Metropolis-Hastings”
  - PGIBBS : “iterated conditional SMC”
  - PGAS : “particle Gibbs ancestral sampling"

Wood, van de Meent, Mansinghka “A new approach to probabilistic programming inference” AISTATS 2014
PIMH Math

- Each sweep of SMC can compute

\[
\hat{Z} = \prod_{n=1}^{N} \hat{Z}_n = \prod_{n=1}^{N} \frac{1}{K} \sum_{k=1}^{K} w(\hat{x}_{1:n}^k)
\]

- PIMH is MH that accepts entire new particle sets w.p.

\[
\alpha_{PIMH}^s = \min \left( 1, \frac{\hat{Z}^*}{\hat{Z}^{s-1}} \right)
\]

- And all particles can be used

\[
\hat{E}_{PIMH}[Q(x)] = \frac{1}{S} \sum_{s=1}^{S} \sum_{k=1}^{K} W^{s,k} Q(x^{s,k})
\]

Wood, van de Meent, Mansinghka “A new approach to probabilistic programming inference” AISTATS 2014
Paige and Wood “A Compilation Target forProbabilistic Programming Languages” ICML 2014
Particle Cascade

Asynchronously
- simulate
- weight
- branch

Paige, W., Doucet, Teh; NIPS 2014
(defn sample& [a dist k]
  (let [;; reuse previous value,
        ;; or sample from prior
        x (or (get-cache a)
             (sample dist))]
    ;; add to log-weight when reused
    (when (get-cache a)
       (add-log-weight! (observe dist x)))
    ;; store value and its log prob in trace
    (store-in-trace! a x dist)
    ;; continue with value x
    (k x)))

(defn observe& [dist value k]
  ;; Compute and record the log weight
  (add-log-weight! (observe dist value))
  ;; Call the continuation with no arguments
  (k))
D. Wingate, A. Stuhlmueller, and N. D. Goodman. 

"C3: Lightweight Incrementalized MCMC for Probabilistic Programs using Continuations and Callsite Caching." 

"Venture: a higher-order probabilistic programming platform with programmable inference." 
Inference Backends in Anglican

- 14+ algorithms
- Average 165 lines of code per!
- Can implement and use without touching core code base.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type</th>
<th>Lines of Code</th>
<th>Citation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>smc</td>
<td>IS</td>
<td>127</td>
<td>Wood et al. AISTATS, 2014</td>
<td>Sequential Monte Carlo</td>
</tr>
<tr>
<td>importance</td>
<td>IS</td>
<td>21</td>
<td></td>
<td>Likelihood weighting</td>
</tr>
<tr>
<td>pcascade</td>
<td>IS</td>
<td>176</td>
<td>Paige et al., NIPS, 2014</td>
<td>Particle cascade: Anytime asynchronous sequential Monte Carlo</td>
</tr>
<tr>
<td>pgbbs</td>
<td>PMCMC</td>
<td>121</td>
<td>Wood et al. AISTATS, 2014</td>
<td>Particle Gibbs (iterated conditional SMC)</td>
</tr>
<tr>
<td>pimh</td>
<td>PMCMC</td>
<td>68</td>
<td>Wood et al. AISTATS, 2014</td>
<td>Particle independent Metropolis-Hastings</td>
</tr>
<tr>
<td>pgas</td>
<td>PMCMC</td>
<td>179</td>
<td>van de Meent et al., AISTATS, 2015</td>
<td>Particle Gibbs with ancestor sampling</td>
</tr>
<tr>
<td>lmh</td>
<td>MCMC</td>
<td>177</td>
<td>Wingate et al., AISTATS, 2011</td>
<td>Lightweight Metropolis-Hastings</td>
</tr>
<tr>
<td>almh</td>
<td>MCMC</td>
<td>320</td>
<td>Tolpin et al., ECML PKDD, 2015</td>
<td>Adaptive scheduling lightweight Metropolis-Hastings</td>
</tr>
<tr>
<td>rmh*</td>
<td>MCMC</td>
<td>319</td>
<td>-</td>
<td>Random-walk Metropolis-Hastings</td>
</tr>
<tr>
<td>palmh</td>
<td>MCMC</td>
<td>66</td>
<td>-</td>
<td>Parallelised adaptive scheduling lightweight Metropolis-Hastings</td>
</tr>
<tr>
<td>plmh</td>
<td>MCMC</td>
<td>62</td>
<td>-</td>
<td>Parallelised lightweight Metropolis-Hastings</td>
</tr>
<tr>
<td>bamc</td>
<td>MAP</td>
<td>318</td>
<td>Tolpin et al., SoCS, 2015</td>
<td>Bayesian Ascent Monte Carlo</td>
</tr>
<tr>
<td>siman</td>
<td>MAP</td>
<td>193</td>
<td>Tolpin et al., SoCS, 2015</td>
<td>MAP estimation via simulated annealing</td>
</tr>
</tbody>
</table>
Wrap Up
Where We Stand

• Probabilistic programming concept
  • Long well established

• Tool maturity
  • Homework
  • Prototyping
  • Research
  • Advanced research
  • Small real-world applications

• Put-offs
  • Some highly optimized models that you know to scale well don’t necessarily scale well in current probabilistic programming systems.
Opportunities
Static Efficiencies

- Automated program transformations that simplify or eliminate inference (moving observes up and out)

```
(defquery beta-bernoulli [observation]
  (let [dist (beta 1 1)
         theta (sample dist)
         like (flip theta)]
    (observe like observation)
    (predict :theta theta)))
```

```
(defquery beta-bernoulli [observation]
  (let [dist (beta (if observation 2 1) (if observation 1 2))
         theta (sample dist)]
    (predict :theta theta)))
```

Yang - Keynote Lecture, APLAS (2015)
Normalization

- Program analyses that identify algebraic or algorithmic normalization opportunities

```
(defquery marsaglia [mu sigma]
  (let [u (uniform-continuous -1.0 1.0)]
    (predict :x
      (loop [x (sample u)
                 y (sample u)]
        (let [s (+ (* x x) (* y y))]
          (if (< s 1.0)
            (+ mu (* sigma
                          (* x (sqrt (* -2.0 (/ (log s) s))))))
            (recur (sample u) (sample u))))))))
```

\[ p(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \]
Data driven proposals

Wrap Up
A Gentle Plea

• Bayesians
  • Stop writing assembly code!

• Join us
  • Try writing models in our languages
  • Contribute inference algorithms

• Neural net people
  • Help make inference better
  • Train your neural nets to do something interpretable
Best Way to AI

- Neural nets end to end (DeepMind)
- Generic probabilistic programs used to impose evolutionary regularity on that which is computed by deep networks.
Bubble Up

Models
Probabilistic Programming Language
Probabilistic Programming System
Inference
Bubble Up

AI

Models

Probabilistic Programming Language

Probabilistic Programming System

Inference
Anglican Resources

• General
  • http://www.robots.ox.ac.uk/~fwood/anglican/

• Learning probabilistic programming and Anglican
  • https://bitbucket.org/probprog/mlss2015

• Writing applications
  • https://bitbucket.org/probprog/anglican-user

• The core / looking at inference algorithms
  • https://bitbucket.org/probprog/anglican

• Trying it out (5 min. install)
  • https://bitbucket.org/probprog/anglican-examples
Go-To Resources

• Writing your own probabilistic programming language
  • http://dippl.org

• Model example repository
  • http://forestdb.org/

• Easiest places to start (browser-based)
  • http://webppl.org/
  • https://probmods.org/

• Place to find all the literature in one place
  • http://probabilistic-programming.org/wiki/Home

• Place to go for the most advanced ideas in prob. prog.
  • http://probcomp.csail.mit.edu/venture/
Some Final Thoughts

Adopting the kinds of abstraction boundaries suggested by probabilistic programming practice will move the field of machine learning forward much faster make it easier for inference and modeling experts to work together.

Probabilistic programming is not about making what you already do faster or somehow better but instead about making it possible to do things that would otherwise be nearly impossible to do.
Thank You

Faculty Market

van de Meent

Graduating

Paige

Too Late

Tolpin

Perov

Le

Yang

Tenenbaum

Mansinghka

• Funding : DARPA
Postdoc Openings

• 3 probabilistic programming postdoc openings

• http://goo.gl/BtoCER
Next Tutorial By

van de Meent

Paige

Mansinghka

Pfeffer

Shan

Perov

Wingate

Goodman

Ritchie

Gordon

Stuhlmüller

Roy

Russell

Scibior