Probabilistic Programming

Frank Wood
fwood@robots.ox.ac.uk

AGI 2015
Inverse Graphics

Captcha Solving

Mesh Fitting


Directed Design

Stable Static Structures

Procedural Graphics


Probabilistic Program Induction

Yura Perov and Frank Wood.
"Learning Probabilistic Programs."
Policy Learning in POMDPs

Case Studies

We demonstrate the proposed policy learning method on three problem domains: (1) the Canadian Traveller Problem, (2) a modified version of the RockSample POMDP, and (3) an optimal diagnosis benchmark inspired by the classic children's game Guess Who. Each of these domains can be formulated as a POMDP. This means that there is some form of unobserved state in the problem instance, and the agent must choose actions based on contextual information $x_t$ that can be described in terms of an information state $x_t = (u_0, o_1, ..., u_t, o_t)$.

The aim of these studies is to explore how probabilistic programs can be used to define policies tailored to the structure of each domain. Fundamentally, some information must be discarded when making a decision. Program policies encode our intuition about what information is most relevant in a given context. As such, these studies are not intended to achieve results that are competitive with current state-of-the-art specialized techniques for POMDPs (see Shani et al. [2013] for a recent overview). Rather, we consider probabilistic programs as a concise algorithmic representation of domain-specific probabilistic mappings from information states to actions, in order to describe the search space over policies in terms of a moderate yet not unwieldy number of parameters.

5.1 Evaluation Setup

We use the same experimental setup in each of the three domains. A trial begins with a learning phase, in which BBEM is used to learn the policy hyperparameters, followed by a number of testing episodes in which the agent chooses actions according to a fixed learned policy. At each gradient update step, we use 1000 samples to calculate a gradient estimate. Each testing phase consists of 1000 episodes. All shown results are based on test-phase simulations.

Stochastic gradient methods can be sensitive to the learning rate parameters. Results reported here use a RMSProp style rescaling of the gradient [Hinton et al.], which normalizes the gradient by a discounted rolling decaying average of its magnitude with decay factor $\gamma = 0.9$. We use a step size schedule $\alpha_k = \alpha_0 / (\tau + k)$ as reported in [Hoffman et al., 2013], with $\tau = 1$, $\alpha_0 = 0.5$ in all experiments. We use a relatively conservative base learning rate $\alpha_0 = 0.1$ in all reported experiments. For independent trials performed across a range $1, 2, 5, 10, ..., 1000$ of total gradient steps, consistent convergence was observed in all runs using over 100 gradient steps.

5.2 Canadian Traveller Problem

In the Canadian Traveller Problem [Papadimitriou and Yannakakis, 1991], an undirected graph $G = (V,E)$ is given, along with the cost $w_e$ of traversing every edge $e \in E$, and the probability $p_e$ that the edge is open. The agent must traverse the graph from the initial node $s$ to the goal node $t$ at the lowest possible cost. The agent does not know the state of an edge until it reaches one of the edge's vertices. The problem is NP-hard [Fried et al., 2013], and heuristic online and offline approaches [Eyerich et al., 2010] are used to solve problem instances.

Here we learn a policy based on the depth-first search (DFS) — the agent traverses the graph in the depth-first order until the goal node is reached (only connected instances are considered). Depth-first search
Landscape

ML: Algorithms & Applications

STATS: Inference & Theory

PL: Compilers, Semantics, Analysis

Probabilistic Programming
Conceptualization

Inference

Parameters → Program → Output

Parameters → Program → Observations

\[ p(y|x)p(x) \]

\[ y \]

CS

Probabilistic Programming

Statistics
Operative Definition

“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 et al, 2014
What are the goals of probabilistic programming?
Increase Programmer Productivity

http://www.robots.ox.ac.uk/~fwood/anglican/examples/viewer/?worksheet=complexityreduction

(fn [x] (logb 1.04 (+ 1 x)))

Lines of Matlab/Java Code vs. Lines of Anglican Code

- Collapsed LDA
- DP Conjugate Mixture
- DDPMO, [Neiswanger et al 2014]
- PDIA, [Pfau 2010]
- HPYP, [Wood 2007]

Graph showing the relationship between lines of Matlab/Java code and lines of Anglican code. The graph includes various algorithms and their corresponding line counts.
Latent Dirichlet Allocation is formally written as:

Each of $\pi_d$ is a Dirichlet distribution over the $m$ mixture generative model for the $d$th document, where $m$ denotes the number of words in document $d$, $K$ is the number of documents, $D$ is the number of topics in the model, and $w$ is the number of words in the vocabulary. We use $z_{i,d}$ to denote the topic assignment for the $i$th word in document $d$. Under the uniform deletion model, the number of alive allocation variables at time $t$ is the invariant distribution of the Markov transition kernel $p(c,t|c,t-1)$. In the time series literature, many approaches are available to build such transition kernels based on copulas (Joe, 1997) or Gibbs sampling techniques (Pitt and Walker, 2005). Inference can be performed using techniques such as the collapsed Gibbs sampler (Blei and Ng, 2003), collapsed variational inference (Blei and Jordan, 2003), or using collapsed stick-breaking processes (R Inference Engine(s))
Probabilistic ML, Haskell, Scheme, …

Discrete RV’s

Only

2000

1990

Systems

PL

AI

ML

STATS

Figaro

HANSAI

IBAL

Prism

KMP

Discrete RV’s

Only

webPPL

Probabilistic-C

Venture

Anglican

Church

Factorie

infer.NET

Blog

JAGS

STAN

LibBi

WinBUGS

Bounded

Recursion

Simula

Prolog
Anglican

• Higher order, pure functional

• Compiled (CPS -> Clojure -> JVM bytecode)
  • Complete JVM language family interoperability

• First class distributions

• 15+ composable inference algorithms
  • SMC
  • CASCADE
  • PMCMC (PIMH, PGIBBS, PGAS)
  • (Adaptive) LMH
  • …

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

- [http://www.robots.ox.ac.uk/~fwood/anglican/](http://www.robots.ox.ac.uk/~fwood/anglican/)

- Open source (GPLv3)
  - core: [https://bitbucket.org/probprog/anglican](https://bitbucket.org/probprog/anglican)
  - user: [https://bitbucket.org/probprog/anglican-user](https://bitbucket.org/probprog/anglican-user)
  - tutorial: [https://bitbucket.org/probprog/mlss2015](https://bitbucket.org/probprog/mlss2015)
Traditional Bayesian Statistics

\[
\begin{align*}
\text{(defquery gaussian-model [data]}
& \quad \text{(let [mu (sample (normal 1 (sqrt 5)))}
& \quad \quad \quad \text{sigma (sqrt 2)]}
& \quad \quad \text{(map (fn [x] (observe (normal mu sigma) x)) data)}
& \quad \quad \text{(predict :mu mu)))}
& \quad \text{\(\mu \sim \text{Normal}(1, \sqrt{5})\)}
& \quad \text{\(y_i|\mu \sim \text{Normal}(\mu, \sqrt{2})\)}
& \quad \text{\(y_1 = 9, y_2 = 8\)}
& \quad \mu|y_{1:2} \sim \text{Normal}(7.25, 0.9129)
\end{align*}
\]

(\text{def dataset [9 8]})

(\text{def posterior}
\quad \text{((conditional gaussian-model}
\quad \quad \text{:pgibbs}
\quad \quad \text{:number-of-particles 1000) dataset)})

(\text{def posterior-samples}
\quad \text{(repeatedly 20000 #(sample posterior)))}
(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: ~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))
Inference Over Conditioned Execution Traces

(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)

  obs-dist (normal 2 0.1)]

(observe obs-dist num-balls-in-box)

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

\[ p(z|y, h) = \frac{p(y|z, h)p(z|h)}{p(y|h)} \]

\[ p(y|h) = \int p(y|z, h)p(z|h)dz \]
Automatic Complexity Regularization

\[
p(y|h) = H_{simple}
\]

\[
h = H_{complex}
\]

\[
p(h|y) \propto p(y|h)p(h)
\]

Bayesian Occam’s Razor

Probabilistic Programming Is Fully Generative

\[ x = z \cup h \]

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

<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</td>
<td>reward</td>
</tr>
<tr>
<td>world</td>
<td>simulator output</td>
</tr>
<tr>
<td>automata</td>
<td>sequence</td>
</tr>
</tbody>
</table>
How Does it Work?
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
Trace Probability

- observe data points $y_n$
- internal random choices $x_n$
- simulate from $f(x_n \mid x_{1:n-1})$
- by running the program forward
- weight execution traces by $g(y_n \mid x_{1:n})$

$$p(y_{1:N}, x_{1:N}) = \prod_{n=1}^{N} g(y_n \mid x_{1:n}) f(x_n \mid x_{1:n-1})$$
(let [x-1-1 3
      x-1-2 (sample (discrete (range x-1-1)))]
  (if (not= x-1-2 1)
    (let [x-2-1 (+ x-1-2 7)]
      (sample (poisson x-2-1))))
Observe

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

Iteratively,

- simulate
- weight
- resample
SMC for Probabilistic Programming

\[ p(x_{1:n-1}|y_{1:n-1}) \approx \sum_{\ell=1}^{L} w_{n-1}^{\ell} \delta_{x_{1:n-1}^{\ell}}(x_{1:n-1}) \]

\[ p(x_{1:n}|y_{1:n}) \propto g(y_n|x_{1:n}) f(x_n|x_{1:n-1}) p(x_{1:n-1}|y_{1:n-1}) \]

\[ q(x_{1:n}|y_{1:n}) = f(x_n|x_{1:n-1}) p(x_{1:n-1}|y_{1:n-1}) \]

\[ p(x_{1:n}|y_{1:n}) \approx \sum_{\ell=1}^{L} g(y_n|x_{1:n}^{\ell}) \delta_{x_{1:n}^{\ell}}(x_{1:n}), \quad x_{1:n}^{\ell} = x_{n}^{\ell} x_{1:n-1}^{a_{n-1}^{\ell}} \sim f \]
SMC for Probabilistic Programming

Intuitively
- run
- wait
- fork

Threads
observe delimiter

continuations
Issues

- Degeneracy
- Not iterable (naively)
PMCMC for Probabilistic Programming

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

• Sequential Monte Carlo is now a building block for other inference techniques

• Iterable SMC
  - PIMH: “particle independent Metropolis-Hastings”
  - PGIBBS: “iterated conditional SMC”

Andrieu, Doucet, Holenstein “Particle Markov chain Monte Carlo methods.” JRSSB 2010
Better Inference Per Unit Energy

Simulation

Time [s]

Forward-Backward

PMCMC

RDB
PMCMC (and SMC) Methods Only Require

- Initialization (sample)
  \[ p(x_1) \]

- Forward simulation (sample)
  \[ f(x_n|x_{1:n-1}) \]

- Observation likelihood computation
  - pointwise evaluation up to normalization
  \[ g(y_n|x_{1:n}) \]
Stop Making New Probabilistic Programming Languages

sort-of
Probabilistic C

- Standard C with two new directives: `observe` and `predict`
- Is compiled to parallel machine code by standard compilers
- Relies on standard operating system functionality: processes, forking, mutexes, shared memory
- Compiled programs automatically do inference
- Emits posterior samples of predicted quantities

Paige, W.; ICML 2014
Simple example program

Posterior mean of a Gaussian, given i.i.d. draws

**observe** constrains program execution

**predict** emits sampled values

```c
#include "probabilistic.h"

int main(int argc, char **argv) {
    double var = 2;
    double mu = normal_rng(1, 5);
    observe(normal_lnp(9, mu, var));
    observe(normal_lnp(8, mu, var));
    predict("mu, %f\n", mu);
    return 0;
}
```

mean, 8.013323
mean, 8.013323
mean, 6.132654
mean, 7.229289
mean, 7.027069
mean, 7.194609
mean, 7.194609
mean, 5.218672
mean, 6.184513
A Markov model

\[ z_0 \sim \text{Discrete}(\left[\frac{1}{K}, \ldots, \frac{1}{K}\right]) \quad z_n | z_{n-1} \sim \text{Discrete}(T_{z_{n-1}}) \]

```c
#include "probabilistic.h"
#define K 3
#define N 17

/* Markov transition matrix */
static double T[K][K] = {{ 0.1, 0.5, 0.4 },
                        { 0.2, 0.2, 0.6 },
                        { 0.15, 0.15, 0.7 }};

/* Prior distribution on initial state */
static double initial_state[K] = { 1.0/3, 1.0/3, 1.0/3 };

/* Generative program for a Markov model */
int main(int argc, char **argv) {

    int states[N];
    for (int n=0; n<N; n++) {
        states[n] = (n==0) ? discrete_rng(initial_state, K):
                        discrete_rng(T[states[n-1]], K);
        predict("state[%d],%d\n", n, states[n]);
    }

    return 0;
}
```
Conditioning on observed data

\[ z_0 \sim \text{Discrete}([1/K, \ldots, 1/K]) \quad z_n | z_{n-1} \sim \text{Discrete}(T_{z_{n-1}}) \quad y_n | z_n \sim \text{Normal}(\mu_{z_n}, \sigma^2) \]
Changing the generative model is easy
Suppose the transition matrix were unknown: \( T_k \sim \text{Dirichlet}(\alpha_k) \)

```c
#define N 17

/* Markov transition matrix */
static double T[K][K] = {
    { 0.1, 0.5, 0.4 },
    { 0.2, 0.2, 0.6 },
    { 0.15, 0.15, 0.7 }
};

/* Observed data */
static double data[N] = {
    NAN, .9, .8, .7, 0, -.025,
    -5, -2, -.1, 0, 0.13, 0.45,
    6, 0.2, 0.3, -1, -1
};

/* Prior distribution on initial state */
static double initial_state[K] = {
    1.0/3, 1.0/3, 1.0/3
};

/* Per-state mean of Gaussian emission distribution */
static double state_mean[K] = { -1, 1, 0 }

/* Generative program for a HMM */
int main(int argc, char **argv) {
    int states[N];
    for (int n=0; n<N; n++) {
        states[n] = (n==0) ? discrete_rng(initial_state, K) :
                      discrete_rng(T[states[n-1]], K);
    }
```
Implementation

• Inference: forward simulation (SMC, particle MCMC, particle cascade, …)

• POSIX **fork:**
  - operating-system level call to clone a running process: branch on program execution state, explore many downstream paths
  - duplicates *entire* memory address space
  - efficient: lazy copy-on-write behaviour
  - parallel: each downstream path is explored by an independent OS process
The Next 700 Probabilistic Programming Languages?

W., Jeffrey Mark Siskind and Brooks Paige
(in prep. 2015)
Probabilistic Scheme

Gaussian example, in probabilistic scheme

;;; Define a (random) mean, mu
(define mu (normal 1 (sqrt 5)))
(define sigma (sqrt 2))

;;; Define a likelihood function
(define (likelihood x) (normal-lnp x mu sigma))

;;; Condition on the data
(define data (list 8 9))
(map observe-lnp (map likelihood data))

;;; Emit samples of the mean
(predict-float "mean" mu)
All we need for probabilistic scheme

- existing scheme compiler (i.e. STALIN)
- existing C compiler (i.e. GCC, clang)

```scheme
;;; ERPs
(define poisson-rng
  (foreign-procedure (double) long "poisson_rng" "probabilistic"))

(define normal-lnp
  (foreign-procedure (double double double) double "normal_lnp" "probabilistic"))
;;; plus more -rng and -lnp

;;; directives
(define observe (foreign-procedure (double) void "observe" "probabilistic"))

(define predict-value
  (foreign-procedure (char* double) void "predict_value" "probabilistic"))

;;; necessary boilerplate
(vector-ref argv 0)
```
#include <probabilistic.h>

int main(int argc, char *argv[]) {
    long a = poisson_rng(100.0) - 100;
    long b = poisson_rng(100.0) - 100;
    observe(normal_lnp(7.0, (double)(a+b), 0.00001));
    predict_value("a", (double)a);
    predict_value("b", (double)b);
}
C (GCC, CLANG)

int main(int argc, char *argv[]) {
    long a = poisson_rng(100.0)-100;
    long b = poisson_rng(100.0)-100;
    observe(normal_lnp(7.0, (double)(a+b), 0.00001));
    predict_value("a", (double)a);
    predict_value("b", (double)b);
}

Scheme (STALIN)

(define a (- (poisson-rng 100.0) 100))
(define b (- (poisson-rng 100.0) 100))
(observe (normal-lnp 7.0
            (exact->inexact (+ a b)) .00001))
(predict-value "a" (exact->inexact a))
(predict-value "b" (exact->inexact b))

Standard ML (MLTON)

val a = (poisson_rng 100.0)-100
val b = (poisson_rng 100.0)-100
val _ = observe (normal_lnp (7.0, (int64ToReal (a+b)), 0.00001))
val _ = predict_value("a", (int64ToReal a))
val _ = predict_value("b", (int64ToReal b))
val _ = return_from_main 0

Haskell (GHC)

model = do
    a <- (+(-100)) <$> poisson_rng 100.0
    b <- (+(-100)) <$> poisson_rng 100.0
    observe $ normal_lnp 7
        (realToFrac (a+b)) 0.00001
    predict_value "a" (realToFrac a)
    predict_value "b" (realToFrac b)
    return ()
Perception is a CAPTCHA from AOL, and the fourth row shows an example where our system makes errors. We developed a probabilistic graphics program for reading short snippets of degraded text consisting of arbitrary digits and letters. See Figure 2 for representative inputs and outputs. In this program, is, and how it is rotated:

\[ \text{Observed} (i = 1) = 0.5 \].

The letters, and then blurs the result. We also applied global blur to the original training image before applying the stochastic likelihood model on the blurred original and rendered images. The (S, x) = novel poses id, and the standard deviation of the Gaussian likelihood \( \theta \).

To assess the accuracy of our approach on adversarially obscured text, we developed a corpus con-...
It’s All About Inference

• Parallelism

“Asynchronous Anytime Sequential Monte Carlo” [Paige, W., Doucet, Teh NIPS 2014]

• Backwards passing

“Particle Gibbs with Ancestor Sampling for Probabilistic Programs” [van de Meent, Yang, Mansinghka, W. AISTATS 2015]

• Search

“Maximum a Posteriori Estimation by Search in Probabilistic Models” [Tolpin, W., SOCS, 2015]

• Adaptation


• Novel proposals

“Neural Adaptive Inference for Probabilistic Programming” [Paige, W.; in submission]
Thank You

• Questions?

• Funding: DARPA, Amazon, Microsoft