Probabilistic Programming

Hongseok Yang
University of Oxford
DARLING LOVE,

MY SEDUCTIVE APPETITE CLINGS TO YOUR AMBITION. MY RAPTURE LUSTS AFTER YOUR CRAVING. MY BURNING YEARNS FOR YOUR AMBITION. MY ENCHANTMENT IMPATIENTLY ADORES YOUR CURIOUS WISH. MY LOVING EAGERNESS IMPATIENTLY THIRSTS FOR YOUR LUST.

YOURS CURIOUSLY,

M.U.C.
FANCIFUL CHICKPEA,

YOU ARE MY AMOROUS SYMPATHY. MY PASSIONATE DEVOTION HOPES FOR YOUR HEART. YOU ARE MY SEDUCTIVE FONDNESS. MY WISH PANTS FOR YOUR AMOROUS ARDOUR. MY TENDER ADORATION CLINGS TO YOUR DEVOTION.

YOURS WISTFULLY,

M.U.C.
FANCIFUL DUCK,

MY AFFECTION LUSTS AFTER YOUR BEING. YOU ARE MY
SYMPATHETIC RAPTURE, MY TENDER BURNING. MY SYMPATHY LIKES
YOUR LONGING. MY CURIOUS ENTHUSIASM PANTS FOR YOUR
UNSATISFIED CRAVING.

YOURS SEDUCTIVELY,

M.U.C.
FANCIFUL DUCK,

MY AFFECTION LUSTS AFTER YOUR BEING. YOU ARE MY SYMPATHETIC RAPTURE, MY TENDER BURNING. MY SYMPATHY LIKES YOUR LONGING. MY CURIOUS ENTHUSIASM PANTS FOR YOUR UNSATISFIED CRAVING.

YOURS SEDUCTIVELY,

M.U.C.

Manchester Univ. Computer.

Produced by Strachey’s “Love Letter” (1952)
Strachey’s program

Implements a simple randomised algorithm:

1. Randomly pick two opening words.
2. Repeat the following five times:
   • Pick a sentence structure randomly.
   • Fill the structure with random words.
3. Randomly pick closing words.
Strachey’s Program
Implements a simple randomised algorithm:

1. Randomly pick two opening words.

2. Repeat the following five times:
   • Pick a sentence structure randomly.
   • Fill the structure with random words.

3. Randomly pick closing words.

1. More randomness.

random N times
Strachey’s

Implements a simple randomised algorithm:

1. **Randomly** pick two opening words.
2. Repeat the following five times:
   - Pick a sentence structure **randomly**.
   - Fill the structure with **random** words.
3. **Randomly** pick closing words.

1. More randomness.
2. Adjust randomness. Use data.
What is probabilistic programming?
(Bayesian) probabilistic modelling of data

1. Develop a new probabilistic (generative) model.
2. Design an inference algorithm for the model.
3. Using the algo., fit the model to the data.
(Bayesian) probabilistic modelling of data in a prob. prog. language

1. Develop a new probabilistic (generative) model.
2. Design an inference algorithm for the model.
3. Using the algo., fit the model to the data.
(Bayesian) probabilistic modelling of data in a prob. prog. language as a program

1. Develop a new probabilistic (generative) model.
2. Design an inference algorithm for the model.
3. Using the algo., fit the model to the data.
(Bayesian) probabilistic modelling of data in a prob. prog. language

1. Develop a new probabilistic (generative) model.

2. Design an inference algorithm for the model.

3. Using the algo., fit the model to the data.

as a program

a generic inference algo.
of the language
Line fitting
Line fitting

\[ f(x) = s \cdot x + b \]
Bayesian generative model
Bayesian generative model

\[ f(x) = s \times x + b \]

\[ y_i \sim \text{normal}(f(i), 1) \]

where \( i = 1 \ldots 5 \)

\[ s \sim \text{normal}(0, 10) \]

\[ b \sim \text{normal}(0, 10) \]
Bayesian generative model

\[ s \sim \text{normal}(0, 10) \]
\[ b \sim \text{normal}(0, 10) \]
\[ f(x) = s \times x + b \]
\[ y_i \sim \text{normal}(f(i), 1) \]
where \( i = 1 \ldots 5 \)
Bayesian generative model

\[ s \sim \text{normal}(0, 10) \]
\[ b \sim \text{normal}(0, 10) \]
\[ f(x) = s \times x + b \]
\[ y_i \sim \text{normal}(f(i), 1) \]
where \( i = 1 \ldots 5 \)

Q: posterior of \((s, b)\) given \(y_1=2.5, \ldots, y_5=10.1?\)
Posterior of \( s \) and \( b \) given \( y_i \)'s

\[
P(s, b \mid y_1, \ldots, y_5) = \frac{P(y_1, \ldots, y_5 \mid s, b) \times P(s, b)}{P(y_1, \ldots, y_5)}
\]
Posterior of s and b given $y_i$'s

$$P(s, b \mid y_1, \ldots, y_5) = \frac{P(y_1, \ldots, y_5 \mid s, b) \times P(s, b)}{P(y_1, \ldots, y_5)}$$
Posterior of s and b given $y_i$'s

\[
P(s, b \mid y_1, \ldots, y_5) = \frac{P(y_1, \ldots, y_5 \mid s, b) \times P(s, b)}{P(y_1, \ldots, y_5)}
\]
Posterior of $s$ and $b$ given $y_i$'s

$$P(s, b \mid y_1, \ldots, y_5) = \frac{P(y_1, \ldots, y_5 \mid s, b) \times P(s, b)}{P(y_1, \ldots, y_5)}$$
Posterior of $s$ and $b$ given $y_i$'s

\[
P(s, b \mid y_1, .., y_5) = \frac{P(y_1, .., y_5 \mid s, b) \times P(s, b)}{P(y_1, .., y_5)}
\]
Anglican program

(let [s (sample (normal 0 10))
     b (sample (normal 0 10))
     f (fn [x] (+ (* s x) b))]
  (observe (normal (f 1) 1) 2.5)
  (observe (normal (f 2) 1) 3.8)
  (observe (normal (f 3) 1) 4.5)
  (observe (normal (f 4) 1) 8.9)
  (observe (normal (f 5) 1) 10.1))
(let [s (sample (normal 0 10))
     b (sample (normal 0 10))
     f (fn [x] (+ (* s x) b))]
  (observe (normal (f 1) 1) 2.5)
  (observe (normal (f 2) 1) 3.8)
  (observe (normal (f 3) 1) 4.5)
  (observe (normal (f 4) 1) 8.9)
  (observe (normal (f 5) 1) 10.1))
Anglican program

(let [s (sample (normal 0 10))
     b (sample (normal 0 10))
     f (fn [x] (+ (* s x) b))]

  (observe (normal (f 1) 1) 2.5)
  (observe (normal (f 2) 1) 3.8)
  (observe (normal (f 3) 1) 4.5)
  (observe (normal (f 4) 1) 8.9)
  (observe (normal (f 5) 1) 10.1)

  (predict :sb [s b]))
Anglican program

(let [s (sample (normal 0 10))
      b (sample (normal 0 10))
      f (fn [x] (+ (* s x) b))]

  (observe (normal (f 1) 1) 2.5)
  (observe (normal (f 2) 1) 3.8)
  (observe (normal (f 3) 1) 4.5)
  (observe (normal (f 4) 1) 8.9)
  (observe (normal (f 5) 1) 10.1)

  (predict :sb [s b]))
Samples from posterior
Why should one care about prob. programming?
My favourite answer

“Because probabilistic programming is a good way to build an AI.”  (My ML colleague)
Procedural modelling

Ritchie, Mildenhall, Goodman, Hanrahan [SIGGRAPH’15]
Procedural modelling

```python
future.create(function(i, frame, prev)
    if flip(T.branchProb(depth, i)) then
        -- Theta mean/variance based on avg weighted by
        local theta_mu, theta_sigma = T.estimateThetaD:
        local theta = gaussian(theta_mu, theta_sigma)
        local maxbranchradius = 0.5*(nextframe.center.
        local branchradius = math.min(uniform(0.9, 1),
        local bframe, prev = T.branchFrame(splitFrame,
        branch(bframe, depth+1, prev)
    end
```

Ritchie, Mildenhall, Goodman, Hanrahan [SIGGRAPH’15]
Procedural modelling

Asynchronous function call via future

```python
def future.create(function(i, frame, prev)
    if flip(T.branchProb(depth, i)) then
        -- Theta mean/variance based on avg weighted by
        local theta_mu, theta_sigma = T.estimateThetaD
        local theta = gaussian(theta_mu, theta_sigma)
        local maxbranchradius = 0.5*(nextframe.center
        local branchradius = math.min(uniform(0.9, 1)
        local bframe, prev = T.branchFrame(splitFrame,
        branch(bframe, depth+1, prev)
    end
```

Ritchie, Mildenhall, Goodman, Hanrahan [SIGGRAPH’15]
Captcha solving

Le, Baydin, Wood [2016]
Abstract

We introduce a method for using deep neural networks to amortize the cost of inference in models from the family induced by universal probabilistic programming languages, establishing a framework that combines the strengths of probabilistic programming and deep learning methods. We call what we do “compilation of inference” because our method transforms a denotational specification of an inference problem in the form of a probabilistic program written in a universal programming language into a trained neural network denoted in a neural network specification language. When at test time this neural network is fed observational data and executed, it performs approximate inference in the original model specified by the probabilistic program. Our training objective and learning procedure are designed to allow the trained neural network to be used as a proposal distribution in a sequential importance sampling inference engine. We illustrate our method on mixture models and Captcha solving and show significant speedups in the efficiency of inference.

1 INTRODUCTION

Probabilistic programming uses computer programs to represent probabilistic models (Gordon et al., 2014). Probabilistic programming systems such as STAN (Carpenter et al., 2015), BUGS (Lunn et al., 2000), and Infer.NET (Minka et al., 2014) allow efficient inference in a restricted space of generative models, while systems such as Church (Goodman et al., 2012), Venture (Mansinghka et al., 2014), and Anglican (Wood et al., 2014)—which we call universal—allow inference in unrestricted models. Universal probabilistic programming systems are built upon Turing complete programming languages which support constructs such as higher-order functions, stochastic recursion, and control flow. There has been a spate of recent work addressing the production of artifacts via “compiling away” or “amortizing” inference (in the sense of Gershman and Goodman (2014)). This body of work is roughly organized into two camps. The one in which this work lives, arguably the camp organized around “wake-sleep” (Hinton et al., 1995), is about unsupervised learning of observation-parameterized importance-sampling distributions for Monte Carlo inference algorithms. In this camp, the approach of Paige and Wood (2016) is closest to ours in spirit; they propose learning autoregressive neural density estimation networks to approximate inverse factorizations of graphical models so that at test time, the trained “inference network” starts with the values of all observed quantities and progressively proposes parameters for latent nodes in the original structured model. However, inversion of the dependency structure is impossible in the universal probabilistic program model family, so our approach...
Approximating prob. programs by neural nets.

Le, Baydin, Wood [2016]
Nonparametric Bayesian: Indian buffer process

(define (ibp-stick-breaking-process concentration base-measure)
  (let ((sticks (mem (lambda j (random-beta 1.0 concentration))))
        (atoms (mem (lambda j (base-measure)))))
    (lambda ()
      (let loop ((j 1) (dualstick (sticks 1)))
        (append (if (flip dualstick) ;; with prob. dualstick
                     (atoms j) ;; add feature j
                     '()) ;; otherwise, next stick
                (loop (+ j 1) (* dualstick (sticks (+ j 1))))) )))

Roy et al. 2008
Nonparametric Bayesian: Indian buffer process

(define (ibp-stick-breaking-process concentration base-measure)
  (let ((sticks (mem (lambda j (random-beta 1.0 concentration)))))
    (atoms (mem (lambda j (base-measure))))))
  (lambda ()
    (let loop ((j 1) (dualstick (sticks 1)})
      (append (if (flip) dualstick
tom j)
        (atoms j)
          ;; with prob. dualstick
        (if (flip) 'terminal 'binary-production))
          ;; add feature j
        ;; otherwise, next stick
        (let loop (+ j 1) (* dualstick (sticks (+ j 1))))))))

Lazy infinite array

Roy et al. 2008
Nonparametric Bayesian
Indian buffer process

(define (ibp-stick-breaking-process concentration base-measure)
  (let ((sticks (mem (lambda j (random-beta 1.0 concentration))))
        (atoms (mem (lambda j (base-measure))))
        (lambda ()
          (let loop ((j 1) (dualstick (sticks 1)))
            (append (if (flip dualstick) ;; with prob. dualstick
                       (atoms j) ;; add feature j
                     '()) ;; otherwise, next stick
                    (loop (+ j 1) (* dualstick (sticks (+ j 1)))))))
    ...) | Higher-order parameter

Roy et al. 2008
My research:
Denotational semantics

Joint work with Chris Heunen, Ohad Kammar, Sam Staton, Frank Wood
[LICS 2016]
(let [s (sample (normal 0 10))
     b (sample (normal 0 10))
     f (fn [x] (+ (* s x) b))]

  (observe (normal (f 1) 1) 2.5)
  (observe (normal (f 2) 1) 3.8)
  (observe (normal (f 3) 1) 4.5)
  (observe (normal (f 4) 1) 8.9)
  (observe (normal (f 5) 1) 10.1)

  (predict :sb [s b]))
(let [s (sample (normal 0 10))
   b (sample (normal 0 10))
   f (fn [x] (+ (* s x) b))]

  (observe (normal (f 1) 1) 2.5)
  (observe (normal (f 2) 1) 3.8)
  (observe (normal (f 3) 1) 4.5)
  (observe (normal (f 4) 1) 8.9)
  (observe (normal (f 5) 1) 10.1)

  (predict :sb [s b]))
  (predict :f f)
(let [s (sample (normal 0 10))
    b (sample (normal 0 10))
    f (fn [x] (+ (* s x) b))]
  (observe (normal (f 1) 1) 2.5)
  (observe (normal (f 2) 1) 3.8)
  (observe (normal (f 3) 1) 4.5)
  (observe (normal (f 4) 1) 8.9)
  (observe (normal (f 5) 1) 10.1)

(predict :sb [s b]))
(predict :f f)

Generates a random function of type \( \mathbb{R} \rightarrow \mathbb{R} \). But its mathematical meaning is not clear.
Measurability issue

• Measure theory is the foundation of probability theory that avoids paradoxes.

• Silent about high-order functions.
  • [Halmos] $\text{ev}(f,a) = f(a)$ is not measurable.
  • The category of measurable sets is not CCC.

• But Anglican supports high-order functions.
Use category theory to extend measure theory.
Use category theory to extend measure theory.

The diagram shows a category labeled `Meas` connected by arrows to the category `[Meas^{op}, Set]_\Pi` via a Yoneda Embedding. There is also a Monad transformation that maps `Meas` to `[Meas^{op}, Set]_\Pi` through another Yoneda Embedding.
Use category theory to extend measure theory.
Use category theory to extend measure theory.
Use category theory to extend measure theory.

```
Meas  ↪ Yoneda Embedding  \[\text{Meas}^\text{op}, \text{Set}\]_\prod
        ↘
Meas  ↪ Yoneda Embedding  \[\text{Meas}^\text{op}, \text{Set}\]_\prod
```

Enough structure for function types

Left Kan Extension
Use category theory to extend measure theory.

Monad

Yoneda Embedding

$\text{Meas} \rightarrow [\text{Meas}^{\text{op}}, \text{Set}]_\Pi$

Enough structure for function types

Yoneda Embedding

Preserves nearly all the structures

Meas

Monad

$\text{Meas} \rightarrow [\text{Meas}^{\text{op}}, \text{Set}]_\Pi$

Left Kan Extension
[Question] Are all definable functions from R to R in a high-order probabilistic PL measurable?

Our semantics says that the answer is yes for a core call-by-value language, such as Anglican.
The monad $\mathbb{M}(\mathbb{R} \to \mathbb{R})$ at $\mathbb{R} \to \mathbb{R}$ consists of:

- equivalence classes of measurable functions $f : \Omega \times \mathbb{R} \to \mathbb{R}$ for probability spaces $\Omega$.

The function $f$ is what probabilists call a measurable stochastic process.
The extended monad $\mathcal{M}$ describes computations with dynamically allocated read-only variables.

$$\mathcal{M}(T)(w) = \{ [(a, f)]_{\sim} \mid \exists v. a \in T(v) \land f : w \to^m \text{Prob}(v) \}$$
The extended monad $\mathbb{M}$ describes computations with dynamically allocated read-only variables.

\[
\mathbb{M}(T)(w) = \\
\{ [(a, f)]_\sim \mid \exists v. \ a \in T(v) \land f : w \rightarrow_m \text{Prob}(v) \}
\]

$T$ is the type of a value.
The extended monad $\mathcal{M}$ describes computations with dynamically allocated read-only variables.

$$\mathcal{M}(T)(w) = \{ [(a, f)]_\sim \mid \exists v. \ a \in T(v) \land f : w \rightarrow_m \text{Prob}(v) \}$$

$T$ is the type of a value.

$w$ represents a space of all random vars so far.
The extended monad $\mathcal{M}$ describes computations with dynamically allocated read-only variables.

$\mathcal{M}(T)(w) = \{ [(a, f)] \sim | \exists v. a \in T(v) \land f : w \to^m \text{Prob}(v) \}$

$T$ is the type of a value.  
$w$ represents a space of all random vars so far.  
$v$ extends $w$ with new random variables according to $f$.  

Try a probabilistic prog. language. It is fun.

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

- WebPPL: