Probabilistic Programming

Mint Tanprasert and Lana Sodol
Probability basics

- X and Y are random variables that follow some distribution
- X is a model, Y are observations
- Marginal p(X) prior probability
- Marginal p(Y) evidence probability
- Conditional p(X|Y) posterior probability
- Conditional p(Y|X) likelihood
- Joint p(X, Y) = p(Y|X)*p(X)
Conditional Probability and Bayes Rule

- \( p(X|Y) = \frac{p(X,Y)}{p(Y)} \)
- \( p(Y|X) = \frac{p(X,Y)}{p(X)} \)
- Bayes Rule: \( p(X|Y) = p(Y|X) \times \frac{p(X)}{p(Y)} \)
- Bayes rule is at the heart of inference

\[
P(H|X) = P(H) \times \left(1 + P(C) \times \left(\frac{P(x|H)}{P(x)} - 1\right)\right)
\]

**H: HYPOTHESIS**
**X: OBSERVATION**
**P(H): PRIOR PROBABILITY THAT H IS TRUE**
**P(x): PRIOR PROBABILITY OF OBSERVING X**
**P(C): PROBABILITY THAT YOU'RE USING BAYESIAN STATISTICS CORRECTLY**
Marginalization

\[
P(a) = \sum_b P(a, b) = \sum_b P(a \mid b) P(b)
\]

\[
P(b) = \sum_a \sum_c \sum_d P(a, b, c, d)
\]

\[
P(c \mid b) = \sum_a \sum_d P(a, c, d \mid b) = \left(1 / P(b)\right) \sum_a \sum_d P(a, c, d, b)
\]

Can get any marginal or conditional probability of interest from the joint
Factorization

\[ P(a, b, c, \ldots, z) = P(a \mid b, c, \ldots, z) \ P(b, c, \ldots, z) \]

\[ P(a, b, c, \ldots, z) = P(a \mid b, c, \ldots, z) \ P(b \mid c, \ldots, z) \ P(c \mid \ldots, z) \ldots P(z) \]

Chain rule for probability
Holds for any variable ordering
Sprinkler and Rain are parents of WetGrass

Value of each child depends on the values of its parents

Structure the network such that all children are conditionally dependent on parents

And order factors in most efficient way

\[ p(X_1, X_2, \ldots, X_N) = \prod p(X_i \mid \text{parents}(X_i)) \]
Factor graphs

- Probabilities have to be normalized (sum to 1)
- Simplify by having the conditional probabilities as factors - unnormalized densities
- Graph becomes undirected
Graphical models: Markov Chain

Parents are the previous states in time
Markov assumption: the future is independent of the past given the present
Stationary assumption: the probability of moving from a state to the next is the same for all states

Need to specify prior distribution $P(s_0)$
And proposal distribution $P(s' | s)$ where $s'$ is the new state
(this distribution proposes how we will move to the next state)
Problem

We toss 2 fair coins. Given that not both are heads, what is the probability that the first coin is heads?

Answer: 1/3

All combinations: TT HT TH HH
Combinations with condition: TT HT TH
in 1/3 of the conditioned combinations the first coin is heads
How could we solve this?

- Bernoulli distribution: \( p(\text{coin1})p(\text{coin2})p(\text{bothHeads}|\text{coin1, coin2}) \)
- Graphical model
- Simulation:
  ```cpp
  bool coin1 = random() < 0.5
  bool coin2 = random() < 0.5
  bool bothHeads = coin1 & coin2
  ```
How would we solve this?

- Mathematical approach
- Simulation approach: rejection sampling
- Inference on graphical model

HARD

TIME CONSUMING

PROBABILISTIC PROGRAMMING!
Probabilistic Programming

THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."

HEY! GET BACK TO WORK!

COMPILING!

OH. CARRY ON.
Probabilistic Programming

Compiles model into graph
Probabilistic Programming

Intuition

Parameters

Program

Output

Parameters

Program

Observations

$p(x|y)$

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

$y$

Inference

CS

Probabilistic Programming

Statistics
Probabilistic programming

● Take your favorite language

● Add *randomness* to have probabilistic behaviour

● Add *constraints* to condition on observed data

● Indicate which variables’ distributions are to be *inferred*

● (Andrew Gordon)
First Order Probabilistic Programming Language

\[ v ::= \text{variable} \]
\[ c ::= \text{constant value or primitive operation} \]
\[ f ::= \text{procedure} \]
\[ e ::= c \mid v \mid (\text{let} \ [v \ e_1] \ e_2) \mid (\text{if} \ e_1 \ e_2 \ e_3) \]
\[ \quad \mid (f \ e_1 \ldots \ e_n) \mid (c \ e_1 \ldots \ e_n) \]
\[ \quad \mid (\text{sample} \ e) \mid (\text{observe} \ e_1 \ e_2) \]
\[ q ::= e \mid (\text{defn} \ f \ [v_1 \ldots \ v_n] \ e) \ q \]
Syntax

- Have primitives for function distributions like `normal` and `discrete`
- Arithmetic `+`, `-`, `*`, `/`
- Vectors representing data
  
  \[
  \text{(vector } e_1 \ldots e_N) = [e_1 \ldots e_N]
  \]

- Hash-maps as sequence of key-value pairs
  
  \[
  \text{(hash-map } e_1 e_1' \ldots e_N e_N') = \{e_1 e_1' \ldots e_N e_N'\}
  \]
Let it Sugar

\[
(\text{let } [v_1 \ e_1 \\
\vdots \\
v_n \ e_n] \\
e_{n+1} \ldots e_{m-1} \ e_m)
\]

\[
(\text{let } [v_1 \ e_1] \\
\vdots \\
(\text{let } [v_n \ e_n] \\
(\text{let } [\_ \ e_{n+1}] \\
\vdots \\
(\text{let } [\_ \ e_{m-1}] \\
e_m)\ldots))))
\]

will be expanded by generating some fresh variable symbol, say x284xu,

\[
(\text{let } [x284xu \ (\text{observe } (\text{normal} \ 0 \ 1) \ 2.0)] \ldots)
\]
Sugar one more time

```
(foreach c
    [v_1 e_1 ... v_n e_n]
    e'_1 ... e'_k)
```

```
(vector
    (let [v_1 (get e_1 0)
          ...
          v_n (get e_n 0)]
        e'_1 ... e'_k)
    ...
    (let [v_1 (get e_1 (- c 1))
          ...
          v_n (get e_n (- c 1))]
        e'_1 ... e'_k))
```

```
(loop c e f e_1 ... e_n)
```

```
(let [a_1 e_1
      a_2 e_2
      ...
      a_n e_n]
  (let [v_0 (f 0 e a_1 ... a_n)]
    (let [v_1 (f 1 v_0 a_1 ... a_n)])
```
0-th order expressions

- Sub-language for purely deterministic computations
- No randomness in these

\[
c ::= \text{constant value or primitive operation} \\
v ::= \text{variable} \\
E ::= c \mid v \mid (\text{if } E_1 \ E_2 \ E_3) \mid (c \ E_1 \ldots \ E_n)
\]
Compilation to graph

- Mapping from procedure names to definitions
- Logical predicate for control flow context
- Expression we need to compile (source code)

- graphical model $G$
- expression $E$ in the deterministic sublanguage (no sample or observe)
  describes return value of original expression in terms of random variables in $G$
Example

- 2-component Gaussian mixture
- Single observation
- Defines a joint distribution \( p(y = 0.5, z) \)
- The problem is to characterize the posterior distribution \( p(z|y) \)
Compilation to graph

(let [z (sample (bernoulli 0.5))
   mu (if (= z 0) -1.0 1.0)
   d (normal mu 1.0)
   y 0.5]
(observe d y)
2)

\[
G = (V, A, P, \mathcal{Y})
\]

\[
V = \{z, y\},
A = \{(z, y)\},
P = [z \mapsto (p_{bern} z 0.5),
y \mapsto (p_{norm} y (if (= z 0) -1.0 1.0) 1.0)],
\mathcal{Y} = [y \mapsto 0.5]
E = z
\]
Deterministic Computation - no new vertices

\[
\rho, \phi, c \downarrow G_{\text{emp}}, c \\
\rho, \phi, z \downarrow G_{\text{emp}}, z
\]

\[
\rho, \phi, e_1 \downarrow G_1, E_1 \\
\rho, \phi, e_2[v := E_1] \downarrow G_2, E_2 \\
\rho, \phi, (\text{let } [v \ e_1] \ e_2) \downarrow (G_1 \oplus G_2), E_2
\]

\[
\rho, \phi, e_1 \downarrow G_1, E_1 \\
\rho, (\text{and } \phi E_1), e_2 \downarrow G_2, E_2 \\
\rho, (\text{and } \phi (\text{not } E_1)), e_3 \downarrow G_3, E_3 \\
\rho, \phi, (\text{if } e_1 \ e_2 \ e_3) \downarrow (G_1 \oplus G_2 \oplus G_3), (\text{if } E_1 \ E_2 \ E_3)
\]
\[
\begin{align*}
\rho, \phi, e \downarrow (V, A, \mathcal{P}, \mathcal{Y}), E & \quad \text{Choose a fresh variable } v \\
Z = \text{FREE-VARS}(E) \cap V & \quad F = \text{SCORE}(E, v) \neq \bot \\
\rho, \phi, \text{(sample } e) \downarrow (V \cup \{v\}, A \cup \{(z, v) : z \in Z\}, \mathcal{P} \oplus [v \mapsto F], \mathcal{Y}), v
\end{align*}
\]

\[
\text{SCORE}(\text{if } E_1 E_2 E_3, v) = \text{if } E_1 F_2 F_3 \\
\quad \text{(when } F_i = \text{SCORE}(E_i, v) \text{ for } i \in \{2, 3\} \text{ and it is not } \bot) \\
\text{SCORE}(c E_1 \ldots E_n, v) = (p_c v, E_1 \ldots E_n) \\
\quad \text{(when } c \text{ is a constructor for distribution and } p_c \text{ its pdf or pmf)} \\
\text{SCORE}(E, v) = \bot \\
\quad \text{(when } E \text{ is not one of the above cases)}
\]

\[
\begin{align*}
\rho, \phi, e_1 \downarrow G_1, E_1 & \quad \rho, \phi, e_2 \downarrow G_2, E_2 \\
(V, A, \mathcal{P}, \mathcal{Y}) = G_1 \oplus G_2 & \quad \text{Choose a fresh variable } v \\
F_1 = \text{SCORE}(E_1, v) \neq \bot & \quad F = \text{if } \phi F_1 1 \\
Z = (\text{FREE-VARS}(F_1) \setminus \{v\}) \cap V & \quad \text{FREE-VARS}(E_2) \cap V = \emptyset \\
B = \{(z, v) : z \in Z\} & \\
\rho, \phi, \text{(observe } e_1 e_2) \downarrow (V \cup \{v\}, A \cup B, P \oplus [v \mapsto F], \mathcal{Y} \oplus [v \mapsto E_2]), E_2
\end{align*}
\]
\[\rho, \phi, e_i \downarrow G_i, E_i \text{ for all } 1 \leq i \leq n\]
\[\rho(f) = (\text{defn } f \ [v_1 \ldots v_n] \ e)\]

\[\rho, \phi, e[v_1 := E_1, \ldots v_n := E_n] \downarrow G, E\]

\[\rho, \phi, (f \ e_1 \ldots e_n) \downarrow G_1 \oplus \ldots \oplus G_n \oplus G, E\]

\[\rho, \phi, e_i \downarrow G_i, E_i \text{ for all } 1 \leq i \leq n\]

\[\rho, \phi, (c \ e_1 \ldots e_n) \downarrow G_1 \oplus \ldots \oplus G_n, (c \ E_1 \ldots E_n)\]
Partial Evaluation

- Optimize by evaluating E expressions
- Same values in every execution
- Can collapse edges

\[
\begin{align*}
\rho, \phi, e_1 \downarrow G_1, E_1 & \quad \rho, \text{EVAL((} \text{and} \phi E_1)) , e_2 \downarrow G_2, E_2 \\
\rho, \text{EVAL((} \text{and} \phi (\text{not} E_1))) , e_3 \downarrow G_3, E_3 \\
\rho, \phi, (\text{if } e_1 e_2 e_3) \downarrow (G_1 \oplus G_2 \oplus G_3), \text{EVAL((if } E_1 E_2 E_3))
\end{align*}
\]

and

\[
\begin{align*}
\rho, e_i \downarrow G_i, E_i \text{ for all } 1 \leq i \leq n \\
\rho, \phi, (c e_1 \ldots e_n) \downarrow G_1 \oplus \ldots \oplus G_n, \text{EVAL((c } E_1 \ldots E_n))
\end{align*}
\]

\[\text{SCORE}(c, v) = (p_c v)\]

(when \(c\) is a distribution and \(p_c\) is its pdf or pmf)
Evaluation-based Inference

- **Problem:** What if the graph is dynamic?
- **Solution:** Don’t construct the graph at all!
  - We can’t characterize the probabilistic program, but we can ensure that we get the samples from the same distribution when running the program.

Can we have unbounded number of variables now?
Example: Likelihood Weighting

- **Importance sampling**: approximate posterior distribution $p(x|y)$ with a set of weighted samples given $p(x, y)$ and a proposal distribution $q(x|y)$

\begin{align*}
  x^{(m)} &\sim q(x|y) & m = 1, \ldots, M \\
  w_m &= p(x^{(m)}, y)/q(x^{(m)}|y) & m = 1, \ldots, M \\
  W_m &= w_m/\sum_j w_j & m = 1, \ldots, M
\end{align*}

\[ p(x|y) \approx \sum_{m=1}^{M} W_m \delta(x - x^{(m)}) \]

[Le, Baydin, Wood, 2016]

- It is only efficient when the proposal distribution matches target distribution
- Likelihood weighting is a special (and simplest) case where $q(x|y) = p(x)$. 
Likelihood Weighting

\[
\begin{align*}
\ell(v) &= c & \rho, \ell, e_1 \downarrow c_1, l_1 & \rho, \ell \oplus [v_1 \mapsto c_1], e_0 \downarrow c_0, l_0 \\
\rho, \ell, c &\downarrow c, 0 & \rho, \ell, v \downarrow c & \rho, \ell, (\text{let } [v_1 \ e_1] \ e_0) \downarrow c_0, l_0 + l_1 \\
\rho, \ell, e_1 \downarrow \text{true}, l_1 & \rho, \ell, e_2 \downarrow c_2, l_2 & \rho, \ell, c_3 \downarrow c_3, l_3 \\
\rho, \ell, (\text{if } e_1 \ e_2 \ e_3) \downarrow c_2, l_1 + l_2 & \rho, \ell, (\text{if } e_1 \ e_2 \ e_3) \downarrow c_3, l_1 + l_3 \\
\rho(f) &= [v_1, \ldots, v_n], e_0 & \rho, \ell, e_i \downarrow c_i, l_i & \text{for } i = 1, \ldots, n \\
\rho, \ell \oplus [v_1 \mapsto c_1, \ldots, v_n \mapsto c_n], e_0 \downarrow c_0, l_0 & \rho, \ell, (f \ e_1 \ldots \ e_n) \downarrow c_0, l_0 + l_1 + \ldots + l_n \\
\rho, \ell, e_i \downarrow c_i, l_i & \text{for } i = 1, \ldots, n & c(c_1, \ldots, c_n) = c_0 \\
\rho, \ell, (c \ e_1 \ldots \ e_n) \downarrow c_0, l_1 + \ldots + l_n \\
\rho, \ell, e \downarrow d, l & \ c \sim d & \rho, \ell, e_1 \downarrow d_1, l_1 & \rho, \ell, e_2 \downarrow c_2, l_2 & \log p_{d_1}(c_2) = l_0 \\
\rho, \ell, (\text{sample } e) \downarrow c, l & \rho, \ell, (\text{observe } e_1 \ e_2) \downarrow c_2, l_0 + l_1 + l_2 \\
\end{align*}
\]
How to do this with graph-based inference?

1. For each $x \in X$: sample from the prior $x^l \sim p(x | \text{PA}(x))$.

2. For each $y \in Y$: calculate the weights $W^l_y = p(y | \text{PA}(y))$.

3. Return the weighted set of return values $r(X^l)$

$$
\sum_{l=1}^{L} \frac{W^l}{\sum_{k=1}^{L} W_k} \delta_{r(X^l)}; W^l := \prod_{y \in Y} W^l_y.
$$

where $\delta_x$ denotes an atomic mass centered on $x$. 

[Wood 2018]
How to do this with evaluation-based inference?

Initialize a state variable $\sigma$ with $\log W = 0$.

1. Sample a value $x \sim d$ when encounter an expression (sample $d$)
2. Calculate the log likelihood $\log p_d(y)$ when encounter an expression (observe $d \ y$)
3. Update the log weight: $\log W \leftarrow \log W + \log p_d(y)$

[Wood 2018]
Implementation

- We want to evaluate a program $e$ to constant $c$.
- **sample** and **observe** are the only expressions that have algorithm-specific evaluations.
- Everything else is deterministic and is the same for any algorithm.

```
Algorithm 7 Evaluation-based likelihood weighting
1: global $\rho, e$  \hspace{1em} \triangleright \text{Program procedures, body}
2: function EVAL($e, \sigma, \ell$)
3: \hspace{1em} match $e$
4: \hspace{1.5em} case (sample $e$)
5: \hspace{2em} $d, \sigma \leftarrow \text{EVAL}(e, \sigma, \ell)$
6: \hspace{2em} return SAMPLE($d$), $\sigma$
7: \hspace{1.5em} case (observe $c_1\ c_2$)
8: \hspace{2em} $d_1, \sigma \leftarrow \text{EVAL}(e_1, \sigma, \ell)$
9: \hspace{2em} $c_2, \sigma \leftarrow \text{EVAL}(e_2, \sigma, \ell)$
10: \hspace{2em} $\sigma(\log W) \leftarrow \sigma(\log W) + \text{LOG-PROB}(d_1, c_2)$
11: \hspace{2em} return $c_2, \sigma$
12: \hspace{1em} ...  \hspace{1em} \triangleright \text{Base cases (as in Algorithm 6)}
13: function LIKELIHOOD-WEIGHTING($L$)
14: \hspace{1em} $\sigma \leftarrow [\log W \mapsto 0]$
15: \hspace{1em} for $i$ in 1, ..., $L$ do  \hspace{1em} \triangleright \text{Initialize state}
16: \hspace{2em} $r^i, \sigma^i \leftarrow \text{EVAL}(e, \sigma, [])$  \hspace{1em} \triangleright \text{Run program}
17: \hspace{2em} $\log W^i \leftarrow \sigma(\log W)$  \hspace{1em} \triangleright \text{Store log weight}
18: return $((r^1, \log W^1), \ldots, (r^L, \log W^L))$

[Wood 2018]
```
Important characteristics of FOPPL

- We can only called primitive function when we define a new function
  - No passing functions/higher-order functions
  - No recursion
- No I/O; all data must be inlined as code
- Finite number of random variables
- Intermediate representation (IR) language, not production language
Why is FOPPL not enough?

“Ultimately, we want to be able to do probabilistic programming using any existing programming language as the modeling language. So, we need to be able to deal with all possible models that could be written and in general, we won’t be able to syntactically prohibit stochastic loop bounds or conditioning data whose size is only known at runtime.”

[Wood 2018]
HOPPL (Higher-order Probabilistic PL)

\begin{align*}
v & ::= \text{variable} \\
c & ::= \text{constant value or primitive operation} \\
f & ::= \text{procedure} \\
e & ::= c \mid v \mid f \mid (\text{if } e \ e \ e) \mid (e \ e_1 \ldots e_n) \mid (\text{sample } e) \\
& \quad \mid (\text{observe } e \ e) \mid (\text{fn } [v_1 \ldots v_n] \ e) \\
q & ::= e \mid (\text{defn } f \ [v_1 \ldots v_n] \ e) \ q.
\end{align*}

**Language 5.4:** Higher-order probabilistic programming language (HOPPL)
A brief revisit to likelihood weighting...

What if we implement \texttt{EVAL()} for HOPPL?

**Dynamic addressing transformation**

- Assigns unique identifier (\textit{address}) to each instance of sample or observe that can be reached during execution
- Use these addresses to implement inference

```
Algorithm 16 Inference controller for Likelihood Weighting
1: repeat
2:   for \( \ell = 1, \ldots, L \) do
3:     \( \sigma \leftarrow \text{NEWID()} \)
4:     \( \log W_\sigma \leftarrow 0 \)
5:     \( \text{SEND("start", } \sigma) \)
6:     \( l \leftarrow 0 \)
7:   while \( l < L \) do
8:     \( \mu \leftarrow \text{RECEIVE()} \)
9:     switch \( \mu \) do
10:       case (*sample*, \( \sigma, \alpha, d \))
11:         \( x \leftarrow \text{SAMPLE}(d) \)
12:         \( \text{SEND(*continue*, } \sigma, x) \)
13:       case (*observe*, \( \sigma, \alpha, d, c \))
14:         \( \log W_\sigma \leftarrow \log W_\sigma + \text{LOG-PROB}(d, c) \)
15:         \( \text{SEND(*continue*, } \sigma, c) \)
16:       case (*return*, \( \sigma, c \))
17:         \( l \leftarrow l + 1 \)
18:         \( \text{OUTPUT}(c, \log W_\sigma) \)
19:     until forever
```

[Wood 2018]
Important characteristics of HOPPL

- Loop bound can be specified at runtime
  - It’s actually implemented with recursion
- No translation to graphical models
  - Can only evaluate the probability of a trace/sample
- If you’re using unmodified existing language as modeling language (Python, C++, Javascript, etc.), you are in the HOPPL land.
Some example languages

Graphical Models
- BUGS
- STAN

Factor Graphs
- Factorie
- Infer.NET

Infinite Dimensional Parameter Space Models

Unsupervised Deep Learning
- Anglican
- WebPPL
- PYRO
- ProbTorch
- Edward

[Wood, 2018]
Application
Application: Procedural Modeling

Figure 1: Controlling the output of highly-variable procedural modeling programs using our Stochastically-Ordered Sequential Monte Carlo algorithm. Here, the controls encourage volumetric similarity to a target shape (shown in black).
Stochastically-Ordered Sequential Monte Carlo (SOSMC)

- SMC receives feedback and improves incrementally (doesn’t need to score on completely generated models.)
- There are many possible way of improving. We can’t know a priori which order is most optimal.

- **Solution**: SOSMC randoms the order according to a likelihood measure under some “ordering policy.”

[Ritchie et al, 2015]
Constrained Modeling

Conditioning with hard constraints like these transforms the prior distribution on trees into a posterior distribution in which all posterior trees conform to the constraint e.g. branch not touching obstacles, volume matching, etc.

[Ritchie et al, 2015]
Application: Captcha Breaking

1. Sample a string.
2. Generate an image.
Generative Model

;; Define a function to sample a single character
(defn sample-char []
  (sample (uniform ["a" "b" ... "z"
                   "A" "b" ... "Z"
                   "0" "1" ... "9"])))

;; Define a function to generate a Captcha
(defn generate-captcha [text]
  (let [char-rotation (sample (normal 0 1))
        add-distortion? (sample (flip 0.5))
        add-lines? (sample (flip 0.5))
        add-background? (sample (flip 0.4))]
    ;; Render a Captcha image
    (render text char-rotation
             add-distortion? add-lines? add-background?)
    and then to perform inference on the text

(let [image ( ... ) ;; read target Captcha from disk
       num-chars (sample (poisson 4))
       text (repeatedly num-chars sample-char)
       generated (generate-captcha text)]
  ;; score using any image similarity measure
  (factor (image-similarity image generated))
  text)

[Wood 2018]
Probabilistic Programming vs Machine Learning

- This algorithm is just a Bayesian inference interpretation of training a neural network model with synthetic data!
  - Actually, this paper trains a deep learning model...

[Le, Baydin, Wood, 2016]
Probabilistic Programming vs Machine Learning

● Bayesian inference → Neural net
  ○ Explain synthetic data mismatch with probability distribution
● Neural net → Bayesian inference
  ○ Create probabilistic model of synthetic data generator to account for uncertainty in output
  ○ Get probability that’s more accurate than just the final softmax layer
Result 1: Synthetic data generator

![Graphs showing model mismatch]

Fig. 3. Illustration of model mismatch. Left: The model encompasses the true data distribution; Middle: the model partially matches the true data distribution; Right: the model is completely mismatched to the true data distribution.

[Le, Baydin, Wood, 2016]
Result 2: Approximate posteriors

[Le, Baydin, Wood, 2016]
Summary

- Probabilistic languages unify general purpose programming with probabilistic modelling
- Users specify model, inference follows automatically
- Full power to specify any complex models
- Reuse of libraries of models, support interactive modeling and formal verification
- Provide a much-needed abstraction barrier to foster generic, efficient inference in universal model classes
References


- Hongseok Yang’s lecture, KAIST, 2017
  https://github.com/hongseok-yang/probprog17?fbclid=IwAR0LL3ZIFzLoR8IyVeIGQS-XpQpxJKer7m9V8dCkfveLISoR998mhUjyWc(KAIST)

- Andrew Gordon lecture
  https://www.cs.uoregon.edu/research/summerschool/summer19/topics.php

- Probabilistic-Programming.org