Programs as probabilistic models
Generative models?

Can you write a program to do this?
Forward models are “easy”

Can you write a program to solve Sudoku problems?

Can you write a program to generate Sudoku problems?
Model relationships between many variables

recent travel abroad?

arrows: causal relationships
circles: random variables

Lauritzen & Spiegelhalter, 1988
Quantification of uncertainty
Motivation: models in machine learning

Data (input)

Generative Model (assumptions)

Inference (output)

Prediction Task

\[ \mu_k, \Sigma_k \sim \text{NormalWishart}(\psi) \]
\[ z_n \sim \text{Discrete}(\pi) \]
\[ y_n \sim \text{Normal}(\mu_{z_n}, \Sigma_{z_n}) \]

Gibbs Sampler

Expectation Maximization
Motivation: models in machine learning

Data (input)

Generative Model (assumptions)

Inference (output)

Machine Learning Software

Prediction Task

\[
\begin{align*}
\mu_k, \Sigma_k &\sim \text{NormalWishart}(\psi) \\
z_n &\sim \text{Discrete}(\pi) \\
y_n &\sim \text{Normal}(\mu_{z_n}, \Sigma_{z_n})
\end{align*}
\]

Gibbs Sampler

Expectation Maximization

(Model-Specific)
Motivation: models in machine learning

Data (input)

Generative Model (assumptions)

Inference (output)

Probabilistic Programming System

Prediction Task → Modeling Language → Inference Back End

(Model-Agnostic)
Intuitive view of probabilistic programming

Programming
- Parameters
  - Algorithm
    - Output

Statistics
- Random variables
  - Model
    - Data

Probabilistic Programming
- Random variables
  - Algorithm (Model)
    - Output (Data)

Inference
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
Why would we do this?

Question: Why are you writing a probabilistic programming language?

Answer 1: I’m really tired of writing the same inference code again and again for each new model!

Answer 2: I have a probabilistic model I can simulate from, but I have no idea how to condition it on data!
An example BUGS program

\[ x \sim \mathcal{N}(a, b^{-1}) \]
\[ y_i \sim \mathcal{N}(x, c^{-1}), \quad i = 1, \ldots, N \]

```r
model {
  x ~ dnorm(a, 1/b)
  for (i in 1:N) {
    y[i] ~ dnorm(x, 1/c)
  }
}
```

Language restrictions?
Model class?
Inference?

Spiegelhalter et al. "BUGS: Bayesian inference using Gibbs sampling"
An example BUGS program

Loop iterations are **deterministic**!

No **if** statement (no branching)

```
# data
list(t = c(94.3, 15.7, 62.9, 126, 5.24, 31.4, 1.05, 1.05, 2.1, 10.5),
     y = c(5, 1, 5, 14, 3, 19, 1, 1, 4, 22),
     N = 10)
# inits
list(a = 1, b = 1)
# model
{
  for (i in 1 : N) {
    theta[i] ~ dgamma(a, b)
    l[i] <- theta[i] * t[i]
    y[i] ~ dpois(l[i])
  }
  a ~ dexp(1)
  b ~ dgamma(0.1, 1.0)
}
```

**Program 2.7:** The Pumps example model from BUGS (OpenBugs, 2009).
### “Inference first” approach to PPLs

“I never want to write this inference code again!”

<table>
<thead>
<tr>
<th>Inference</th>
<th>Models</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gibbs Sampling</td>
<td>Finite graphical models</td>
<td>Bounded loops; no branching</td>
</tr>
<tr>
<td>Hamiltonian Monte Carlo</td>
<td>Continuous latent variables</td>
<td>Bounded loops; no discrete r.v.s</td>
</tr>
<tr>
<td>Expectation Propagation</td>
<td>Factor graphs</td>
<td>Finite composition of factors</td>
</tr>
</tbody>
</table>
Pros: these languages work.
Cons?
**Example: “Anglican”**

*Anglican* is a Turing-complete probabilistic programming language embedded in Clojure.

*(Disclaimer: I helped work on developing it back when I was at Oxford)*

Other similar (and probably more current) projects:

* [turing.jl](#) (Cambridge), *gen* (MIT), *Birch*, *PyProb* (UBC), *webPPL*, …
Syntax: basically Clojure (similar to 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)
```

• Branching


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

• Functions are first class

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

• Local bindings

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

- map

  ;; Apply the function f(x,y) = x + 2y to the
  ;; x values [1 2 3] and the y values [10 9 8]
  ;; Produces [21 20 19]
  (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

  ;; Reduce recursively applies function,
  ;; to result and next element, i.e.
  (reduce + 0 [1 2 3 4])
  ;; does (+ (+ (+ 0 1) 2) ...  
  ;; and evaluates to 10
The need for higher-order languages

Unfortunately, restrictions can be quite limiting!

**Simple example:** sampling from a geometric distribution, by counting number of failures before first success, in independent Bernoulli trials

```
(defn sample-geometric [p]
  (if (sample (flip p))
    0
    (+ 1 (sample-geometric p)))))
```
Other way around: language first

Unrestricted Languages:

• “Open-universe”: unbounded numbers of parameters
• Mixed variable types
• Access to existing software libraries
• Easily extensible

What is the catch?

• Inference is going to be harder
• More ways to shoot yourself in the foot
Bayesian inference

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

Posterior, Likelihood, Prior

\[ E_{p(x \mid y)}[Q(x)] \]

Estimate predict values, under posterior on sample values, given observe values.
Bayesian inference

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

Example: Biased Coin

- \( y \): Observed data (flip outcomes)
- \( x \): Unknown variable (coin bias)
Bayesian inference

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

- **Posterior**
- **Likelihood**
- **Prior**

**Example: Biased Coin**

- \( p(y \mid x) \): Likelihood of outcome given bias
- \( p(x) \): Prior belief about bias
- \( p(x \mid y) \): Posterior belief after seeing data
Bayesian inference

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

Example: Biased Coin

<table>
<thead>
<tr>
<th>x (bias)</th>
<th>p(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td>0.2</td>
<td>0.5</td>
</tr>
<tr>
<td>0.4</td>
<td>0.5</td>
</tr>
<tr>
<td>0.6</td>
<td>0.5</td>
</tr>
<tr>
<td>0.8</td>
<td>0.5</td>
</tr>
<tr>
<td>1.0</td>
<td>0.5</td>
</tr>
</tbody>
</table>

0 heads, 0 tails
Bayesian inference

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

Posterior, Likelihood, Prior

Example: Biased Coin

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

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

Example: Biased Coin
Bayesian inference

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

Example: Biased Coin

- 24 heads, 26 tails
- 16 heads, 14 tails
- 7 heads, 3 tails

\[ p(x \mid y) \]

\[ x \] (bias)
Separating models and inference

Modeling Language (Anglican)

(let [bias (sample (uniform 0 1))]
  likelihood (flip bias))
(observe likelihood true)
(observe likelihood true)
(observe likelihood true)
(predict bias))

Special Forms

1 sample random value x
2 observe condition on value y
3 return value Q(x)

Inference Back End

Estimate distribution over output values under posterior of sample values, given observe values.

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

- Implements (inference-algorithm-specific) sample and observe handlers
- Returns weighted samples
Generative model for Captcha-breaking

Target Image

Model for Characters

(defn sample-char []
{:symbol (sample (uniform ascii))
 :x (sample (uniform-cont 0.0 1.0))
 :y (sample (uniform-cont 0.0 1.0))
 :scale (sample (beta 1 2))
 :weight (sample (gamma 2 2))
 :blur (sample (gamma 1 1))})
Generative model for Captcha-breaking

Target Image

Samples from Program

Model for Characters

```
(def query captcha
  [image max-chars tol]
  (let [[[w h] (size image)]
        ;; sample random characters
        num-chars (sample
                    (uniform-discrete
                     1 (inc max-chars)))
        chars (repeatedly
               num-chars sample-char)]
      ;; compare rendering to true image
      (map (fn [y z]
             (observe (normal z tol) y))
            (reduce-dim image)
            (reduce-dim (render chars w h)))
      ;; output captcha text
      (map :symbol (sort-by :x chars)))))
```
Generative model for Captcha-breaking

Model for Characters

(defquery captcha
  [image max-chars tol]
  (let [[w h] (size image)]
    ;; sample random characters
    (num-chars (sample
      (uniform-discrete
        1 (inc max-chars))))
    (chars (repeatedly
      num-chars sample-char)])
    ;; compare rendering to true image
    (map (fn [y z]
      (observe (normal z tol) y))
      (reduce-dim image)
      (reduce-dim (render chars w h))))
    ;; output captcha text
    (map :symbol (sort-by :x chars))))
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)))

What if we want a “world” that puts ~20% of balls in box?
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)))]
    ;; 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)))

    ;; 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 4 0.1)]

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

  (predict :balls balls)
  (predict :bumper-positions bumper-positions)))
Other sorts of examples

- Coordination game: cell phone dead. Do we meet at the cafe, or meet at the pub?
  - Alice simulates Bob’s decision process
    - … which simulates Alice’s decision process …
      - … which simulates Bob’s decision process …
        - …
  - Mutually recursive functions! Easy to write as functional programming code, very annoying to write out as an explicit game tree…
How can we perform inference?

- Two special forms are the entire interface between model code and inference code:
  
  \[(\text{sample } \ldots) \quad (\text{observe } \ldots)\]

- \textbf{Q:} what kinds of inference algorithms can we develop and implement using \textbf{just this} as our interface?
Inference over partial program executions

From the perspective of the inference engine, what happens as a program runs?

- Sequence of $M$ sample statements $\{(f_j, \theta_j)\}_{j=1}^M$
- Sequence of $N$ observe statements $\{(g_i, \phi_i, y_i)\}_{i=1}^N$
- Sequence of $M$ sampled values $\{x_j\}_{j=1}^M$
- Conditioned on these sampled values the entire computation is deterministic

$$\gamma(x) \triangleq p(x, y) = \prod_{i=1}^N g_i(y_i|\phi_i) \prod_{j=1}^M f_j(x_j|\theta_j).$$
designed to ensure that programs always evaluate a bounded set of sample and observe expressions. Because of this, programs that are written in the FOPPL can be safely eagerly evaluated. It is very easy to create a language in which this is no longer the case. For example, if we simply allow function definitions to be recursive, then we can now write programs such as this one,

```clojure
(defn sample-geometric [alpha]
  (if (= (sample (bernoulli alpha)) 1)
    1
    (+ 1 (sample-geometric p))))

(let [alpha (sample (uniform 0 1))
      k (sample-geometric alpha)]
  (observe (poisson k 15) alpha))
```

In this program, the recursive function `sample-geometric` defines the functional programming equivalent of a while loop. At each iteration, the function samples from a Bernoulli distribution, returning 1 when the sampled value is 1 and recursively calling itself when the value is 0.

Eager evaluation of if expressions would result in an infinite recursion for this program, so the compilation strategy that we developed in the previous chapter would clearly fail here. This makes sense, since the expression

```clojure
(sample (bernoulli p))
```

can in principle be evaluated an unbounded number of times, implying that the number of random variables in the graph is unbounded as well.

Even though we can no longer compile the program above to a static graph, it turns out that we can still perform inference in order to characterize the posterior on the program output. To do so, we rely on the fact that we can always simply run a program (using lazy evaluation for if expressions) to generate a sample from the prior. In other words, even though we might not be able to characterize the support of a probabilistic program, we can still generate a sample that, by construction, is guaranteed to be part of the support. If we additionally keep track of the probabilities associated with each of the observe expressions that is evaluated in a program, then we can implement sampling algorithms that either evaluate an Metropolis-Hastings.
Implementing “checkpoints”: continuations
How do continuations work?

;; 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)))
How do continuations work?

(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))))))))

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

(fn [yy]
  (+& xx yy
      (fn [xxyy]
          (sqrt& xxyy 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/Angli-

10.5.2 AN EXAMPLE OF A PROBABILISTIC MODEL

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

(flip-example true)
Use in probabilistic program inference

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

(let [u (uniform-continuous 0 1)]
  p (sample u)
  dist (flip p)]
  (observe dist outcome)
  (predict :p p))
Use in probabilistic program inference

(defn flip-query& [outcome k1]
  (uniform-continuous& 0 1)\(\rightarrow\)(let [u (uniform-continuous 0 1)]
    (fn [dist1]
      (sample& dist1)\(\rightarrow\)p (sample u)
      (fn [p] ((fn [p k2]
                 (flip& p)\(\rightarrow\)dist (flip p)]
                 (fn [dist2]
                   (observe& dist2 outcome)\(\rightarrow\)(observe dist outcome)
                   (fn []
                     (predict& :p p k2))))))\(\rightarrow\)(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)))
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 =" (log-prob dist value))
  ;; [ ALGORITHM-SPECIFIC IMPLEMENTATION HERE ]
  ;; Call continuation with no arguments
  (k))
Pure compiled deterministic computation

For end of the program, it yields control to the backend.

We initialize by beginning execution of the program.

Importance Sampling

- Sample $(f, \theta, k)$
- Observe $(g, \phi, y, k)$
- Predict $(z, k)$
- Terminate

While executing $P$, the backend samples a value $w$ which continues execution, providing this value as the output of $P$. The backend samples a value $\mathbf{g}$, a parameter vector $\mathbf{\theta}$, and a observed value $y$. We sample a value $f$, a parameter vector $\phi$, and a observed value $y$. A value $k$ is decoded.

If $P$ encounters a statement, or the end of the program is reached do:

- Predict $(z, k)$
- Terminate

When execution of $P$ encounters a statement, or the end of the program is reached, it returns a value $\mathbf{g}$, $\mathbf{\theta}$, and $y$.
Possible inference algorithms

Some inference engines ("backends") we are ready to implement:

- Importance sampling / likelihood weighting  ➔ Easy
- Single-site Metropolis-Hastings ("random DB")  ➔ Harder
- Sequential Monte Carlo
- Particle MCMC methods (PIMH, CSMC, IPMCMC)  ➔ Conceptually Easy
- Black-box variational inference
Where does machine learning come in?
Have fully-specified model?

- Yes
- No

Inference?

- One-shot
- Repeated

Probabilistic Programming

Amortized Inference

Un- and Semi-Supervised Deep Learning

Trends in probabilistic programming
Amortized inference

Can we learn this directly?
Inference networks as proposal distributions

A probabilistic model generates data

An inverse model generates latents

Can we learn how to sample from the inverse model?

Learning an importance sampling proposal for a single dataset

Target density $\pi(x) = p(x|y)$, approximating family $q(x|\lambda)$

Single dataset $y$: $\arg\min_{\lambda} D_{KL}(\pi||q_{\lambda})$

fit $\lambda$ to learn an importance sampling proposal
Inference networks as proposal distributions

A probabilistic model generates data

An inverse model generates latents

Can we learn how to sample from the inverse model?

Idea: amortize inference by learning a map from data to target

Target density $\pi(x) = p(x|y)$, approximating family $q(x|\lambda)$

Averaging over all possible datasets:

$$\lambda = \varphi(\eta, y)$$

$$\arg\min_{\eta} \mathbb{E}_{p(y)} \left[ D_{KL}(\pi||q_{\varphi(\eta, y)}) \right]$$

learn a mapping from arbitrary datasets to $\lambda$
We can thus fit a choice of distributions

We thus generalize the adaptive importance sampling algorithms by learning a family of

This KL divergence between the true posterior distribution is parameterized by a set of higher-level parameters

more explicitly, if we might observe

which has a gradient

This procedure can be performed entirely offline:

Can train entirely offline:
4.2. A hierarchical Bayesian model

Consider as a new example a representative multilevel model where exact inference is intractable, a Poisson model for estimating failure rates of power plant pumps (George et al., 1993). Given $N$ power plant pumps, each having operated for $t_n$ thousands of hours, we see $x_n$ failures, following $\xi_n \sim \text{Exponential}(1.0)$, $\tau \sim \text{Gamma}(0.1, 1.0)$, $\xi_n \sim \text{Gamma}(\tau, y_n)$, $y_n \sim \text{Poisson}(\xi_n t_n)$. The graphical model, an inverse factorization, and the neural network structure are shown in Figure 2. To generating synthetic training data, $t_n$ are sampled iid from an exponential distribution with mean 50. The repeated structure in the inverse factorization of this model allows us to learn a single inverse factor to represent the distribution $\tilde{p}(\xi_n | t_n, y_n)$ across all $n$. This yields a far simpler learning problem than were we forced to fit all of $\tilde{p}(\xi_1: N | t_1: N, y_1: N)$ jointly. Further, the repeated structure allows us to use a divide-and-conquer SMC algorithm (Lindsten et al., 2014) which works particularly efficiently on this model. Each of the $N$ replicated structures are sampled in parallel with independent particle sets, weighted locally, and resampled; once all $\xi_n$ are sampled, we end by sampling $\tau$ and $y_n$ jointly, which need both be included in order to evaluate the final terms in the joint target density. We stress that there is no obvious baseline proposal density to use for a divide-and-conquer SMC algorithm, as neither the marginal prior nor posterior distributions over $\xi_n$ are available in closed form. Any usage of this algorithm requires manual specification of some proposal $q(\xi_n)$. We test our proposals on the actual power pump failure data analyzed in George et al. (1993). The relative convergence speeds of marginal likelihood estimators from importance sampling from prior and neural network proposals, and SMC with neural network proposals, are shown in Figure 5. To capture the wide tails of the broad gamma distributions, we use a mixture of 10 Gaussians here at each output node, and 500 hidden units in each of two hidden layers.

Non-conjugate polynomial regression

Samples from prior
4.2. A hierarchical Bayesian model

Consider as a new example a representative multilevel model where exact inference is intractable, a Poisson model for estimating failure rates of power plant pumps (George et al., 1993). To generate synthetic training data, we use a mixture of 10 Gaussians here at each output node, and 500 hidden units in each of two hidden layers. Sten et al. (2019) stress that there is no obvious baseline proposal density to use for a divide-and-conquer SMC algorithm, as neither the marginal prior nor posterior distributions over $x_n, y_n, y_0$ are sampled jointly. Further, the repeated structure, which need both be included in order across all speeds of marginal likelihood estimators from importance sampling, is analyzed in George et al. (2019).

We test our proposals on the actual power pump failure data in parallel with independent particle sets, weighted locally, and resampled; once all replicated structures are sampled, we end by sampling from prior $N(0, \theta)$ and 500 hidden units in each of two hidden layers.

To capture the wide tails of the broad gamma distributions, we use a mixture of 10 Gaussians here at each output node, and 500 hidden units in each of two hidden layers. Sten et al. (2019) stress that there is no obvious baseline proposal density to use for a divide-and-conquer SMC algorithm, as neither the marginal prior nor posterior distributions over $x_n, y_n, y_0$ are sampled jointly. Further, the repeated structure, which need both be included in order across all speeds of marginal likelihood estimators from importance sampling, is analyzed in George et al. (2019).

We test our proposals on the actual power pump failure data in parallel with independent particle sets, weighted locally, and resampled; once all replicated structures are sampled, we end by sampling from prior $N(0, \theta)$ and 500 hidden units in each of two hidden layers.

To capture the wide tails of the broad gamma distributions, we use a mixture of 10 Gaussians here at each output node, and 500 hidden units in each of two hidden layers. Sten et al. (2019) stress that there is no obvious baseline proposal density to use for a divide-and-conquer SMC algorithm, as neither the marginal prior nor posterior distributions over $x_n, y_n, y_0$ are sampled jointly. Further, the repeated structure, which need both be included in order across all speeds of marginal likelihood estimators from importance sampling, is analyzed in George et al. (2019).
Non-conjugate polynomial regression

Samples from proposal

Metropolis-Hastings
The graphical model, an inverse factorization, and the neural network allows us to learn a single inverse factor to represent a distribution with mean 50.

Consider as a new example a representative multilevel model where exact inference is intractable, a Poisson model is used. The neural network proposal yields estimated polynomial curves close to the true posterior solution, albeit slightly more diffuse.

We test our proposals on the actual power pump failure data. We use a mixture of 10 Gaussians here at each output node, and 500 hidden units in each of two hidden layers.

Non-conjugate polynomial regression
Non-conjugate polynomial regression

Figure 1: Representative output in the polynomial regression example. Plots show 100 samples each at 5% opacity, with the mean marked as a solid dashed line. These are all proposed using the same neural network — not just the same neural network structure, but also identical learned weights. The MCMC posterior is generated by thinning 10000 samples by a factor 100, after 10000 samples of burnin. The neural network proposal density for the weights yields estimated polynomial curves very close to the true posterior solution, albeit slightly more diffuse. Any small mismatch is easily corrected via importance reweighing.

Structure are shown in Figure 2. Here we place a Laplace prior on the regression weights, and have Student-t likelihoods, giving us

\[ w_d \sim \text{Laplace}(0, 10^{-1}) \] for \( d = 0, 1, 2 \);

\[ t_n \sim \text{t}_\nu(w_0 + w_1 z_n + w_2 z_n^2, \kappa^2) \] for \( n = 1, \ldots, N \) for fixed \( \nu = 4 \), \( \kappa = 1 \), and we place a uniform prior on \( (10^{-1}, 10^1) \) for \( z_n \). The goal is to estimate the posterior distribution of weights for the constant, linear, and quadratic terms, given any possible collected dataset \( \{z_n, t_n\}_{n=1}^N \). In the notation of the surrounding sections, we have latent variables \( x \sim \{w_0, w_1, w_2\} \) and observed variables \( y \sim \{z_n, t_n\}_{n=1}^N \).
Inference networks for probabilistic programs

Input: an inference problem denoted in a probabilistic programming language

Output: a trained inference network (deep neural network “compilation artifact”)

Amortized inference in higher-order languages?

• Manually programmed “guide” program?
  ‣ Intersperse model code and inference
  ‣ Requires support over the same set of “addresses” of random choices on every execution

• Automatic?
  ‣ Use a generic regression model to conditionally generate sequences of random choices
Generating synthetic training data

model places style-specific uniform distributions over different parameters to drive the Captcha renderer. In particular, the style shown in Table I, we use different settings of the prior where

\[ p(x, y) \]

is, effectively, the same as generating synthetic training data. The corresponding per-style renderer and its fidelity being the primary component of the letter identities. Given these, we use a custom stochastic mind that there is a separate unique model for each style. The following equations we omit the style subscript while keeping in mind that there is a separate unique model for each style.

\[
\mathbf{x}^{(n)} = \begin{bmatrix} x_1^{(n)} \\ x_2^{(n)} \\ \cdots \\ x_t^{(n)} \end{bmatrix}
\]

That is, \( \mathbf{x}^{(n)} \) is a \( K \times 1 \) one-hot vector, whose \( i \)-th component is one if \( n \) is instance-specific, the LSTM is run for one-to-one to the components of the latent variable, that parameterize a discrete probability distribution. These output layers allow us to match the dimensions of the discrete distributions for the corresponding latent variables. These outputs are then fed to disentangling networks.

\[
x^{(n)}, y^{(n)} \sim p(x, y)
\]

\( p(x, y) \) is the joint density of the actual values that generated the synthetic image in a way similar to that used by Reed and de Freitas [26], using each previous time step, and a label vector. This is the mechanism for breaking Captchas. A hint appears in the probabilistic system and a paucity of labeled training data, how does one go about breaking Captchas?
What does this look like for the CAPTCHA example?

```python
letters = []
num_letters = sample(Poisson(6))
for i in range(num_letters):
    letters.append(sample(Uniform("a", "z", "A", "Z")))

observe(render(letters), observed_captcha)
return letters
```
What does this look like for the CAPTCHA example?

```python
letters = []
num_letters = sample(Poisson(6))
for i in range(num_letters):
    letters.append(sample(Uniform("a", "z", "A", "Z")))

observe(render(letters), observed_captcha)
return letters
```
What does this look like for the CAPTCHA example?

```
letters = []
um_letters = sample(Poisson(6))
for i in range(num_letters):  // i = 1
    letters.append(sample(Uniform("a", ..., "z", "A", ..., "Z")))
observe(render(letters), observed_captcha)
return letters
```
What does this look like for the CAPTCHA example?

```python
def render(letters):
    return ''.join(letters)

def observe(rendered, observed_captcha):
    return rendered

letters = []
um_letters = sample(Poisson(6))
for i in range(num_letters):  # i = 2
    letters.append(sample(Uniform("a", "z", "A", "Z")))

observe(render(letters), observed_captcha)
return letters
```

---

### LSTM time step 3

- **Observation**: `s`
- **Proposal**: `s`
- **Instance**: 2
- **Address**: `a_2`
- **Type**: Uniform

---

### LSTM time step 4

- **Observation**: `X`
- **Proposal**: `X`
- **Instance**: 3
- **Address**: `a_2`
- **Type**: Uniform
Solving Sudoku with diffusion models

https://plai.cs.ubc.ca/2022/11/16/graphically-structured-diffusion-models/
Writing a good generative model is hard
Probabilistic Programming

<table>
<thead>
<tr>
<th>One-shot</th>
<th>Repeated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Amortized Inference</td>
</tr>
<tr>
<td>No</td>
<td>Un- and Semi-Supervised Deep Learning</td>
</tr>
</tbody>
</table>

Pyro
http://pyro.ai
What kind of a language is Pyro?

• Built on top of Pytorch: based on *differentiable programming*, and takes advantage of the existing Python and Pytorch ecosystem

• **Idea:** define a generative model as a program, and a “inference model” as a second program

• Assign a “name” to every latent random variable, and make sure that they line up (be careful if support is unbounded…!)

• **Variational Bayes:** Optimize the parameters of the “inference model” so that it approximates the posterior (i.e. by minimizing a KL divergence)
Generative model for handwritten digits?

- How do you design a generative model for images?

![Diagram of a generative model for handwritten digits]
Learning deep generative models

Inference
(encoder, guide)

Generative model
(decoder)

$q_\phi(z_n|x_n)$

$p(z_n)$

$p_\theta(x_n|z_n)$

Incomprehensible Latent Variable

Kingma & Welling 2014; Rezende et al. 2014
Disentangled representations

Unexplained variation ("nuisance")

"Interpretable" (digit)
Disentangled representations

Separate interpretable \( y \) from nuisance variables \( z \)

\[ y \text{ (digit label)} \]

\( z \) (handwriting style)

\[ \begin{array}{ccccccccccccccccccc} 0 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 5 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 4 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 2 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 6 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ q & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 8 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 5 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \]

\begin{itemize}
  \item \textbf{Generative model:} predict pixels \( x \) from \( y \) and \( z \)
  \item \textbf{Inference:} predict label \( y \) from pixels \( x \), and then predict \( z \) from \( x \) and \( y \)
\end{itemize}

Kingma et al, \textit{Semi-supervised learning with deep generative models}, NIPS 2014
From one digit to many digits

<table>
<thead>
<tr>
<th>8</th>
<th>7</th>
<th>5</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>0</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

Generative model

Label

X

Y

Z

Pixels

Nuisance

Their solution was to augment the loss function by including the densities of the reconstructions. For example, for discrete latent $y$, we recover the expression in Equation (2).
From one digit to many digits

We extend the models from the MNIST experiment by composing it with a stochastic sequence (with 6 labelled example images for each of the 38 individuals, a supervision rate of x).

First experiment, we seek to measure how well the stochastic sequence generator learns to count manipulating the scale and positioning of the standard digits into a combined canvas, evenly balanced.

In the absence of a canonical multi-MNIST dataset, we created our own from the MNIST dataset by images. This model design is similar to the one used in DRAW [the affine transformations that decompose a multi-MNIST image into its constituent MNIST-like available, we compute the probability of loop iteration sampled for each digit in each iteration to transform the digit images the generative model iteratively samples a digit have a varying number of individual digits, which essentially dictates that the model must learn to particularly, we explore the capacity of our framework to handle models with.

Finally, we conduct an experiment that extends the complexity from the prior models even further.

\[ \text{Input Reconstruction} \]
\[ \text{Varying Lighting} \]
\[ \text{Input Recon. Varying Identity} \]

\[
\begin{align*}
\text{Left:} & \quad \text{Decomposition} \\
\text{Right:} & \quad \text{Classification and} \\
\text{Bottom-Right:} & \quad \text{Semi-Supervised}
\end{align*}
\]

The framework is implemented as a PyTorch library [variational autoencoders (VAEs). This is accomplished by defining hybrid generative models which additionally supported by Intel and DARPA D3M, under Cooperative Agreement FA8750-17-2-0093. FW and NDG were supported under EPSRC grant EP/N510129/1. FW was provided by the EPSRC, ERC grant ERC-2012-AdG 321162-HELIOS, EPSRC DARPA PPAML through the U.S. AFRL under Cooperative Agreement FA8750-14-2-0006.

Acknowledgements

Permit more expressive models, incorporating recursive structures and higher-order functions. The probabilistic programming [given images, we are able to simultaneously reconstruct the inputs as well as its constituent parts.

\[ \text{A complex model has a known simpler model as a substructure, the simpler model and its learned} \]
\[ \text{underlying digits. This also demonstrates how we can leverage compositionality of models: when} \]
\[ \text{allows the generative model to sample MNIST-digit images, while also being able to predict the} \]
\[ \text{model to now also incorporate the same} \]
\[ \text{combinations under sampled affine transformations. In the second experiment, we extend the above} \]
\[ \text{here, the generative model presumes the availability of individual MNIST-digit images, generating} \]
\[ \text{shown in Fig.} \]
\[ \text{such as counting, manipulation of scales and} \]
\[ \text{a random variable. For each loop iteration} \]
\[ \text{is a random variable. For each loop iteration} \]
\[ \text{provides another} \]
\[ \text{networks that make predictions in an interpretable and disentangled space, constrained by the structure} \]
\[ \text{and counting. This can be seen akin to providing bounding-boxes around the} \]
\[ \text{input Reconstruction} \]
\[ \text{Varying Lighting} \]
\[ \text{Varying Identity} \]

\[ \text{Figure 4: Alan Turing Institute under the EPSRC grant EP/N510129/1. FW and NDG were supported under} \]
\[ \text{weights can be dropped in directly.} \]
\[ \text{This work was supported by the EPSRC, ERC grant ERC-2012-AdG 321162-HELIOS, EPSRC} \]
\[ \text{Acknowledgements} \]

\[ \text{Figure 5: Generative and recognition models for the intrinsic-faces and multi-MNIST experiments.} \]
How do we build models?

Inference model
(recurrent neural network)

- $z_k$
- $x_k$
- $y_k$
- $a_k$
- $h_{k-1}$
- $h_k$

Generative model

- $z_k$
- $x_k$
- $y_k$
- $a_k$
- $K$

Nuisance

- Count
- Pixels

Label

Transformation

Multi-MNIST. This is an apposite choice as it satisfies both the requirements above – each image can have a varying number of individual digits, which essentially dictates that the model must learn to manipulate the scale and positioning of the standard digits into a combined canvas, evenly balanced images. This model design is similar to the one used in DRAW [20].

In the absence of a canonical multi-MNIST dataset, we created our own from the MNIST dataset by inverting the affine transformations that decompose a multi-MNIST image into its constituent MNIST-like constituent digits as supervision for the labelled data, which must be taken into account when learning from the pixels in the image. For each loop iteration $k$ from the binary cross-entropy in the same manner as in...
Inference: counting and locating

Inference model
(recurrent neural network)

In the absence of a canonical multi-MNIST dataset, we created our own from the MNIST dataset by images. This model design is similar to the one used in DRAW. Particularly, we explore the capacity of our framework to handle models with an increasing number of constituent digits as supervision for the labelled data, which must be taken into account when learning the likelihood term for the MNIST model. When no supervision is available, we deterministically set the number of latent variables itself determined by a random variable, and models that can be composed of other smaller (sub-)models. We conduct this experiment in the domain of multi-MNIST.

Particularly, we explore the capacity of our framework to handle models with a varying number of individual digits, which essentially dictates that the model must learn to have a varying number of latent variables. This also demonstrates how we can leverage compositionality of models: when no supervision is available, we deterministically set the number of latent variables itself determined by a random variable, and models that can be composed of other smaller (sub-)models. We conduct this experiment in the domain of multi-MNIST.

In the recognition model, we predict the number of digits that can be composed of other smaller (sub-)models. We conduct this experiment in the domain of multi-MNIST.

We extend the models from the MNIST experiment by composing it with a stochastic sequence on its own, with no heed paid to disentangling the latent representations for the underlying digits. First experiment, we seek to measure how well the stochastic sequence generator learns to count across the counts (1-3) and digits. We then conducted two experiments within this domain. In the first experiment, we define a Bernoulli-distributed digit image from the binary cross-entropy in the same manner as in Jampani et al. We additionally support this work through the U.S. AFRL for supervised learning schemes in the domain of visual recognition.

Acknowledgements

This work was supported by the EPSRC, ERC grant ERC-2012-AdG 321162-HELIOS, EPSRC D3M, under Cooperative Agreement FA8750-17-2-0093, DARPA PPAML through the U.S. AFRL under Cooperative Agreement FA8750-14-2-0006. FW was additionally supported by Intel and DARPA D3M, under Cooperative Agreement FA8750-17-2-0093.
Real-world examples: molecule generation
Recap!

- Probabilistic programming languages can make writing probabilistic models, and doing inference, faster and more efficient.

- Big challenge: Bayesian inference is, in general, pretty hard. But:
  - ... restricting the probabilistic programming language can help keep inference more tractable.
  - ... even in unrestricted models, it’s possible to define algorithms which will still work (though computational / statistical efficiency is not guaranteed…).

- Deep learning can be useful for amortized inference and for model learning.


- Frank Wood’s graduate course: [https://www.cs.ubc.ca/~fwood/CS532W-539W/](https://www.cs.ubc.ca/~fwood/CS532W-539W/)
Thanks!