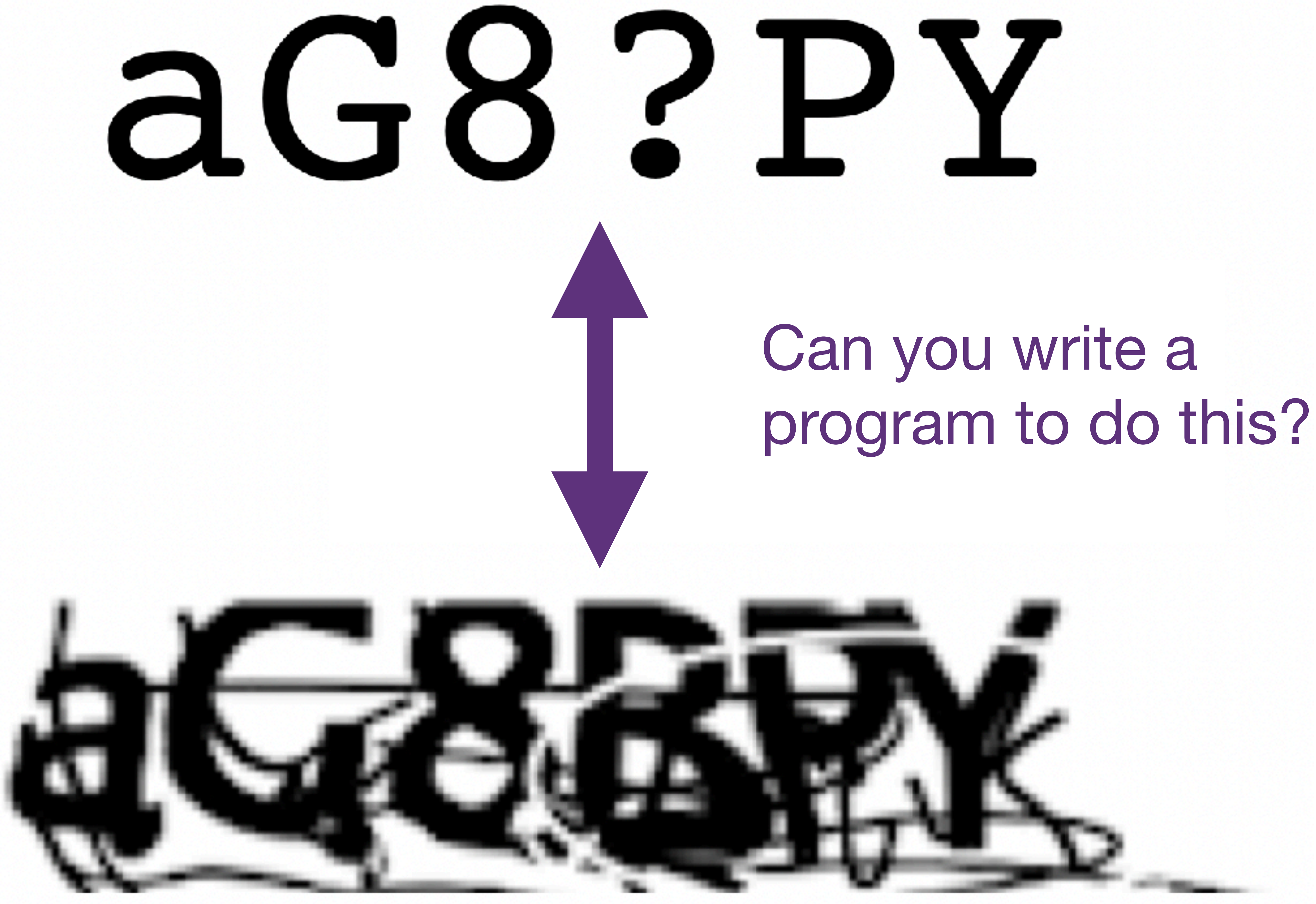
Programs as probabilistic models

Brooks Paige
UCL AI Centre
1 Nov 2023

Generative models?



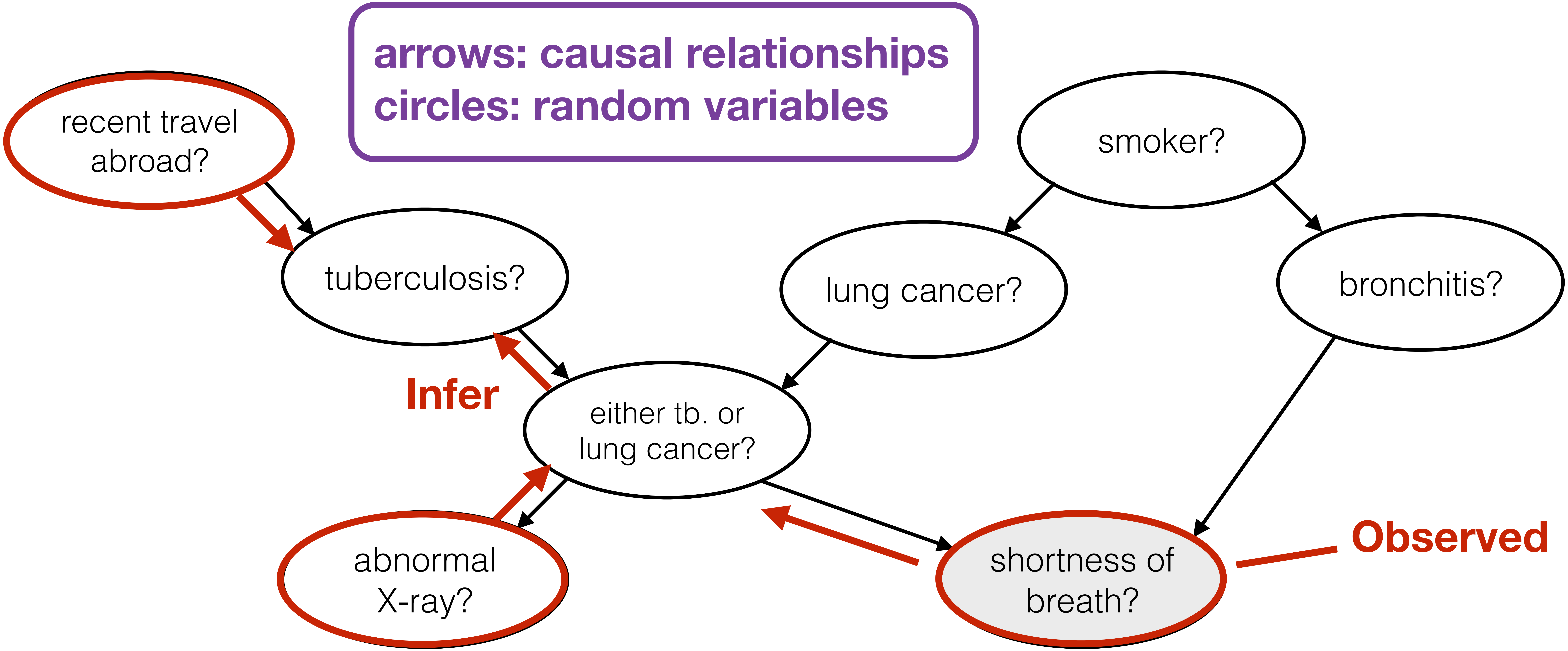
Forward models are “easy”

2	1	4	7	3	**9**	6	5	8
6	8	5	1	4	**2**	9	**3**	7
9	7	3	8	**5**	6	2	4	**1**
4	9	8	6	1	3	**5**	7	2
5	2	**7**	9	8	**4**	3	1	**6**
3	6	1	2	7	5	4	8	**9**
8	5	6	4	**2**	**7**	1	9	3
7	4	2	3	**9**	1	8	6	5
1	3	9	**5**	**6**	8	7	2	4

Can you write a program to solve Sudoku problems?

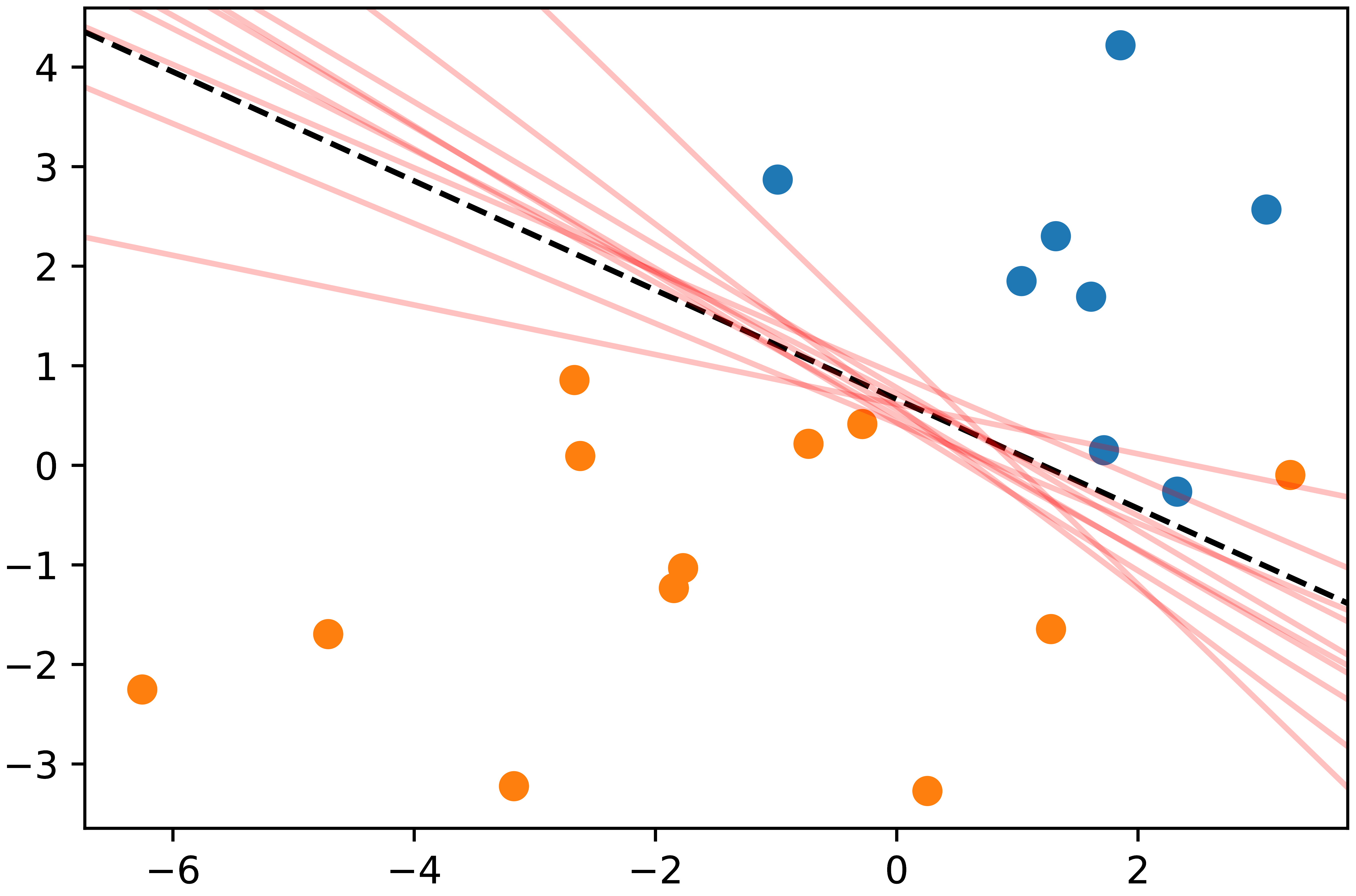
Can you write a program to generate Sudoku problems?

Model relationships between many variables



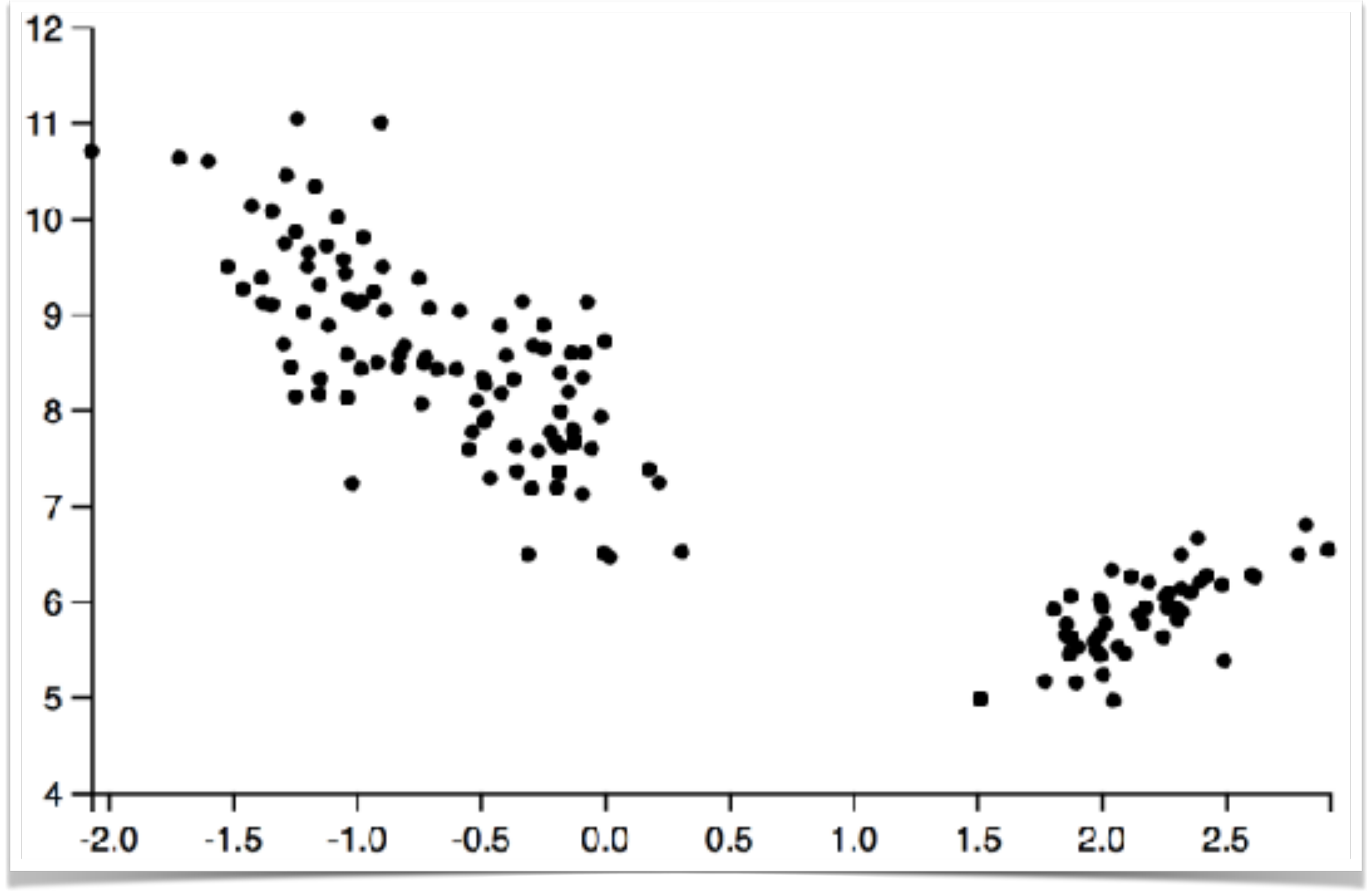
Lauritzen & Spiegelhalter, 1988

Quantification of uncertainty

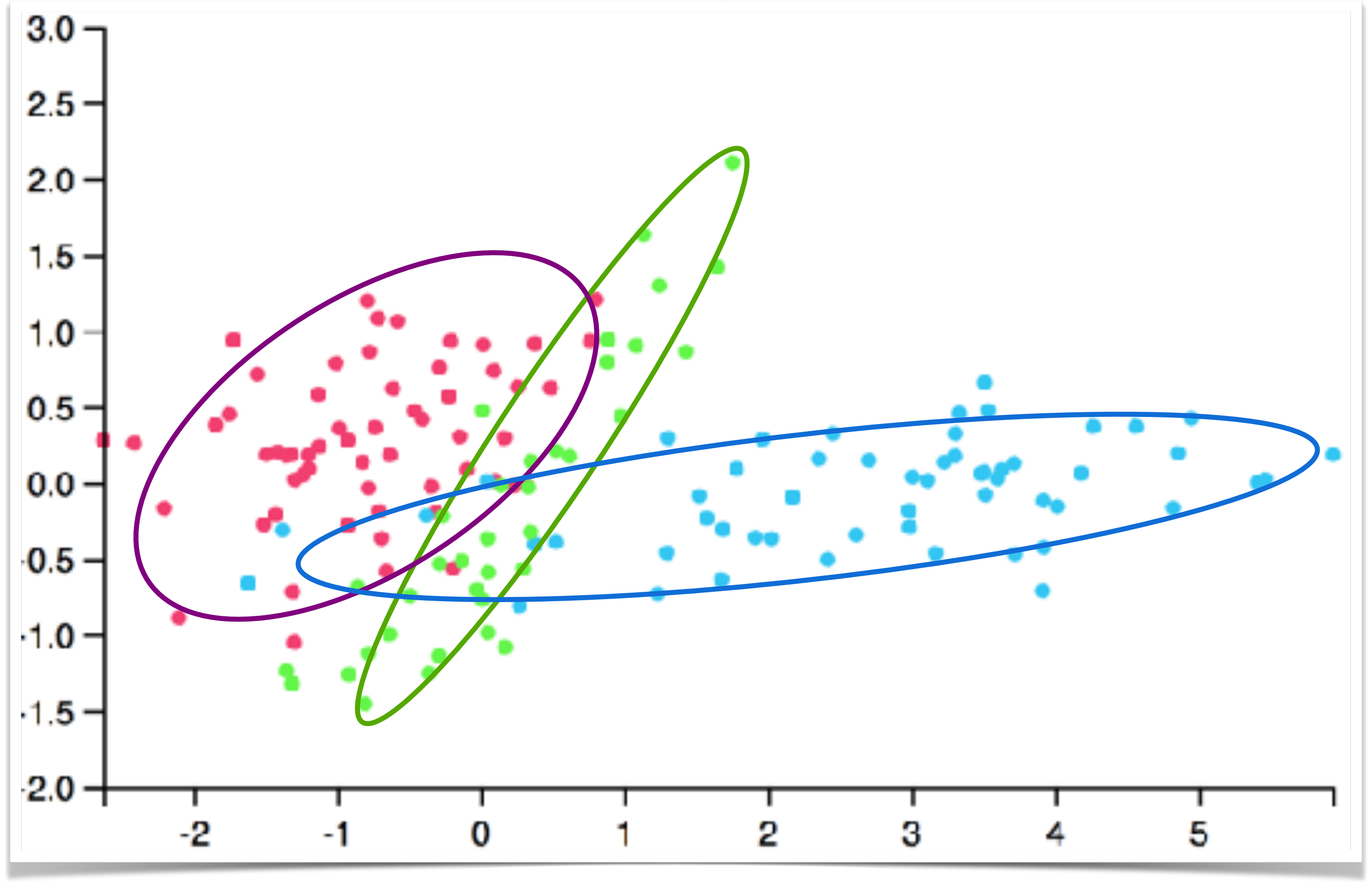


Motivation: models in machine learning

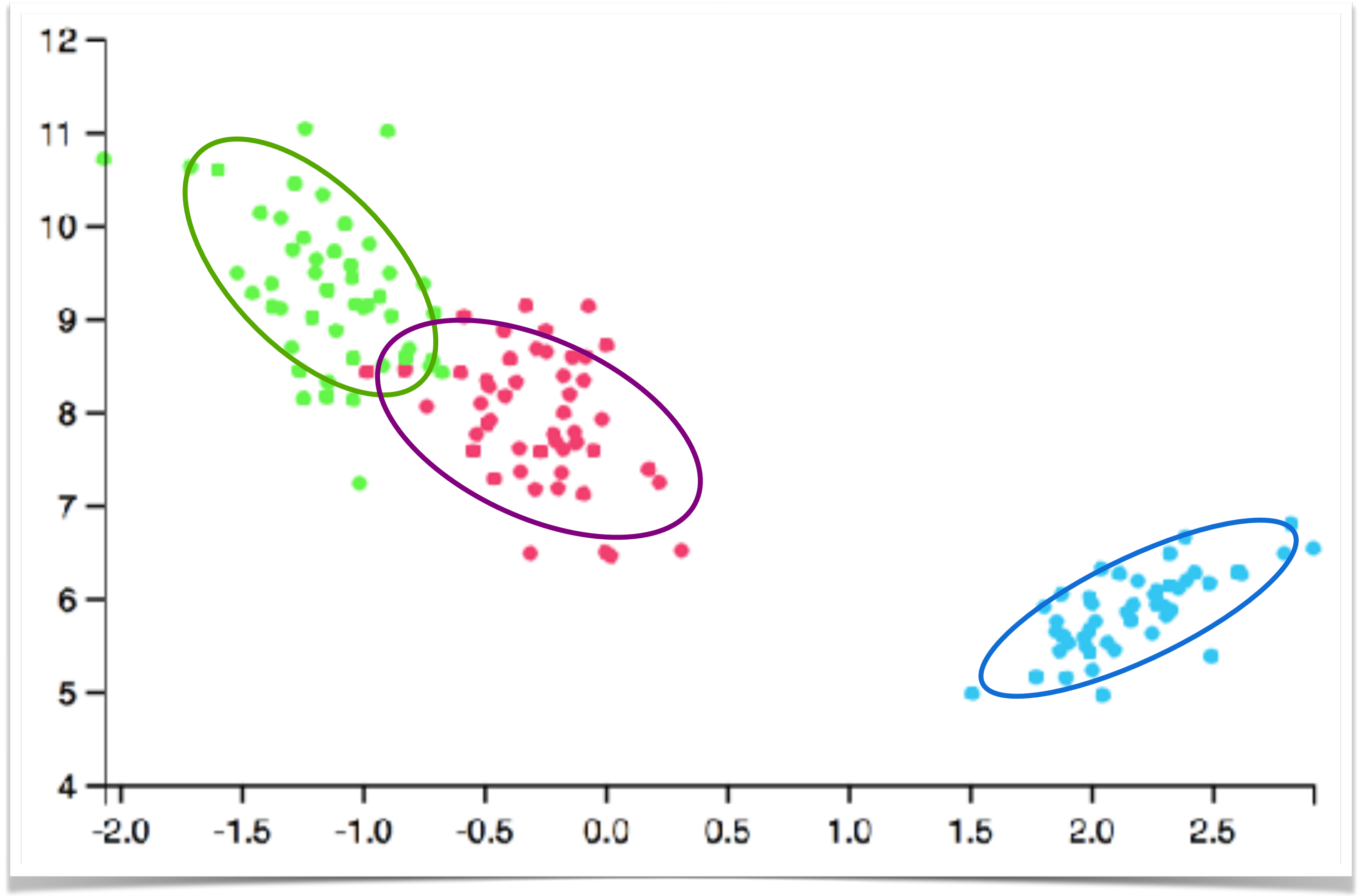
Data
(input)



Generative Model
(assumptions)



Inference
(output)



Prediction Task

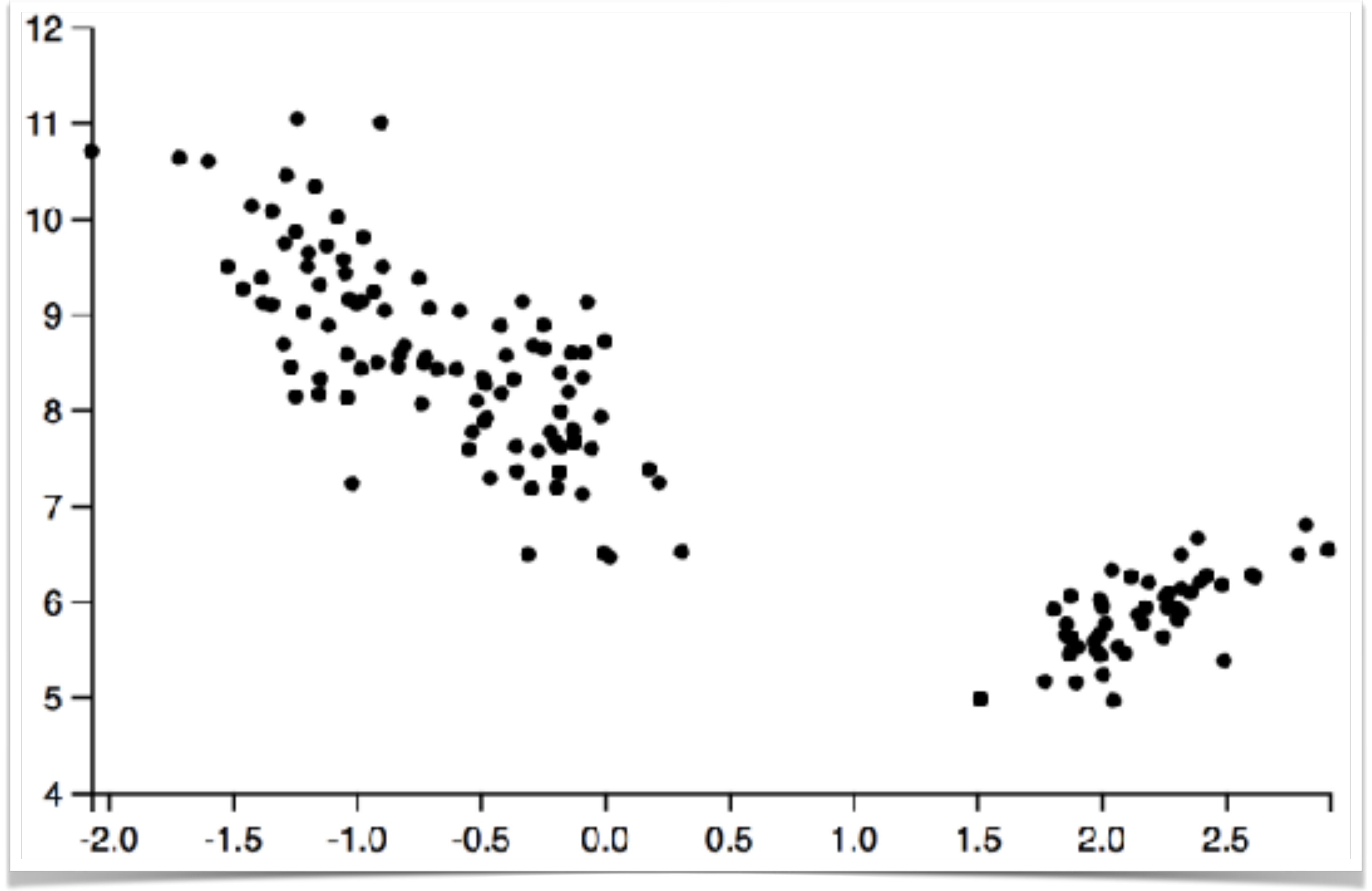
$$\mu_k, \Sigma_k \sim \mathrm{NormalWishart}(\psi)$$
$$z_n \sim \mathrm{Discrete}(\pi)$$
$$y_n \sim \mathrm{Normal}(\mu_{z_n}, \Sigma_{z_n})$$

Gibbs Sampler

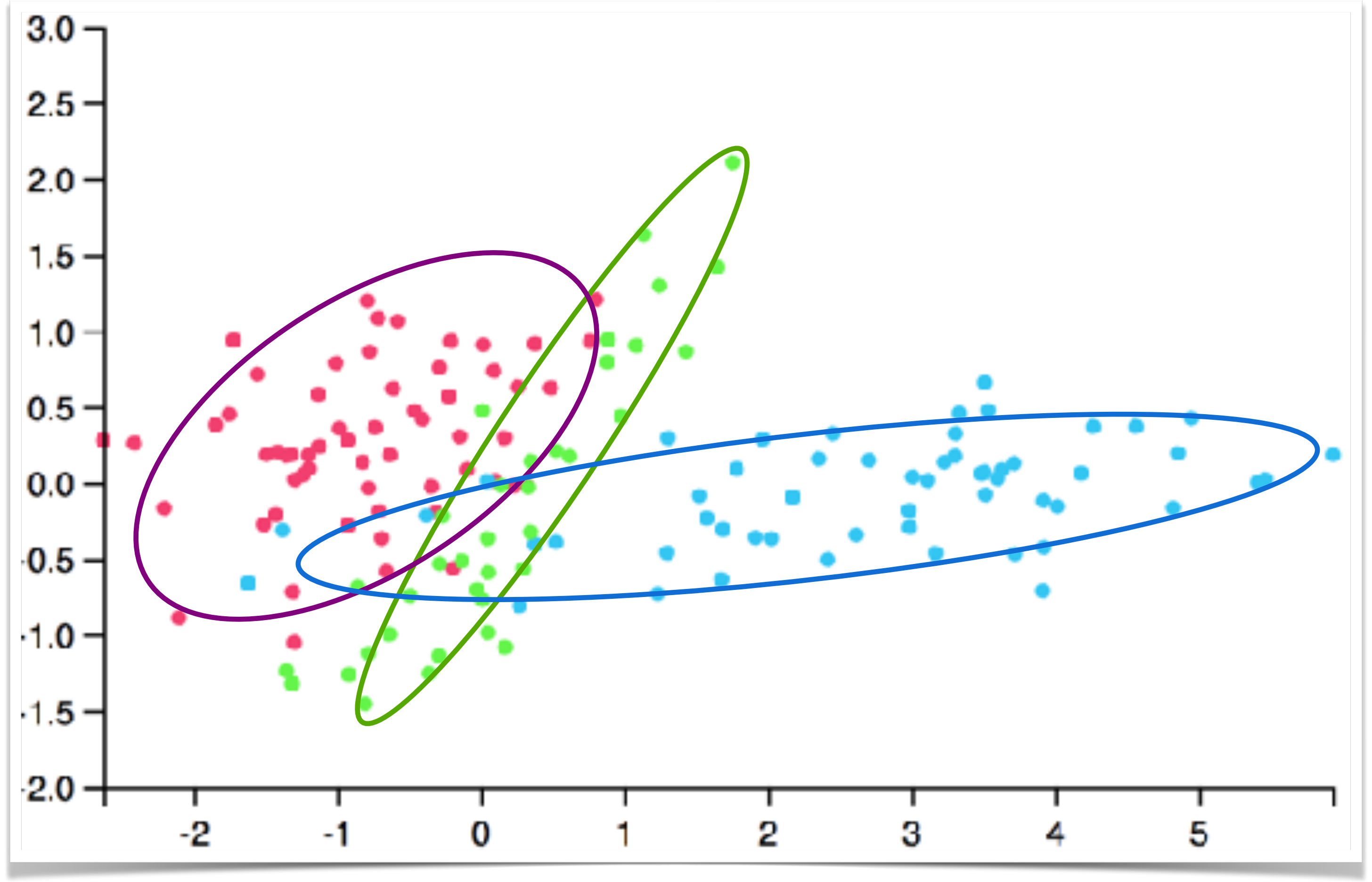
Expectation Maximization

Motivation: models in machine learning

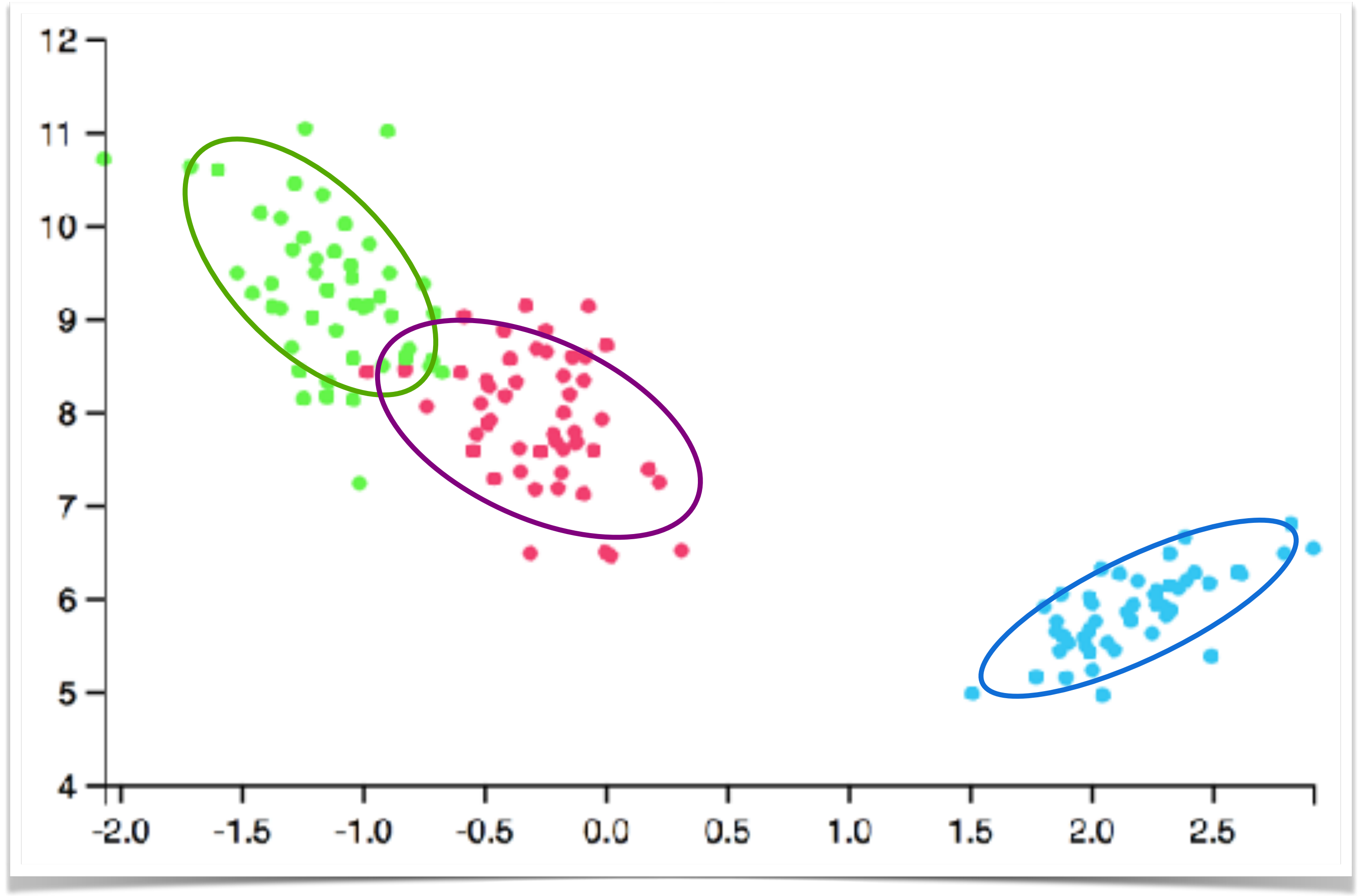
Data
(input)



Generative Model
(assumptions)



Inference
(output)



Machine Learning Software

Prediction Task →

$$\mu_k, \Sigma_k \sim \mathrm{NormalWishart}(\psi)$$
$$z_n \sim \mathrm{Discrete}(\pi)$$
$$y_n \sim \mathrm{Normal}(\mu_{z_n}, \Sigma_{z_n})$$

(Math)

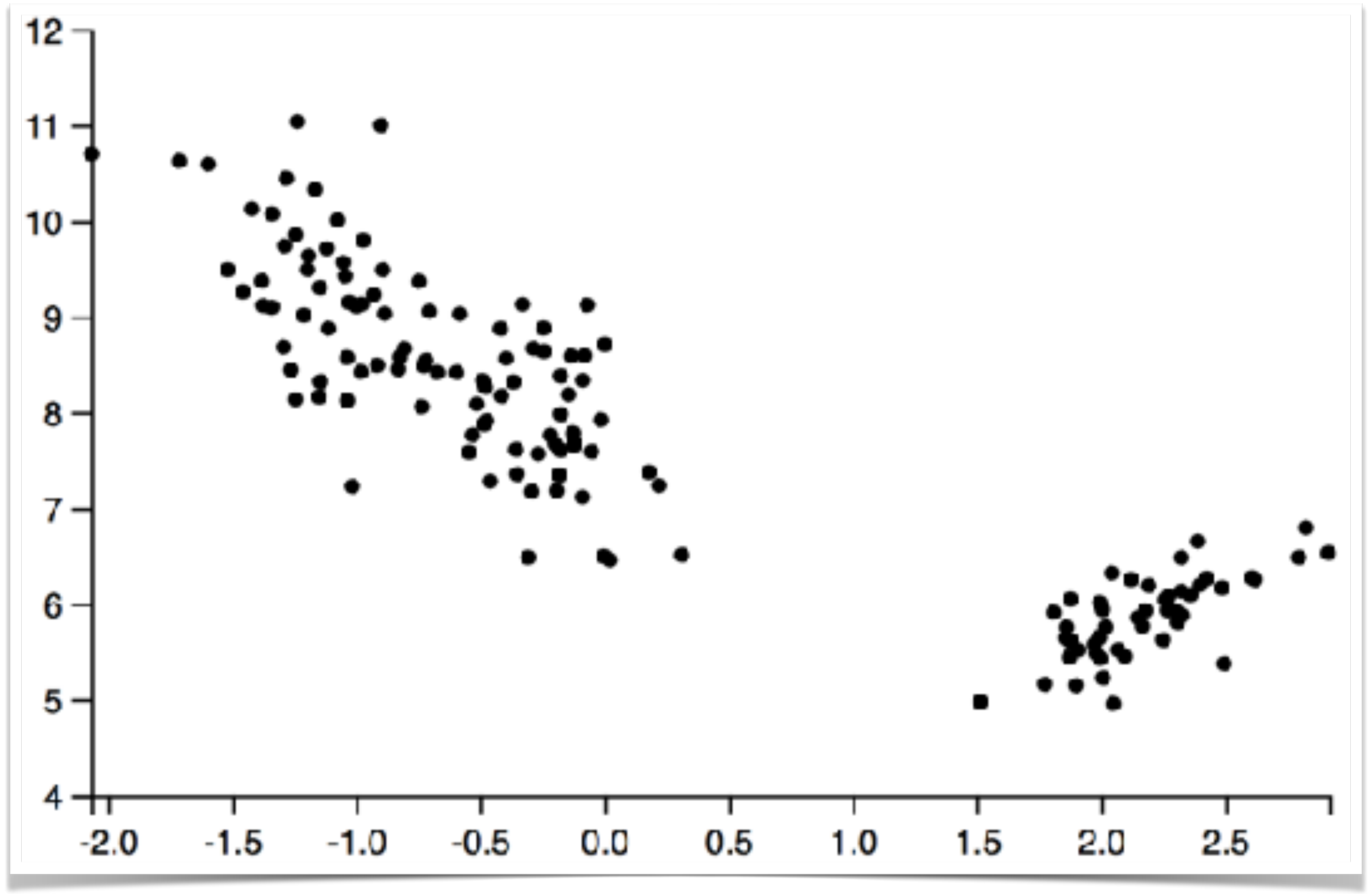
→ Gibbs Sampler

→ Expectation Maximization

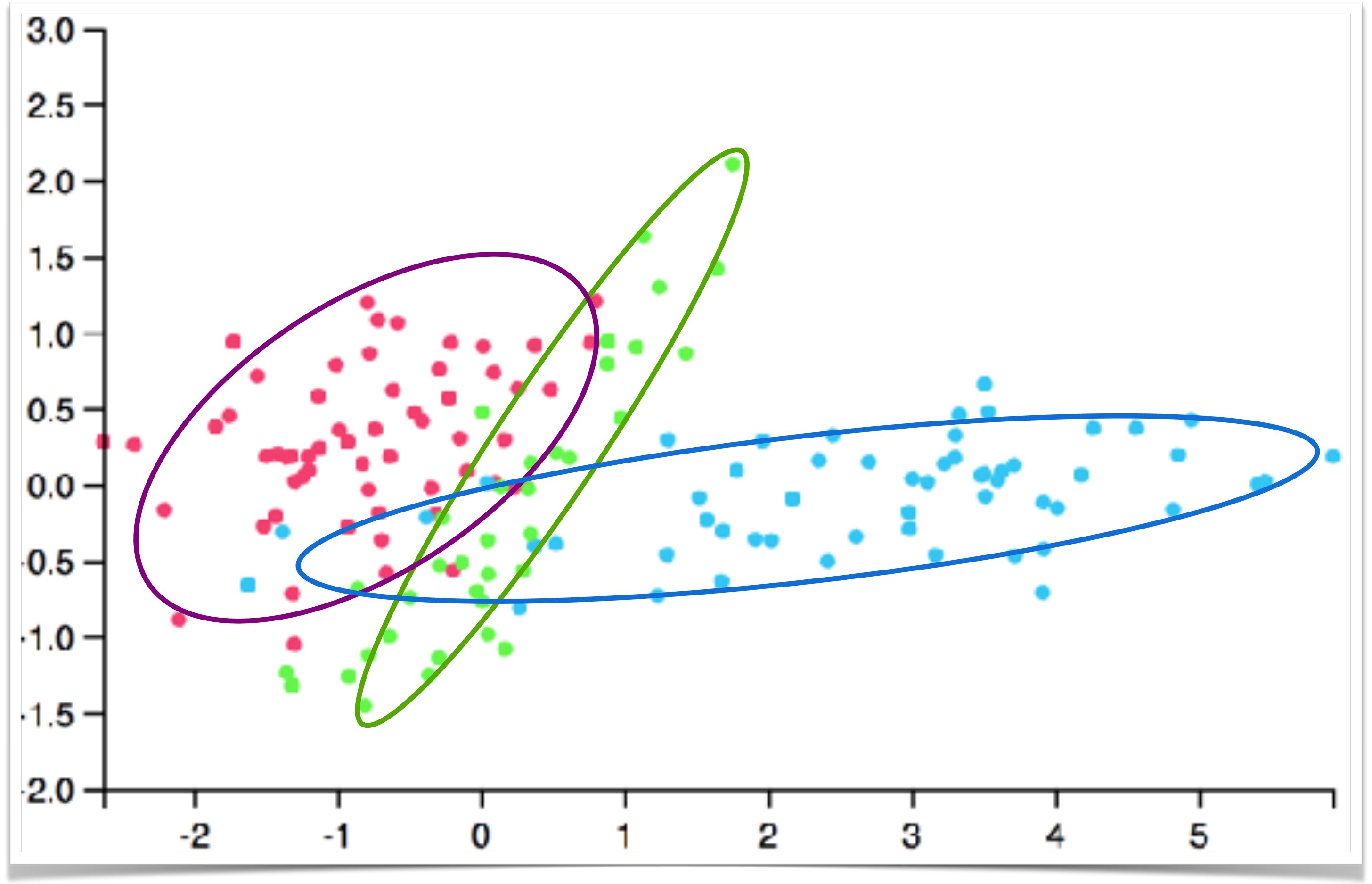
(Model-Specific)

Motivation: models in machine learning

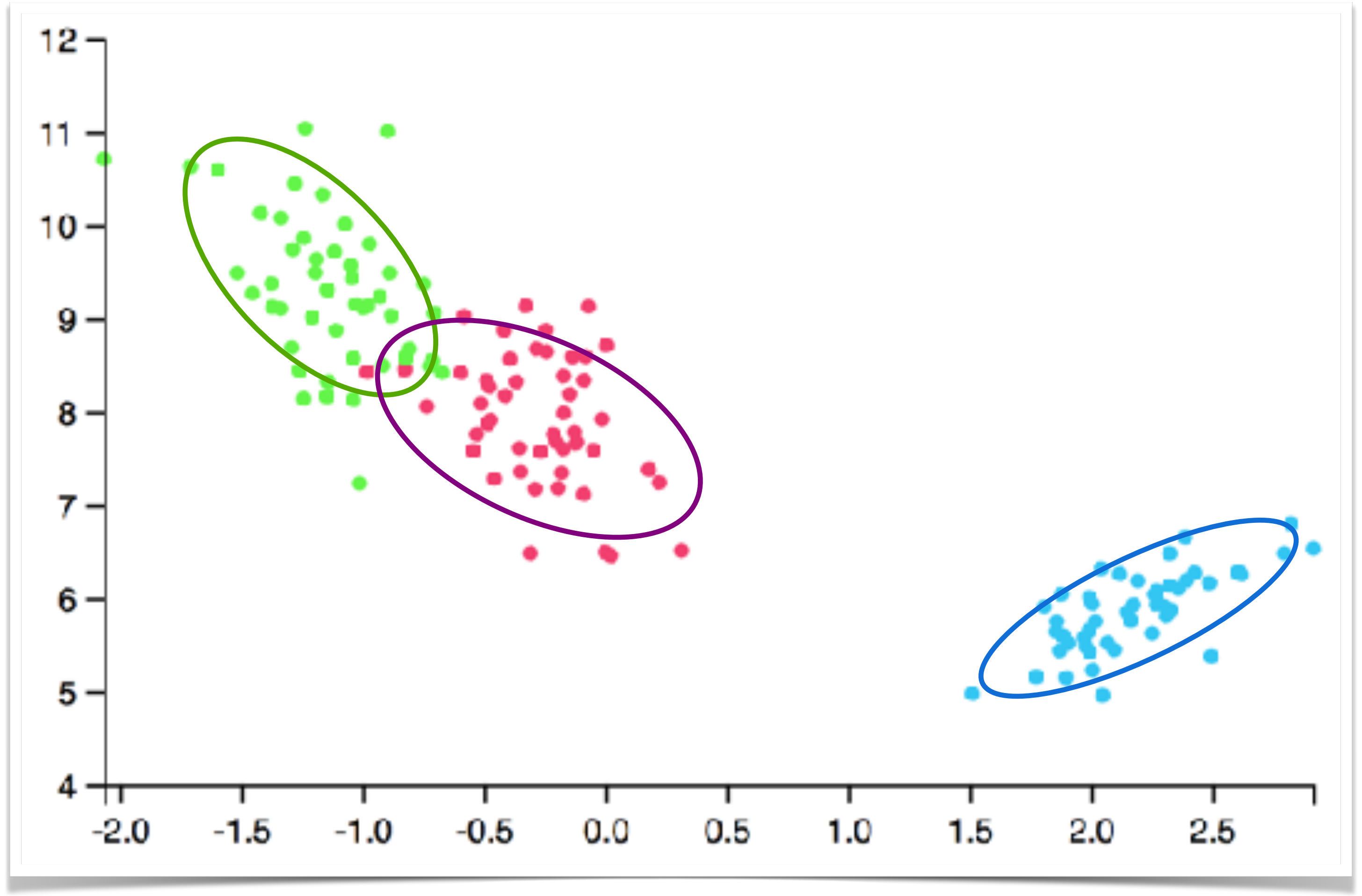
Data
(input)

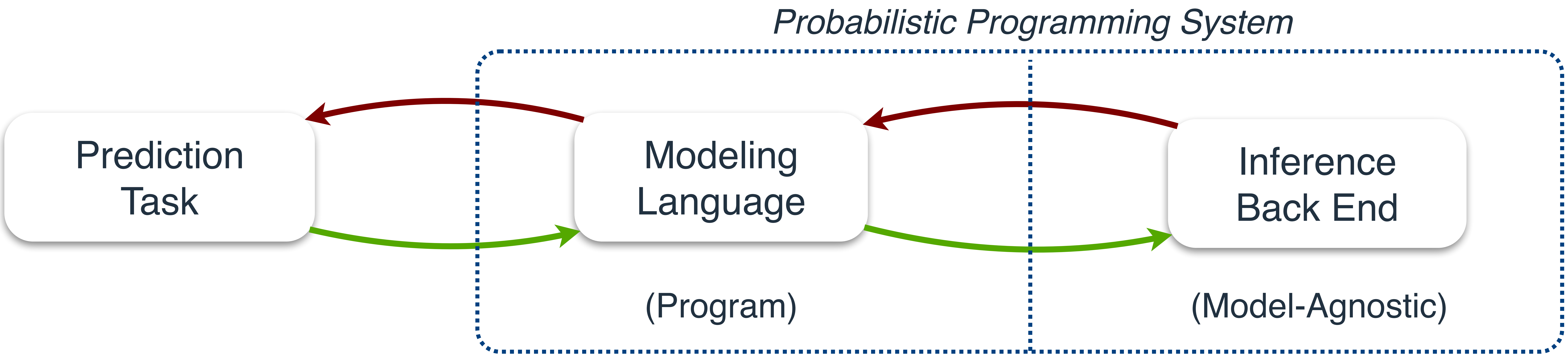


Generative Model
(assumptions)

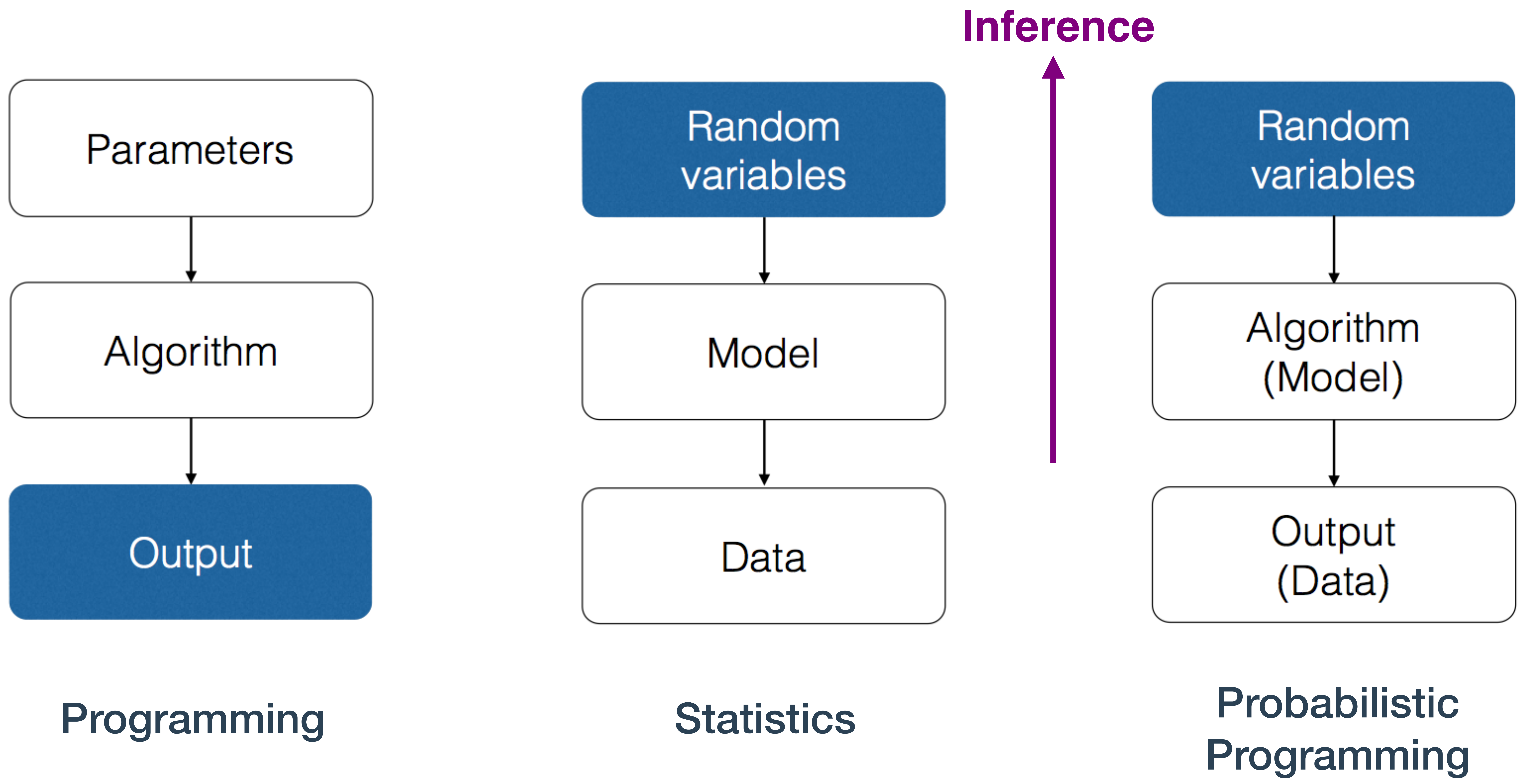


Inference
(output)





Intuitive view of probabilistic programming



A probabilistic *program*

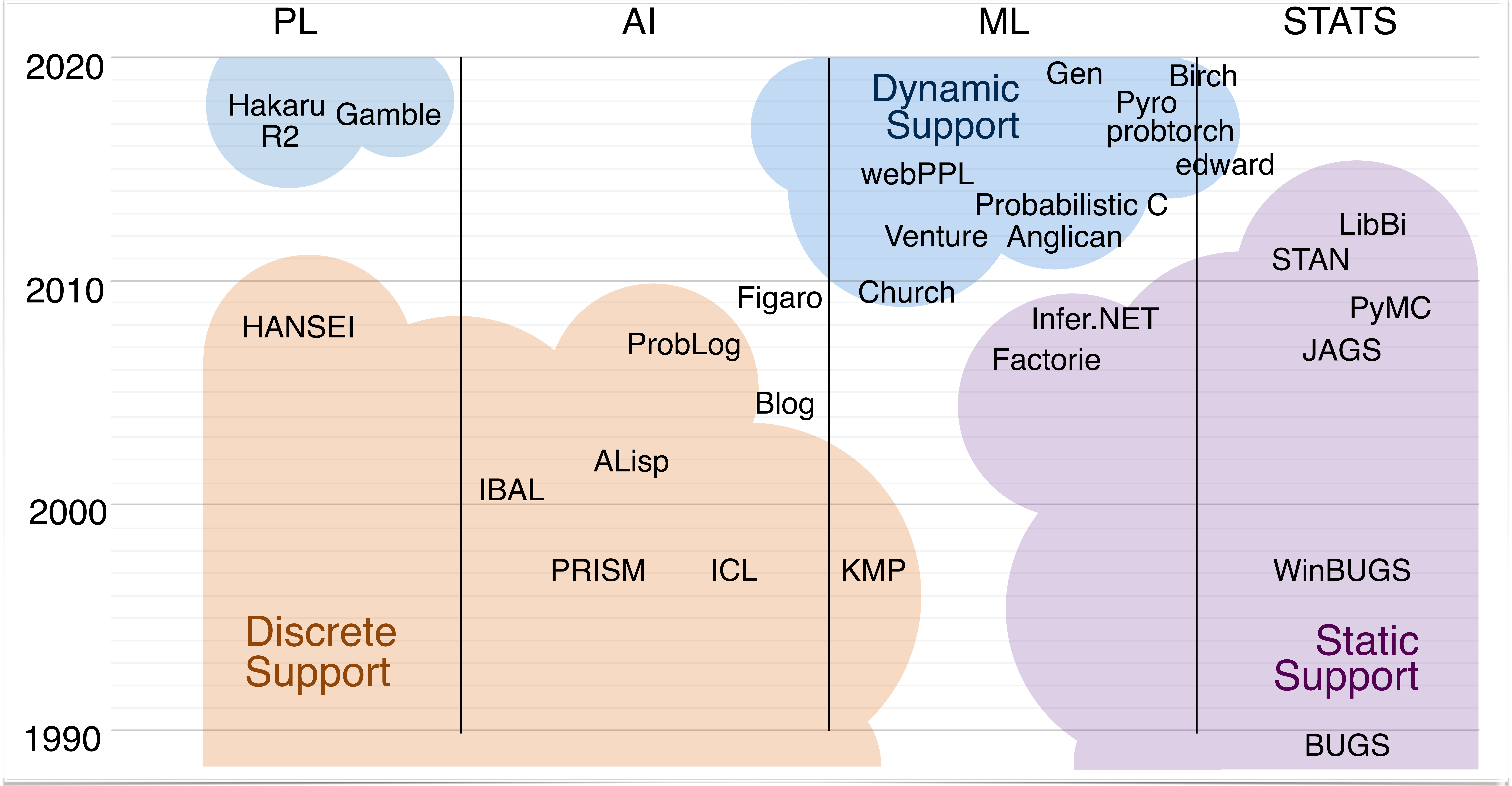
“Probabilistic programs are usual functional or imperative programs with two added constructs:

(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
“Probabilistic programming.” In Proceedings of On The Future of Software Engineering (2014).

Languages and systems



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!

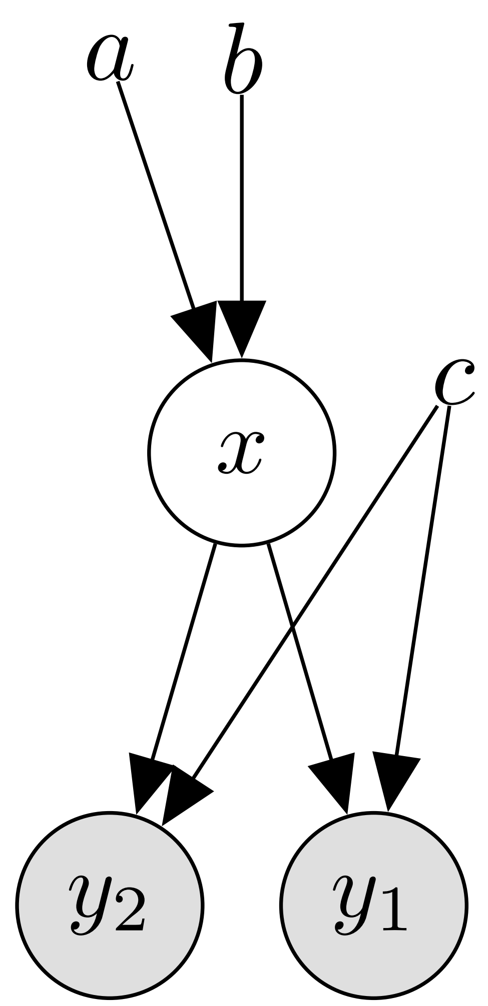
BUGS STAN Infer.NET Pyro

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

$$x \sim \mathcal{N}(a, b^{-1})$$

$$y_i \sim \mathcal{N}(x, c^{-1}), \;\; i = 1, \ldots, N$$




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

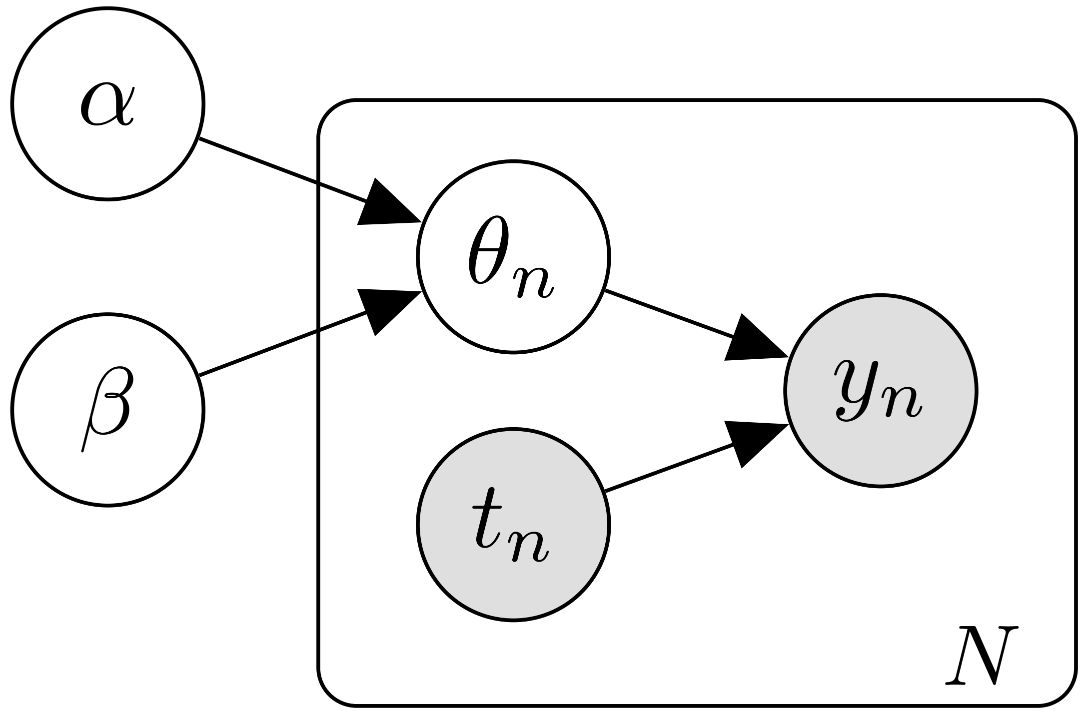
Language restrictions?

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!”

	Inference	Models	Language
BUGS	Gibbs Sampling	Finite graphical models	Bounded loops; no branching
STAN	Hamiltonian Monte Carlo	Continuous latent variables	Bounded loops; no discrete r.v.s
Infer.NET	Expectation Propagation	Factor graphs	Finite composition of factors

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

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

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

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

- Branching

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

Functions

- Functions are first class

```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

```clojure
;; 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

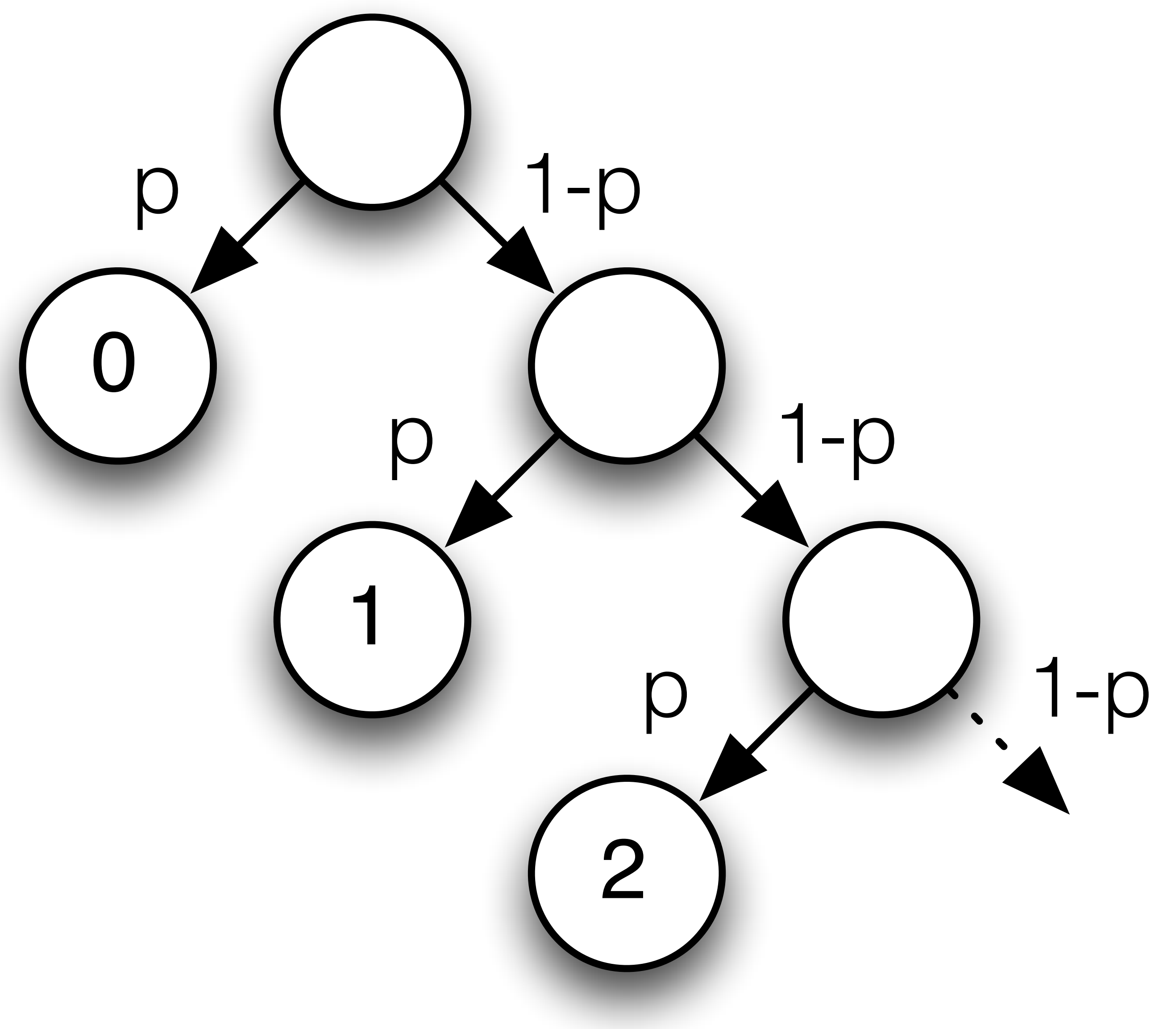
```clojure
;; Reduce recursively applies function,
;; to result and next element, i.e.
(reduce + 0 [1 2 3 4])
;; does (+ (+ (+ 0 1) 2) ...
;; and evaluates to 10
```

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

```
(defm sample-geometric [p]
  (if (sample (flip p))
    0
    (+ 1 (sample-geometric p))))
```



Other way around: language first

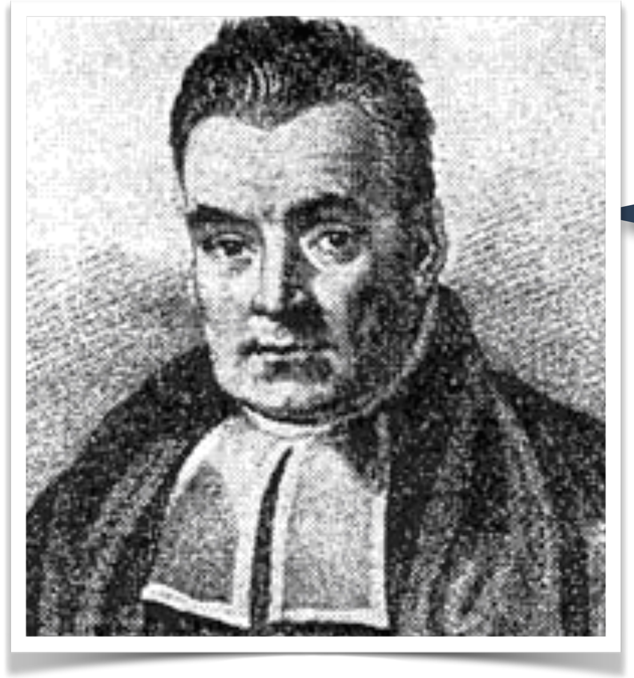
Unrestricted Languages:

- “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



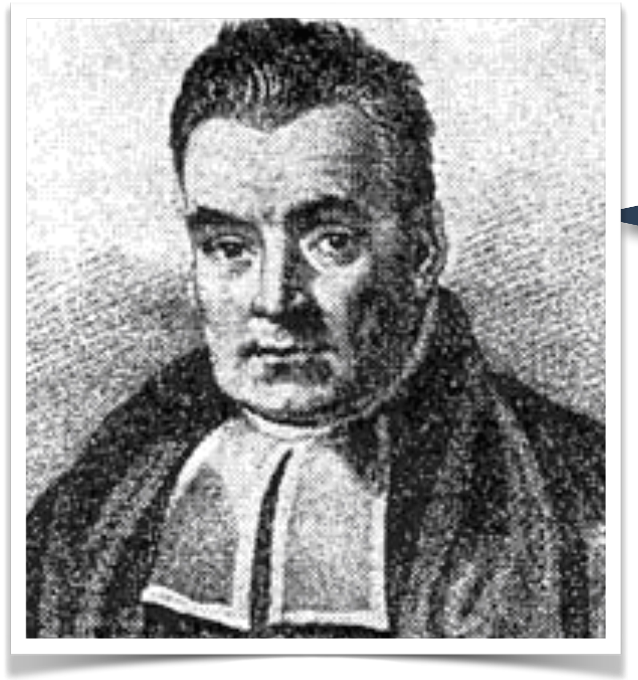
$$p(x \mid y) = p(y \mid x)p(x)/p(y)$$

Posterior Likelihood Prior

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

Estimate ***predict*** *values, under posterior on* ***sample*** *values, given* ***observe*** *values.*

Bayesian inference



$$p(x \mid y) = p(y \mid x)p(x)/p(y)$$

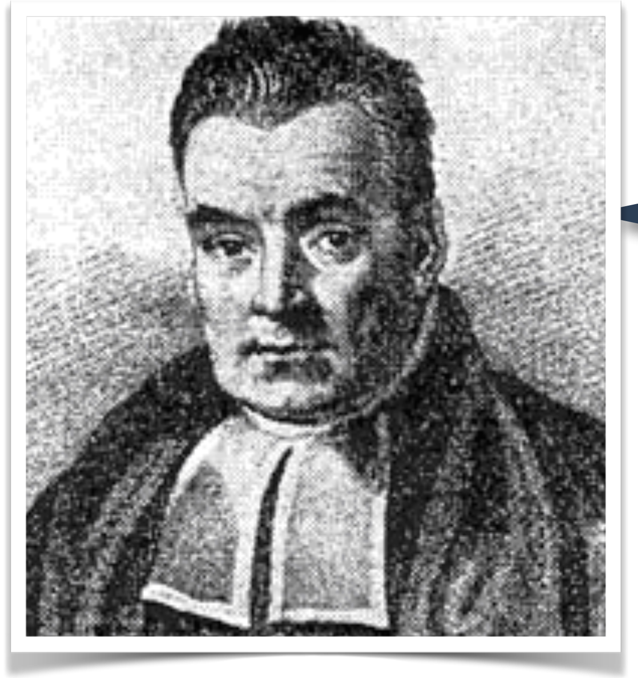
Posterior Likelihood Prior

Example: Biased Coin

y Observed data (flip outcomes)

x Unknown variable (coin bias)

Bayesian inference



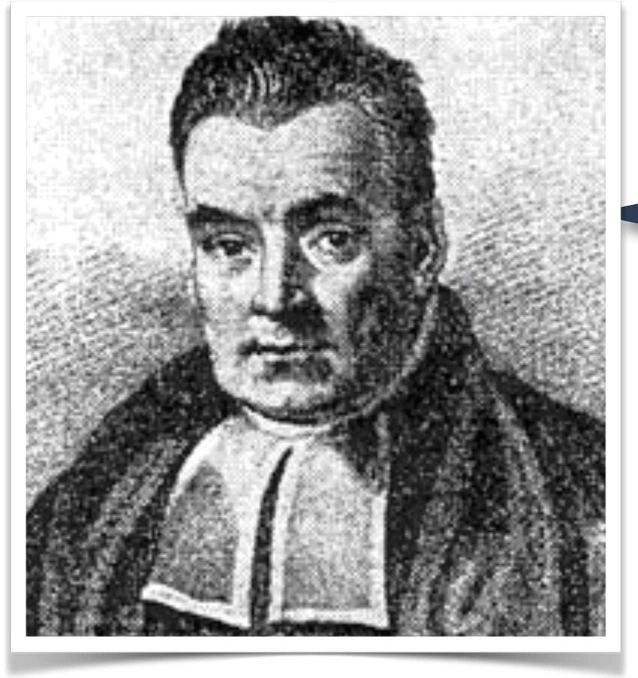
$$p(x \mid y) = p(y \mid x)p(x)/p(y)$$

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

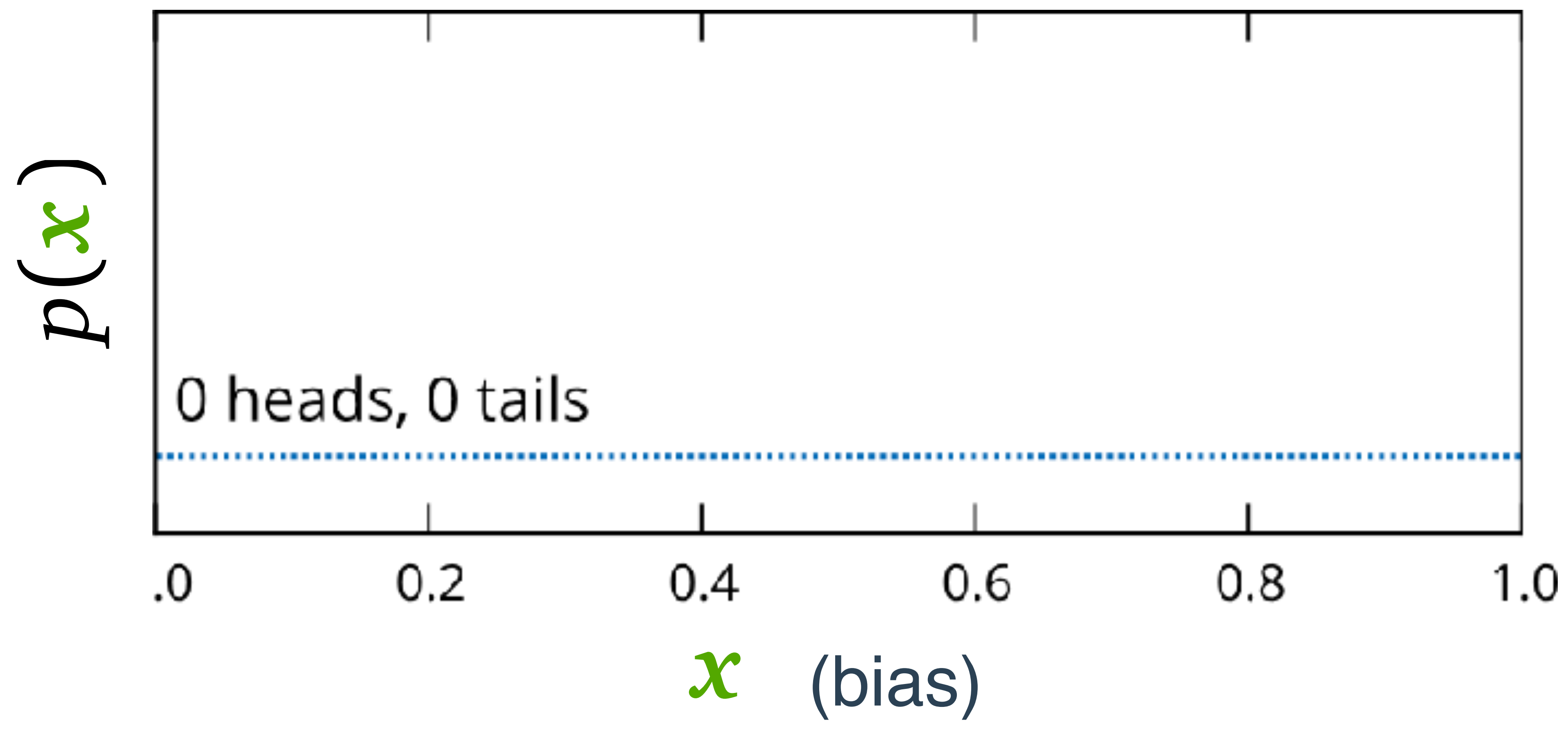
Bayesian inference



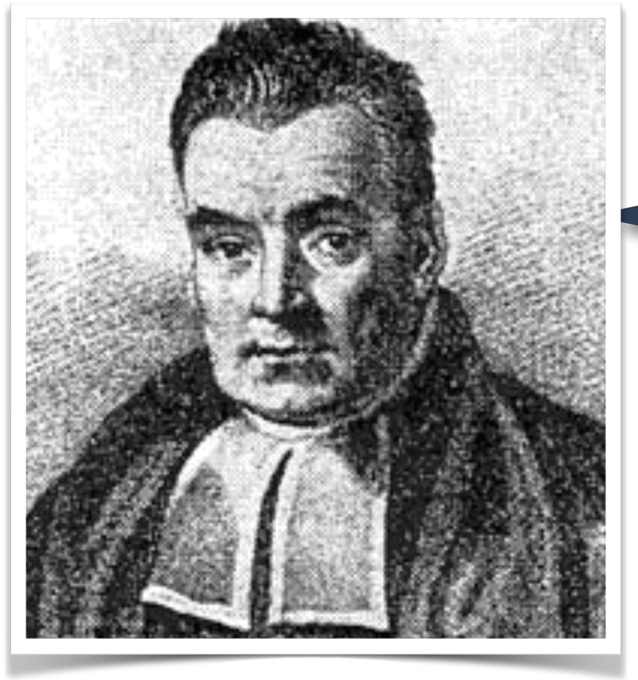
$$p(x \mid y) = p(y \mid x)\,p(x)/p(y)$$

Posterior Likelihood Prior

Example: Biased Coin



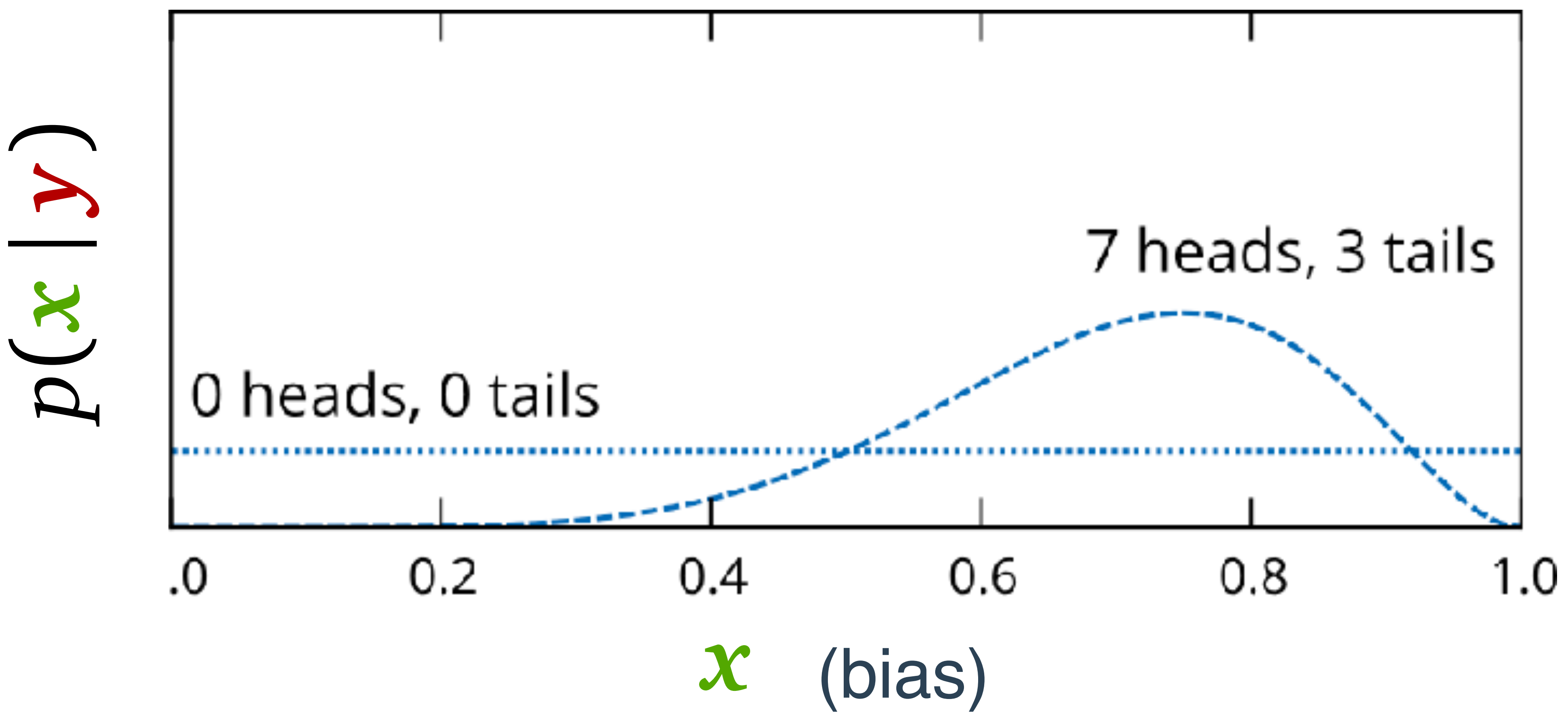
Bayesian inference



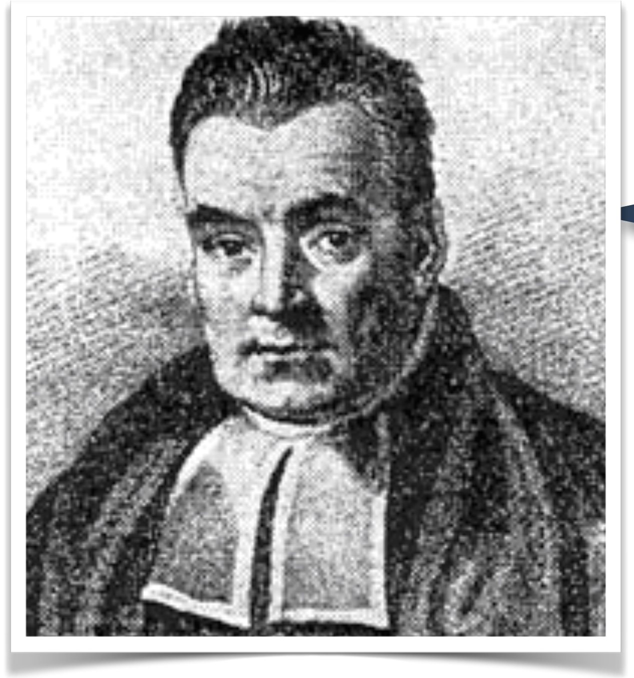
$$p(x \mid y) = p(y \mid x)\,p(x)/p(y)$$

Posterior — Likelihood — Prior

Example: Biased Coin



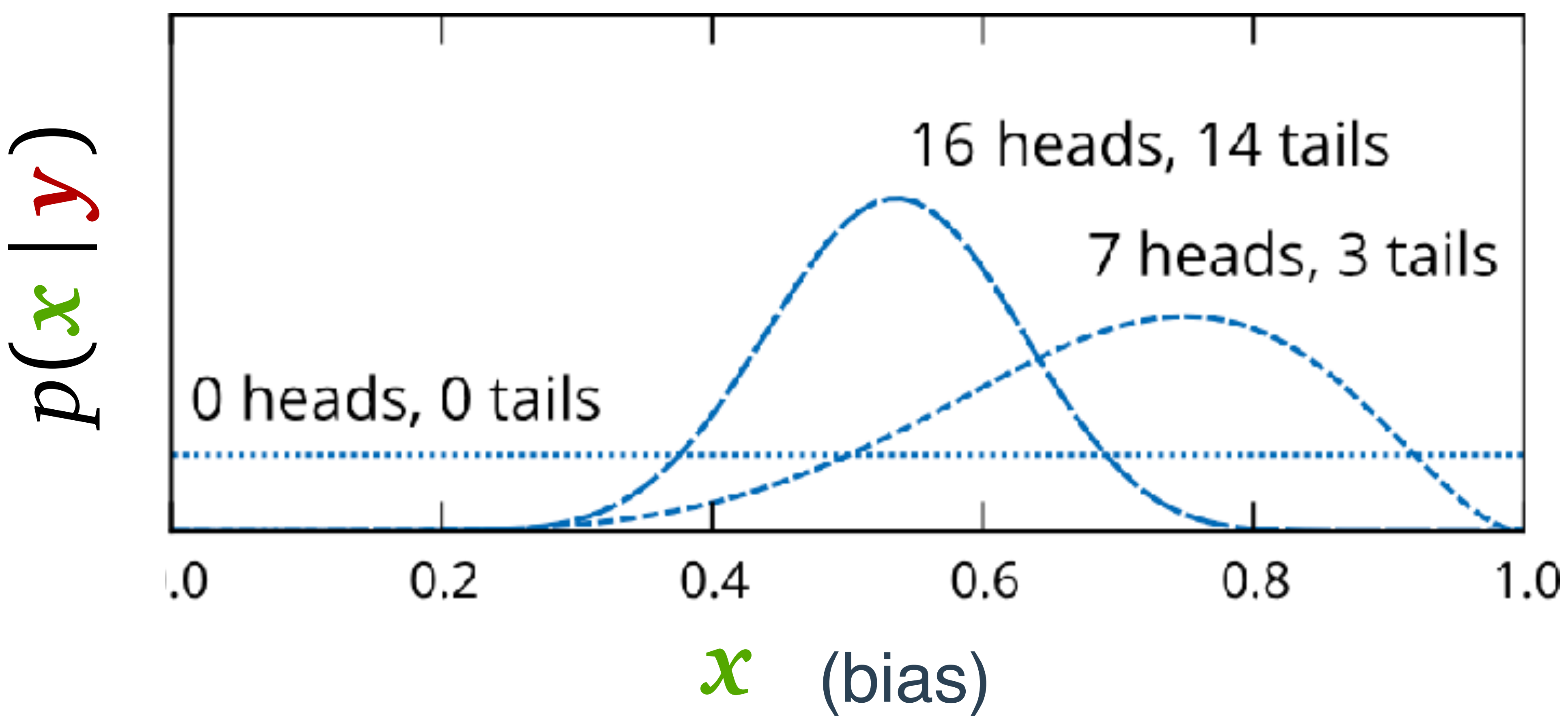
Bayesian inference



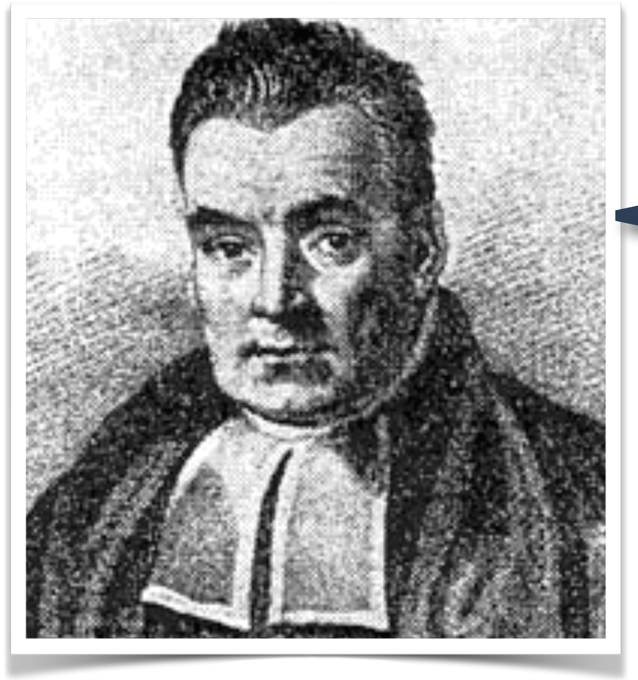
$$p(x \mid y) = p(y \mid x)p(x)/p(y)$$

Posterior Likelihood Prior

Example: Biased Coin



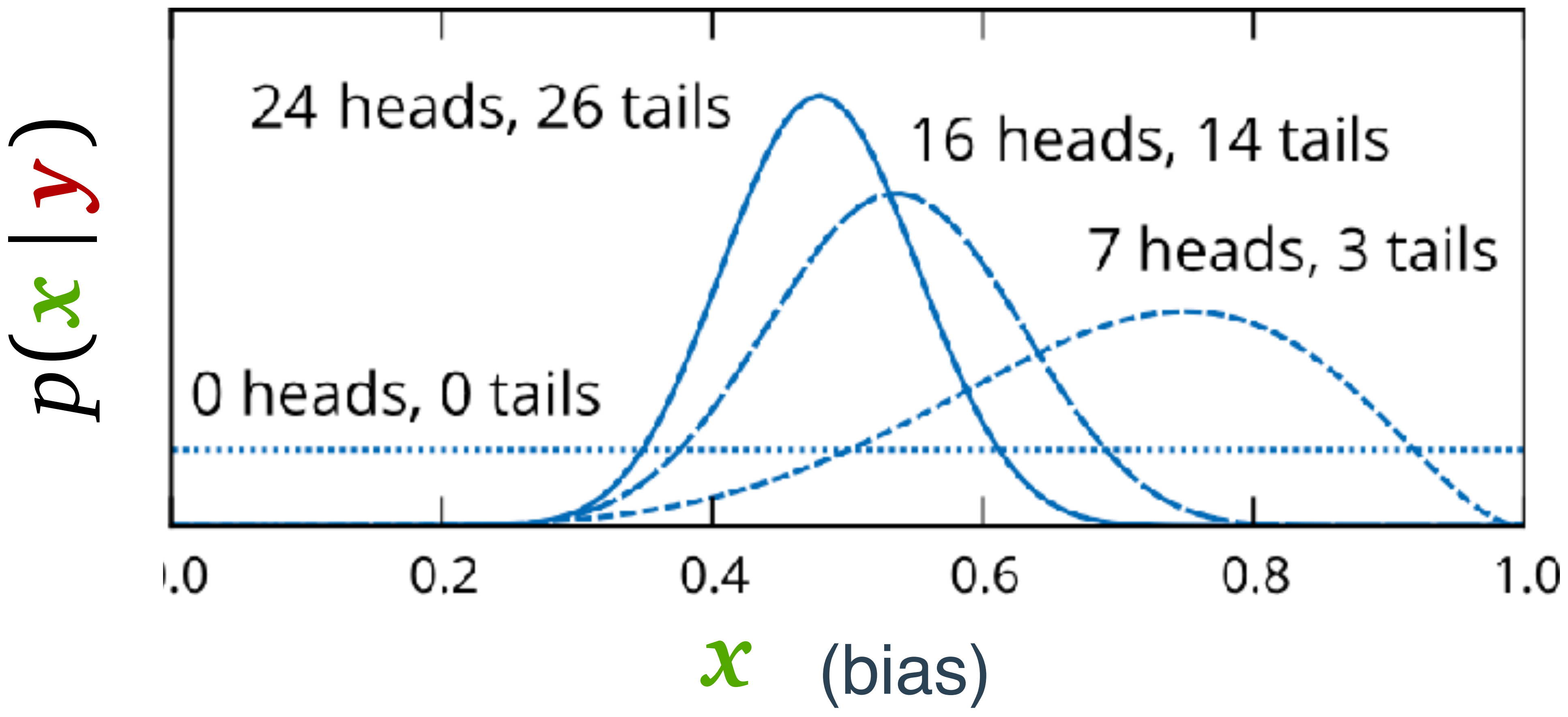
Bayesian inference



$$p(x \mid y) = p(y \mid x)\,p(x)/p(y)$$

Posterior Likelihood Prior

Example: Biased Coin



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

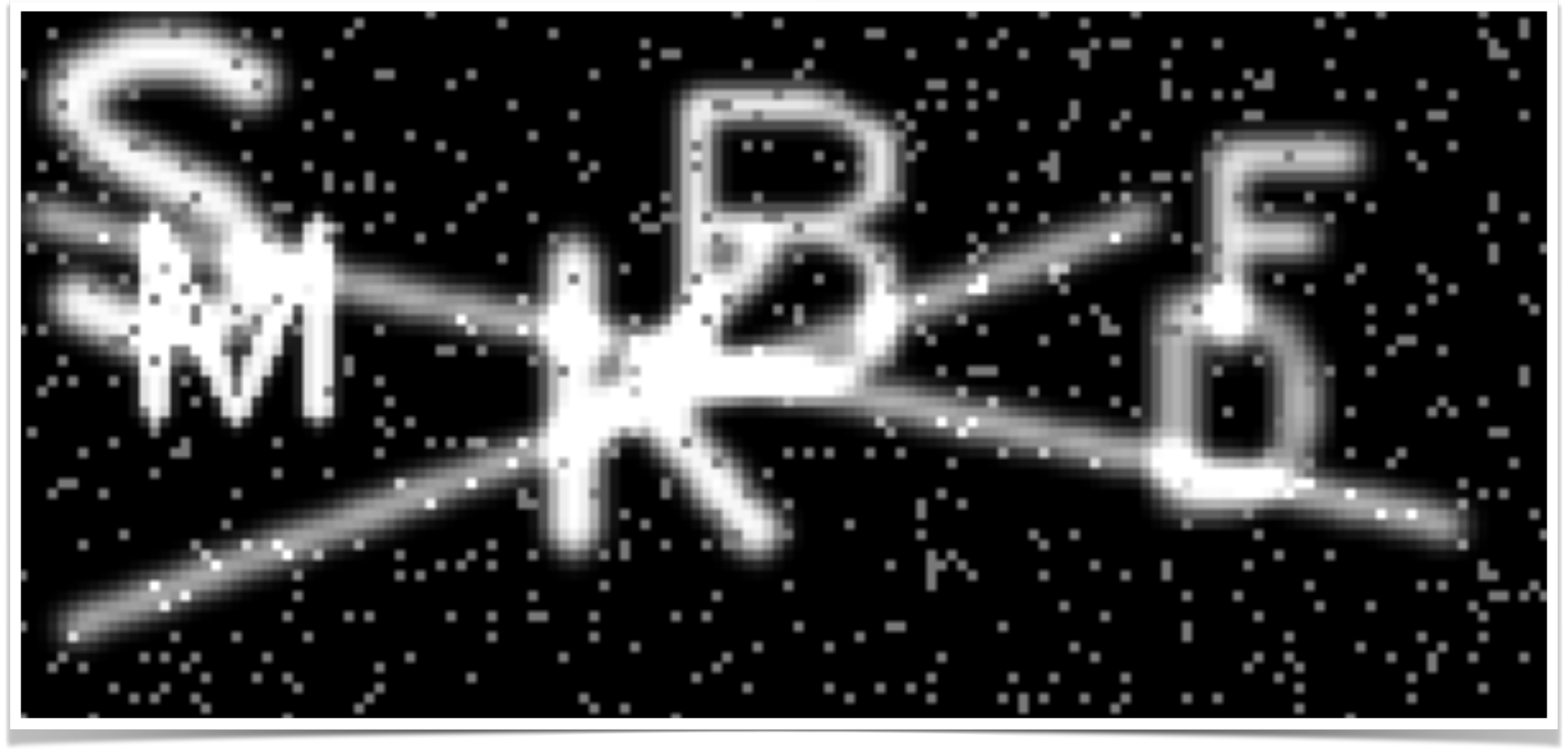
Estimate distribution over output values under posterior of **sample** *values, given* **observe** *values.*

$$p(x \mid y) = p(y \mid x)p(x)/p(y)$$

- Implements (inference-algorithm-specifc) **sample** and **observe** handlers
- Returns weighted samples

Generative model for Captcha-breaking

Target Image

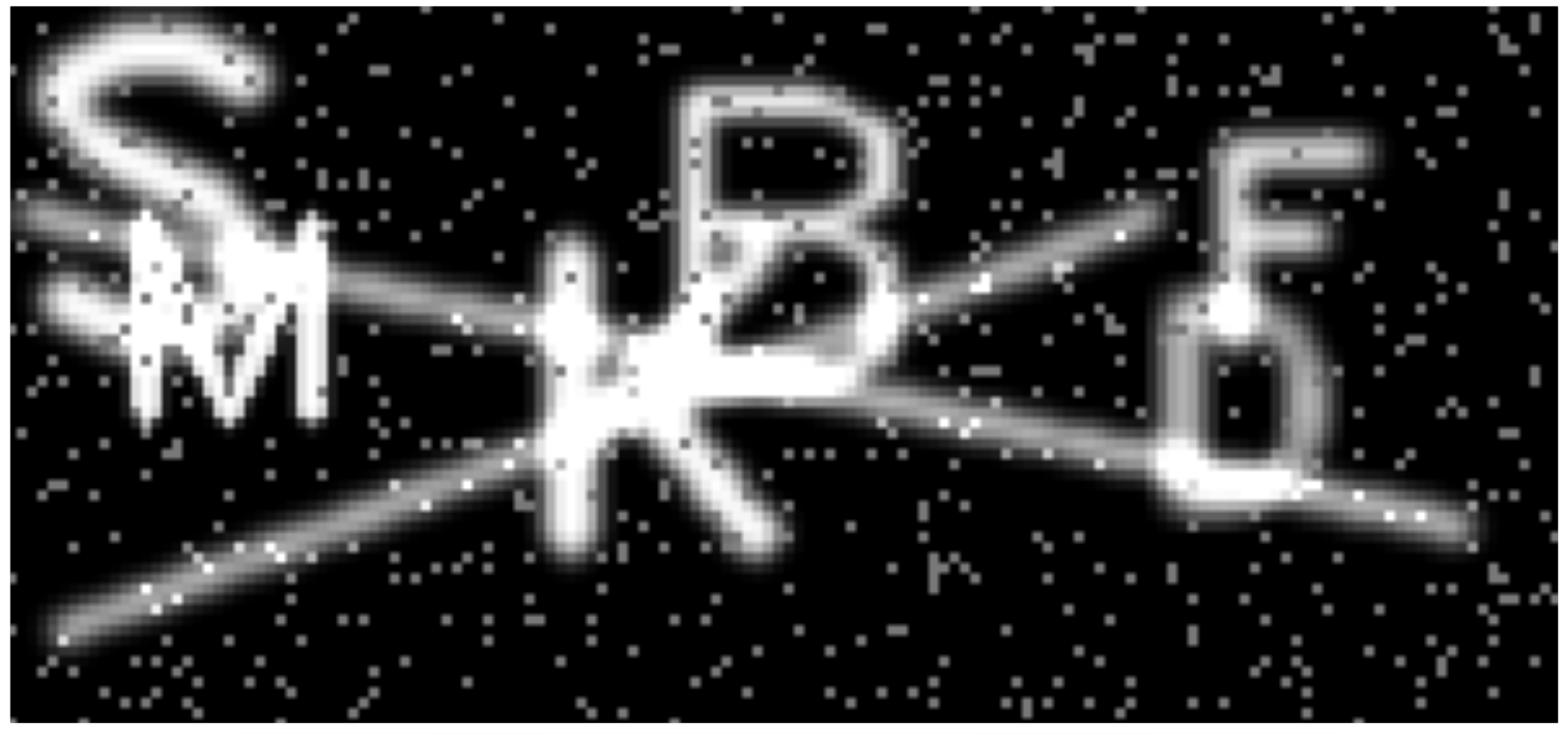


Model for Characters

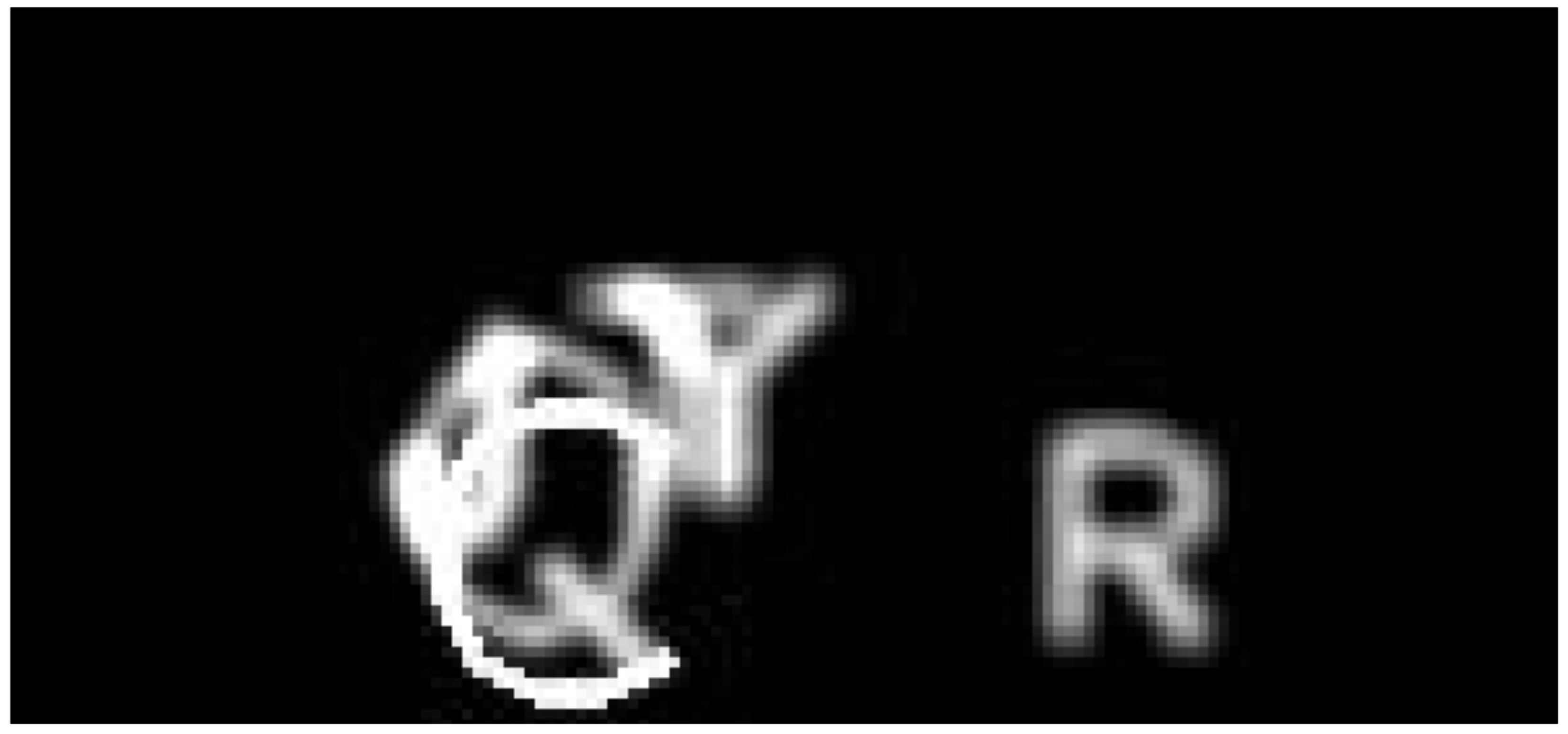
```
(defn sample-char []
 {:symbol (sample (uniform ascii))
  :x (sample (uniform-cont 0.0 1.0))
  :y (sample (uniform-cont 0.0 1.0))
  :scale (sample (beta 1 2))
  :weight (sample (gamma 2 2))
  :blur (sample  (gamma 1 1))})
```

Generative model for Captcha-breaking

Target Image



Samples from Program

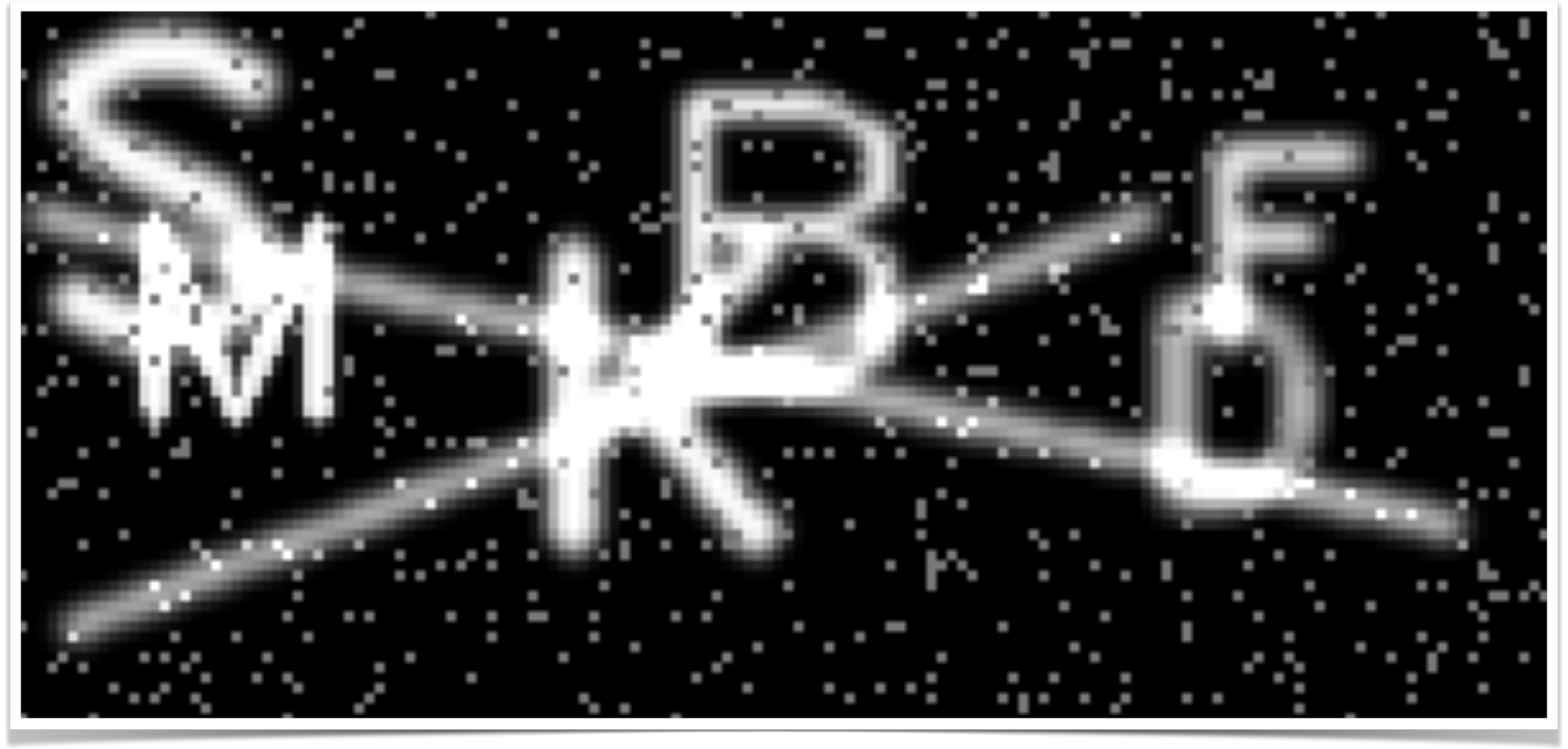


Model for Characters

```
(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))))
```

Generative model for Captcha-breaking

Target Image



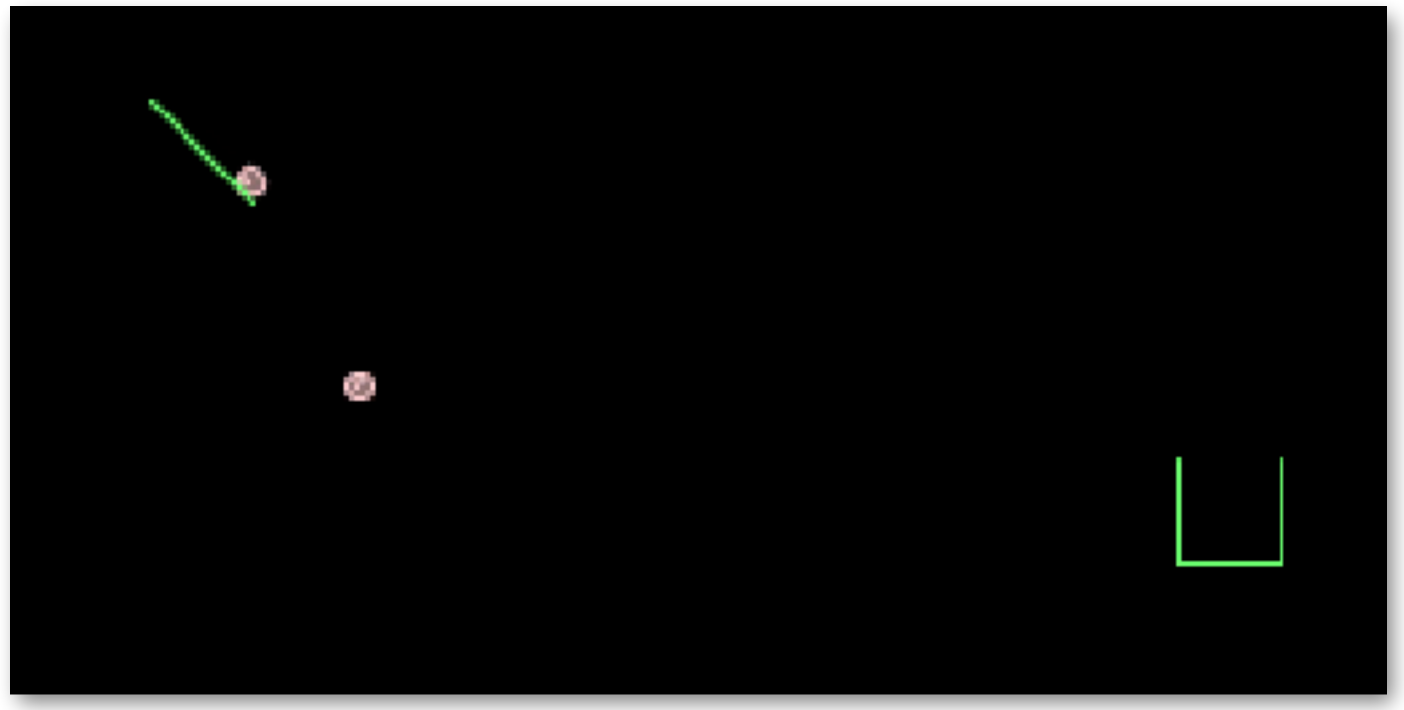
Samples from Program

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

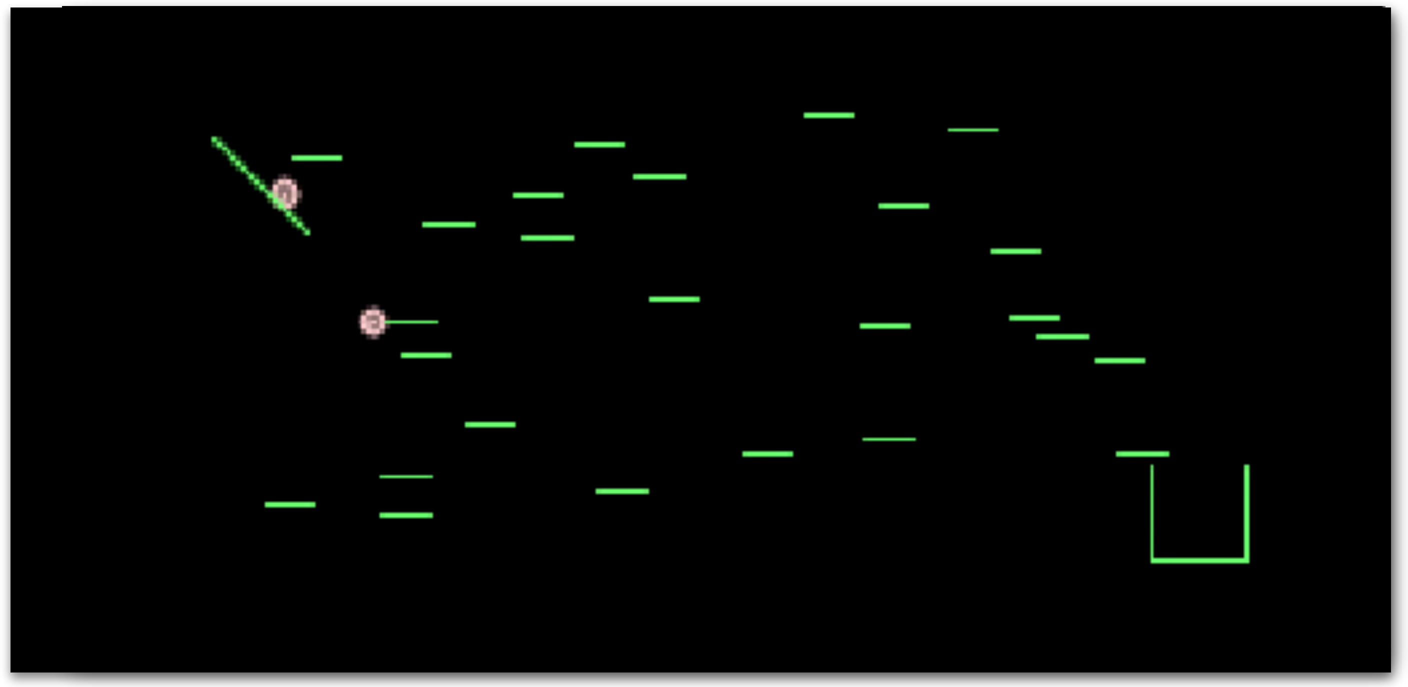
```
(defquery arrange-bumpers []
    (let [bumper-positions []

          ;; 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 :num-balls-in-box num-balls-in-box)
      (predict :bumper-positions bumper-positions)))
```

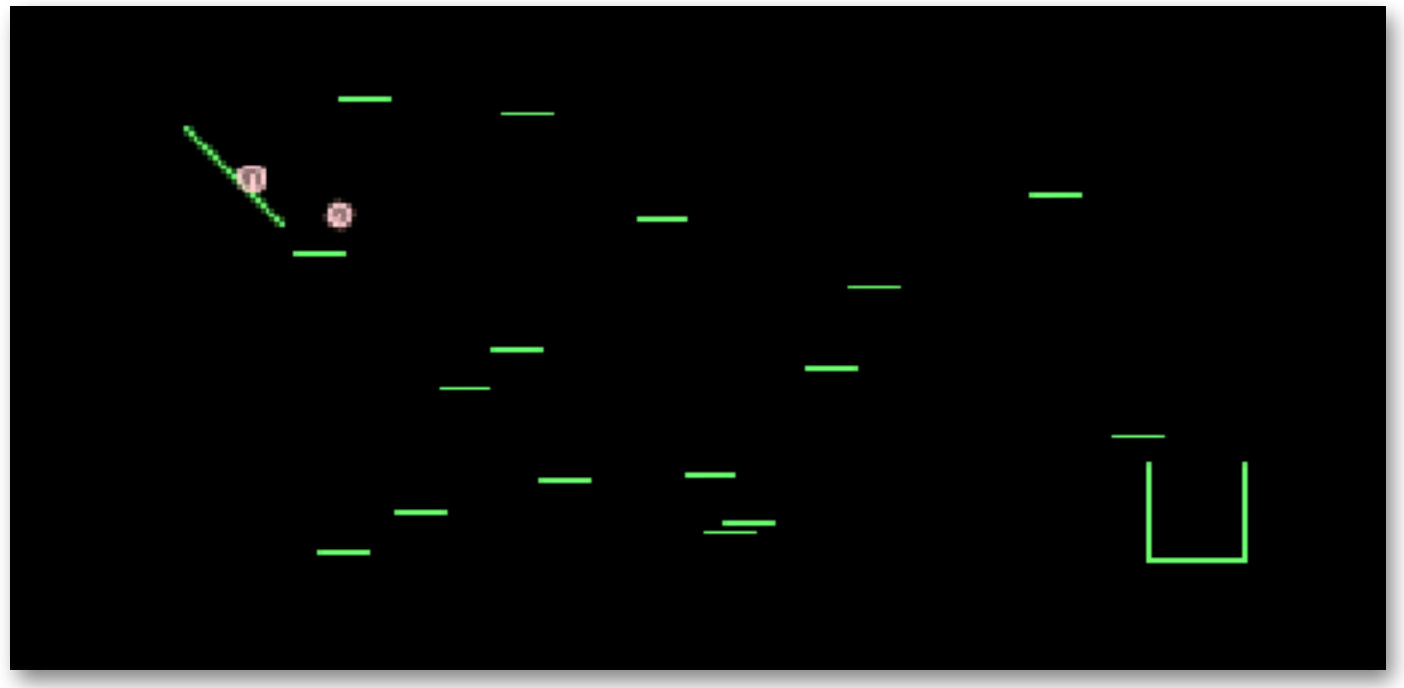


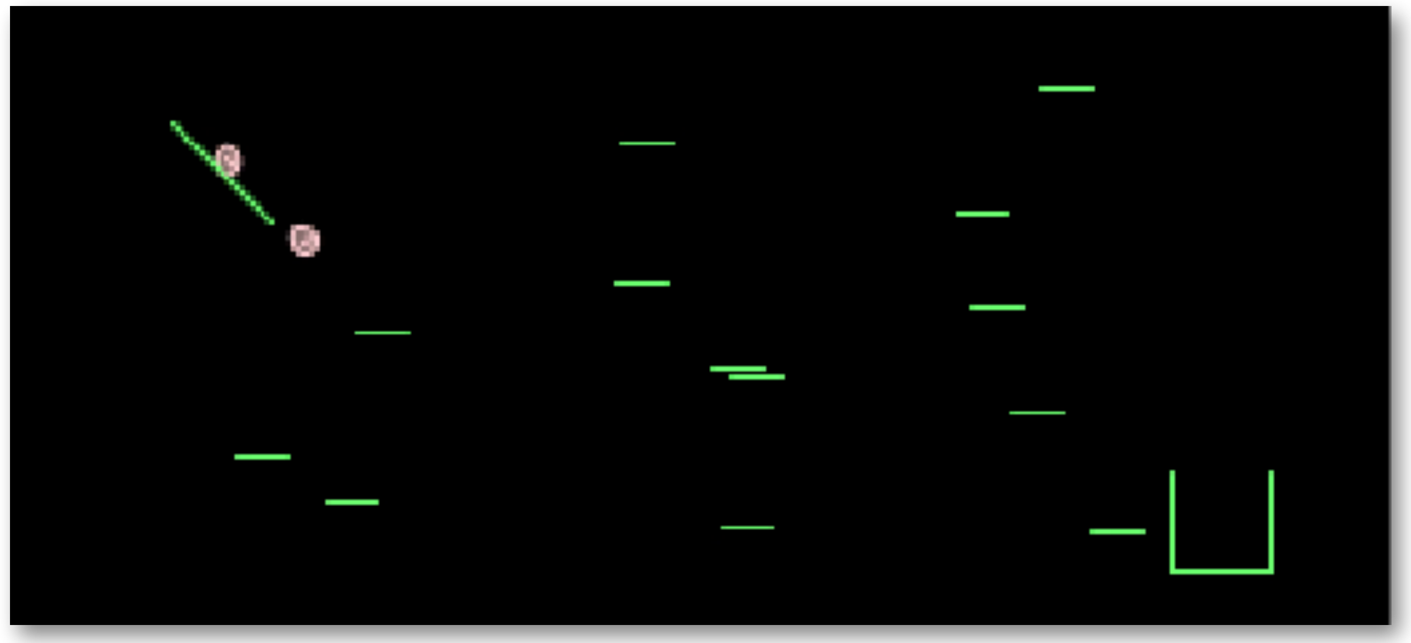
What if we want a “world” that puts ~20% of balls in box?

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 :num-balls-in-box num-balls-in-box)
      (predict :bumper-positions bumper-positions)))
```

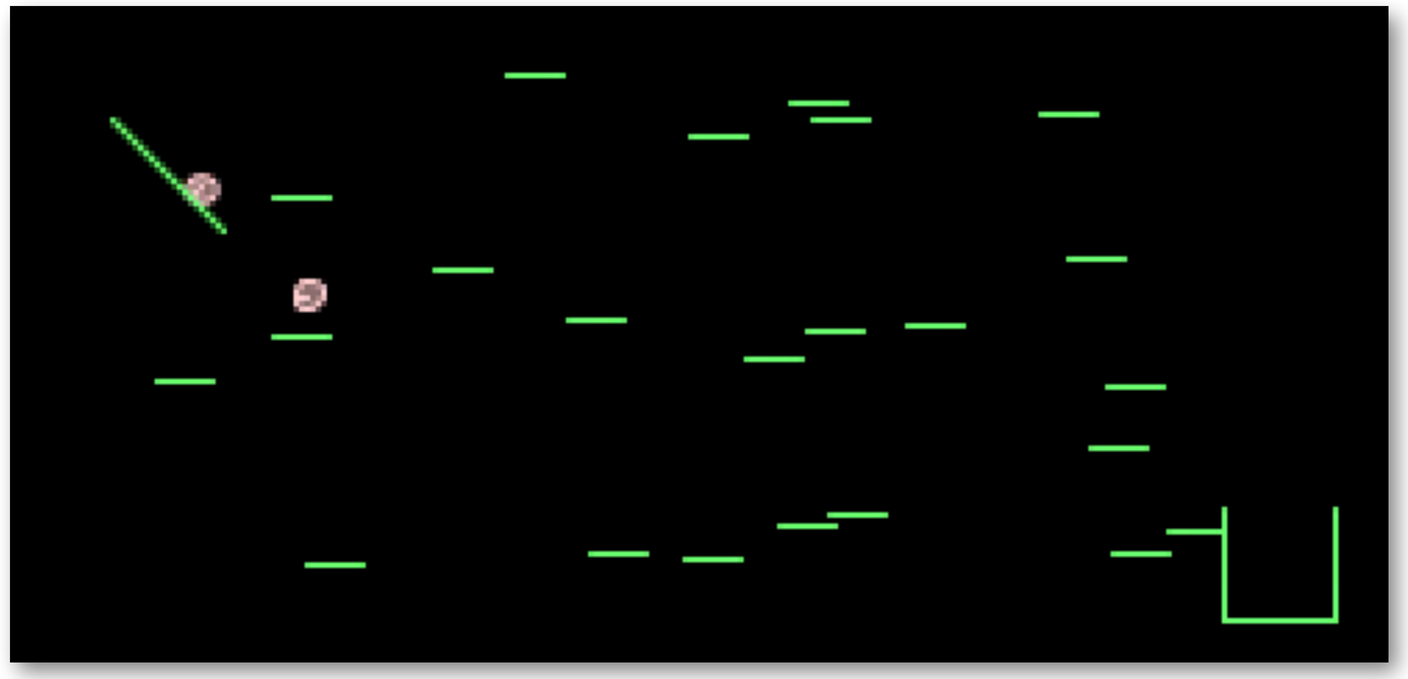


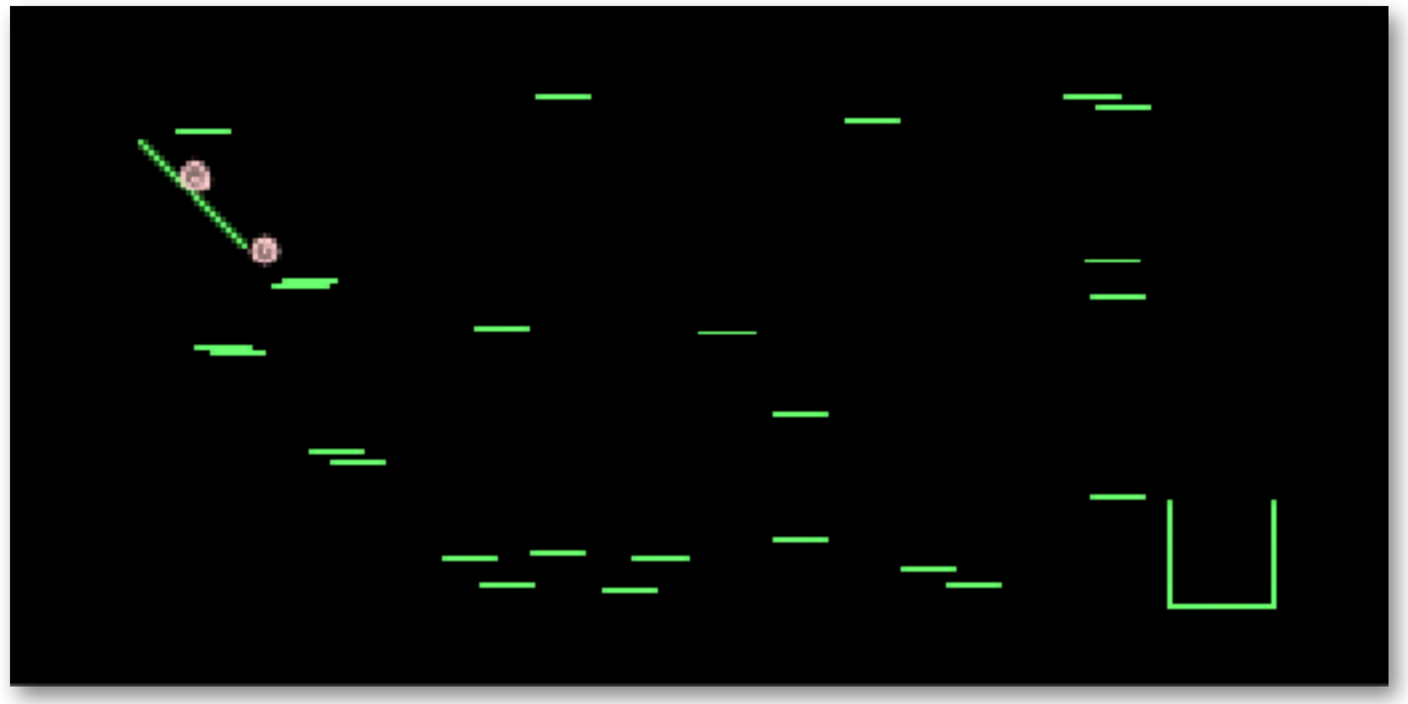


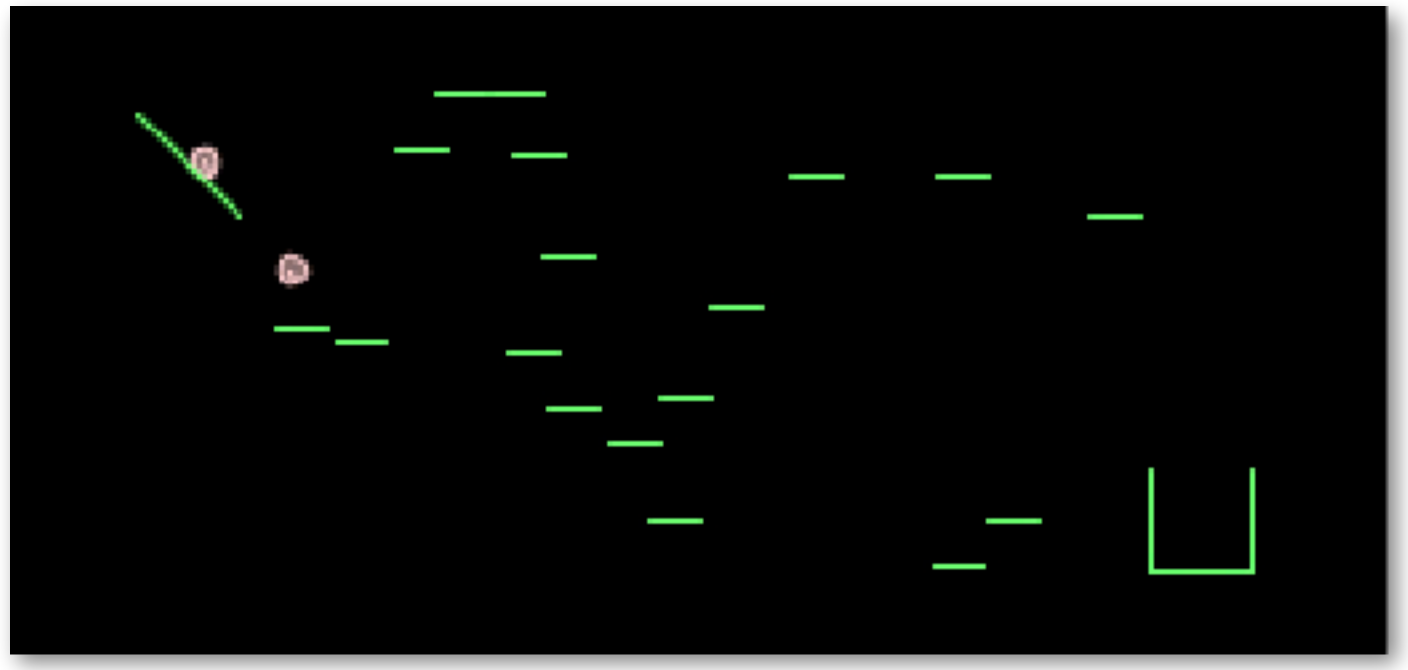


Constrained Stochastic Simulation

```
(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 :num-balls-in-box num-balls-in-box)
      (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:

```
(sample ...)                    (observe ...)
```

- **Q:** what kinds of inference algorithms can we develop and implement using **just this** as our interface?

Inference over partial program executions

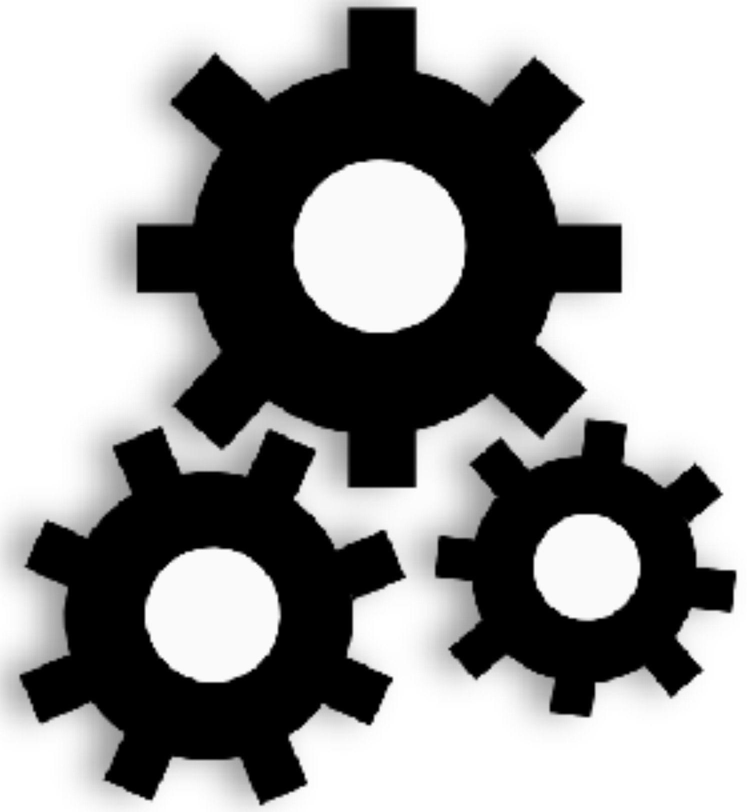
From the perspective of the inference engine, what happens as a program runs?

- 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*

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

Interaction between inference engine and model?

Inference engine (controller)



Program / model:

```
(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)
```

- Inference engine launches (instances of the) program
- `sample` and `observe` “checkpoints” yield control back to engine
- Engine updates internal state, and resumes program execution
- Program yields result to inference engine upon termination

Implementing “checkpoints”: continuations

How do continuations work?

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

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

Second cont.

First continuation

```
;; 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))))))))
```

$$xx = x^2$$

$$yy = y^2$$

$$xxyy = xx + yy$$

$$\cdot = \sqrt{xxyy}$$

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
    (fn [dist1]
      (sample& dist1
        (fn [p] ((fn [p k2]
                   (flip& p
                     (fn [dist2]
                       (observe& dist2 outcome
                         (fn []
                           (predict& :p p k2))))))
                 p k1))))))
```

```
(let [u (uniform-continuous 0 1)
      p (sample u)
      dist (flip p)]
  (observe dist outcome)
  (predict :p p))
```

```
;; 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
start $\mathcal{P}$
continue $\mathcal{P}$
continue $\mathcal{P}$
terminate $\mathcal{P}$
"Backend"
(k x)
(k)
(k)
sample
(f, θ, k)
observe
(g, ϕ, y, k)
predict
(z, k)
terminate

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?

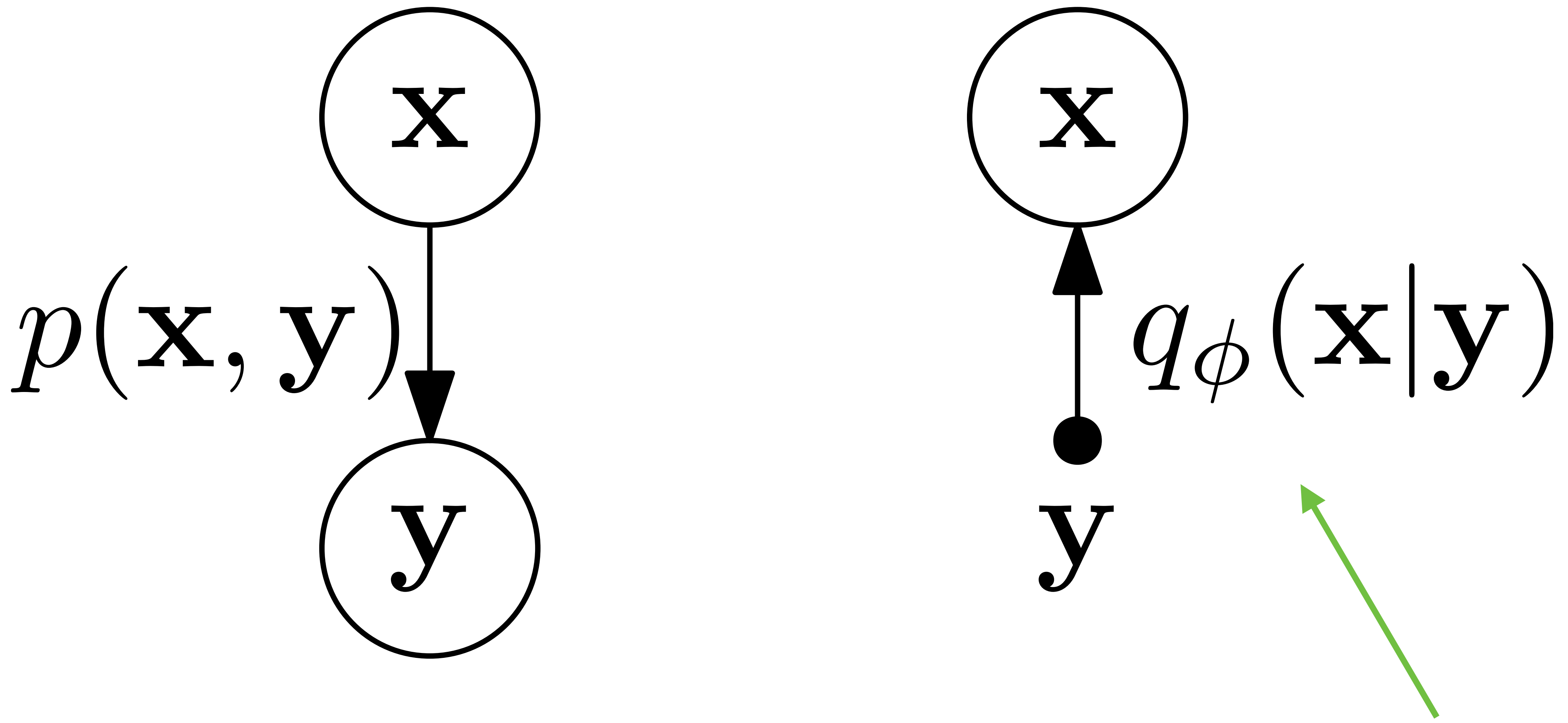
Trends in probabilistic programming

Inference?

	Yes	No
One-shot	Probabilistic Programming	?
Repeated	Amortized Inference	Un- and Semi-Supervised Deep Learning

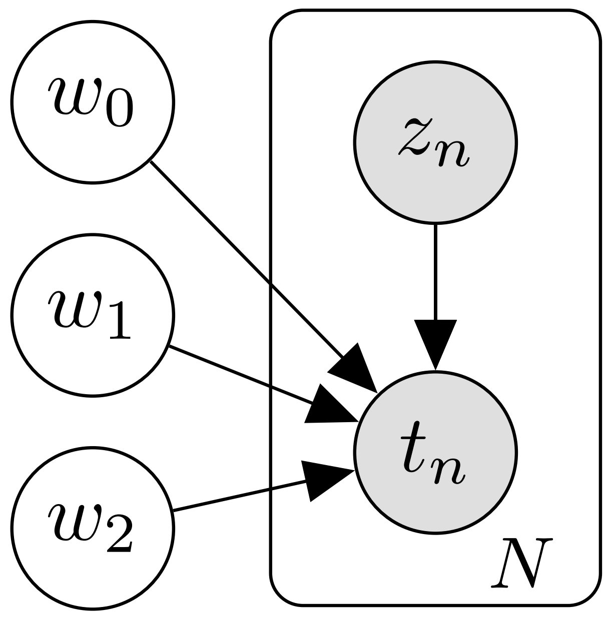
Have fully-specified model?

Amortized inference

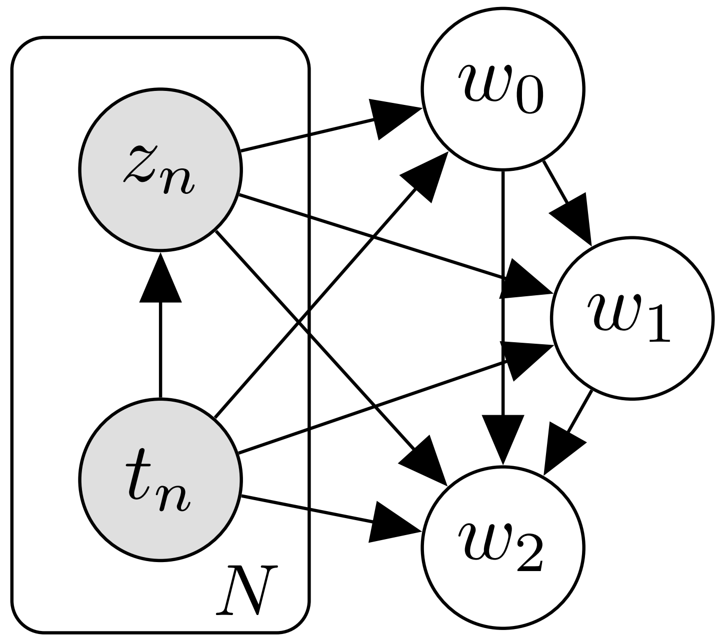


Can we learn this directly?

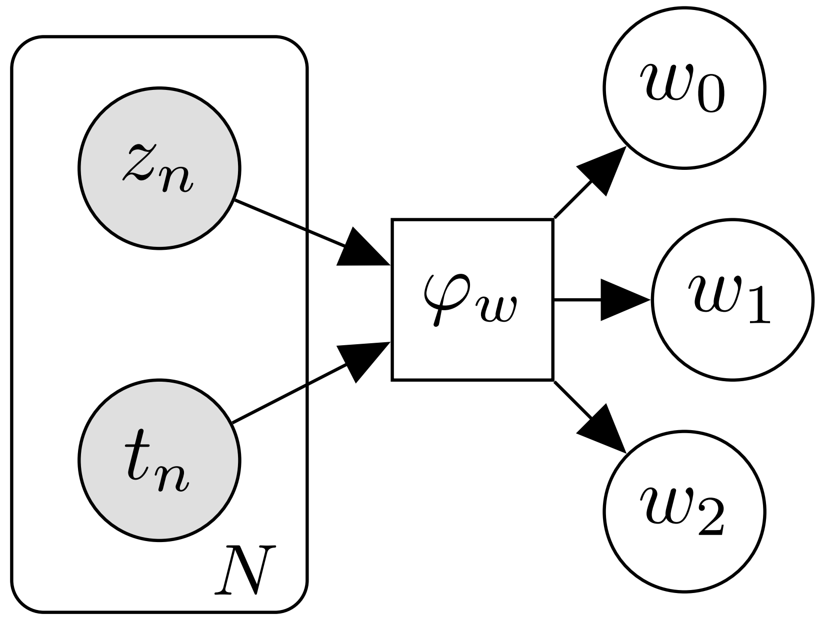
Inference networks as proposal distributions



A probabilistic model **generates data**



An inverse model **generates latents**



Can we **learn how to sample** from the inverse model?

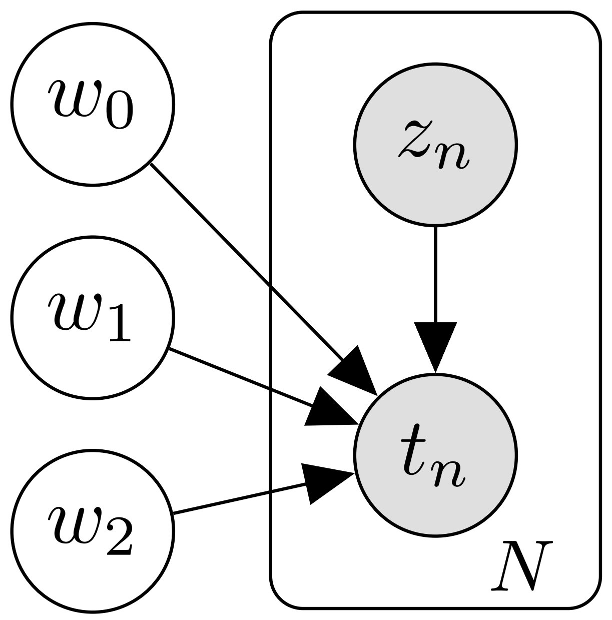
Learning an importance sampling proposal for a single dataset

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

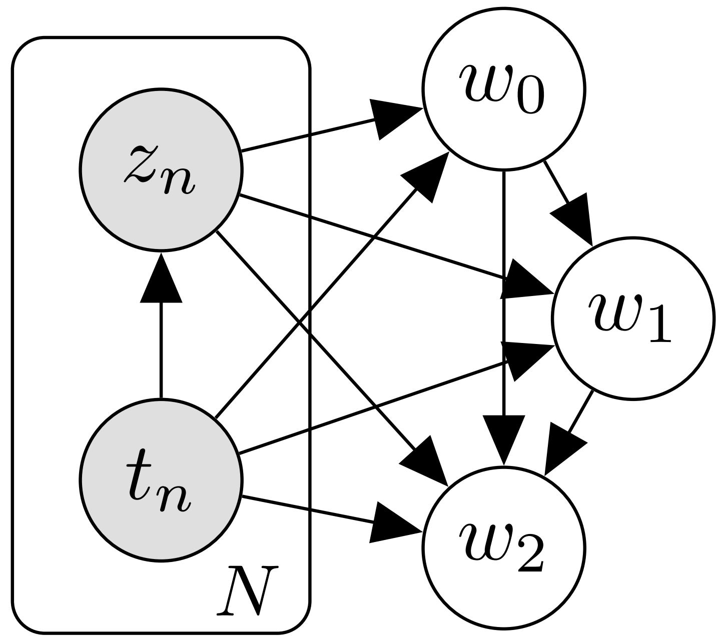
Single dataset $\mathbf{y}$: $\underset{\lambda}{\operatorname{argmin}}\, D_{KL}(\pi || q_\lambda)$

fit λ to learn an importance sampling proposal

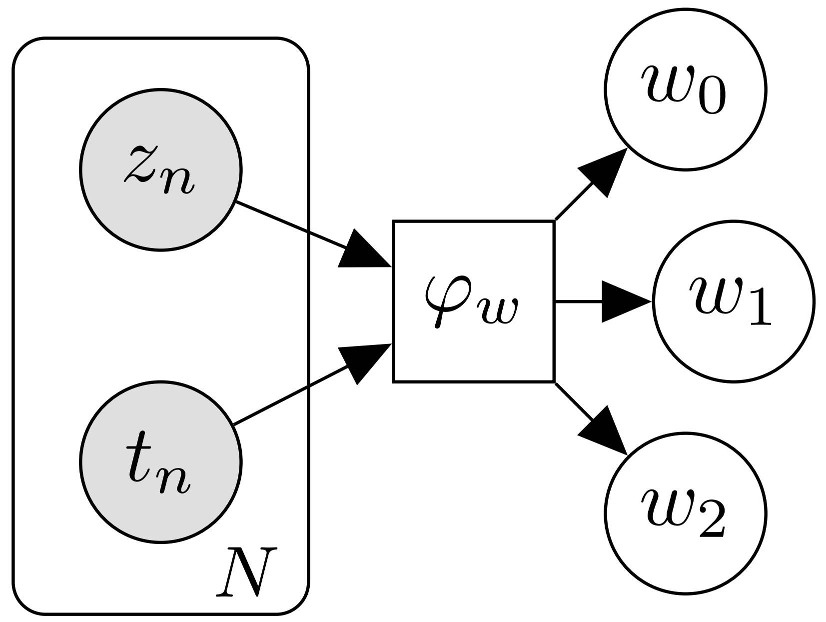
Inference networks as proposal distributions



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 $\pi(\mathbf{x}) = p(\mathbf{x}|\mathbf{y})$, approximating family $q(\mathbf{x}|\lambda)$

Averaging over all possible datasets:

$$\lambda = \varphi(\eta, \mathbf{y})$$

learn a mapping from arbitrary datasets to λ

$$\underset{\eta}{\operatorname{argmin}}\, \mathbb{E}_{p(\mathbf{y})}\left[D_{KL}(\pi || q_{\varphi(\eta,\mathbf{y})})\right]$$

Training inference network on synthetic data

Averaging over all possible datasets:

$$\lambda \;=\; \varphi(\eta, \mathbf{y})$$

$$\underset{\eta}{\operatorname{argmin}}\, \mathbb{E}_{p(\mathbf{y})}\left[D_{KL}(\pi || q_{\varphi(\eta,\mathbf{y})})\right]$$

expectation over any data we might observe

New objective function, upper-level parameters:

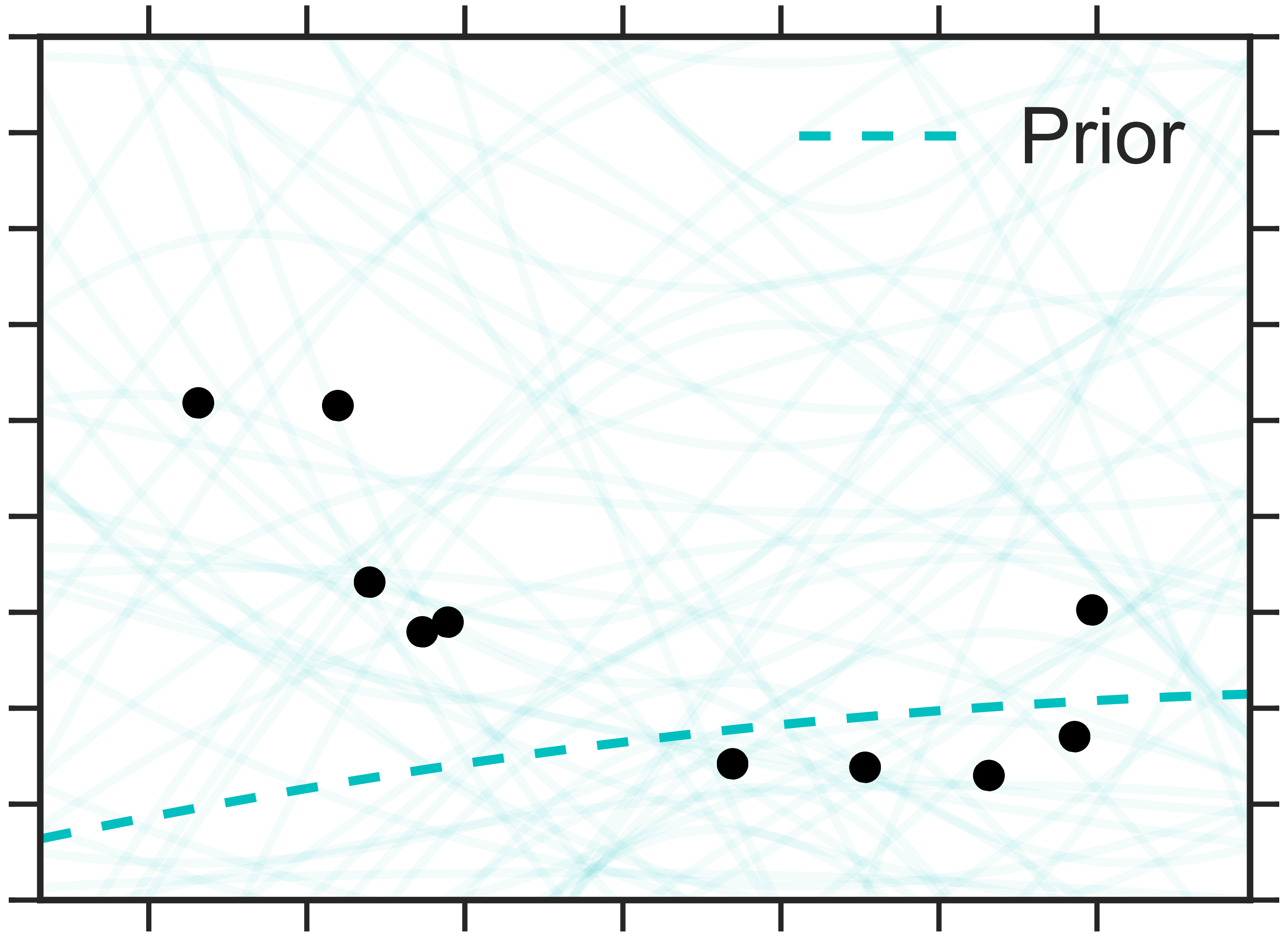
$$\begin{aligned}\mathcal{J}(\eta) &= \int D_{KL}(\pi || q_\lambda) p(\mathbf{y}) \mathrm{d}\mathbf{y} \\ &= \int p(\mathbf{y}) \int p(\mathbf{x}|\mathbf{y}) \log\left[\frac{p(\mathbf{x}|\mathbf{y})}{q(\mathbf{x}|\varphi(\eta,\mathbf{y}))}\right] \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\ &= \mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[-\log q(\mathbf{x}|\varphi(\eta,\mathbf{y}))\right] + const.\end{aligned}$$

approximate with samples from the joint distribution

Tractable gradient! Can train entirely offline:

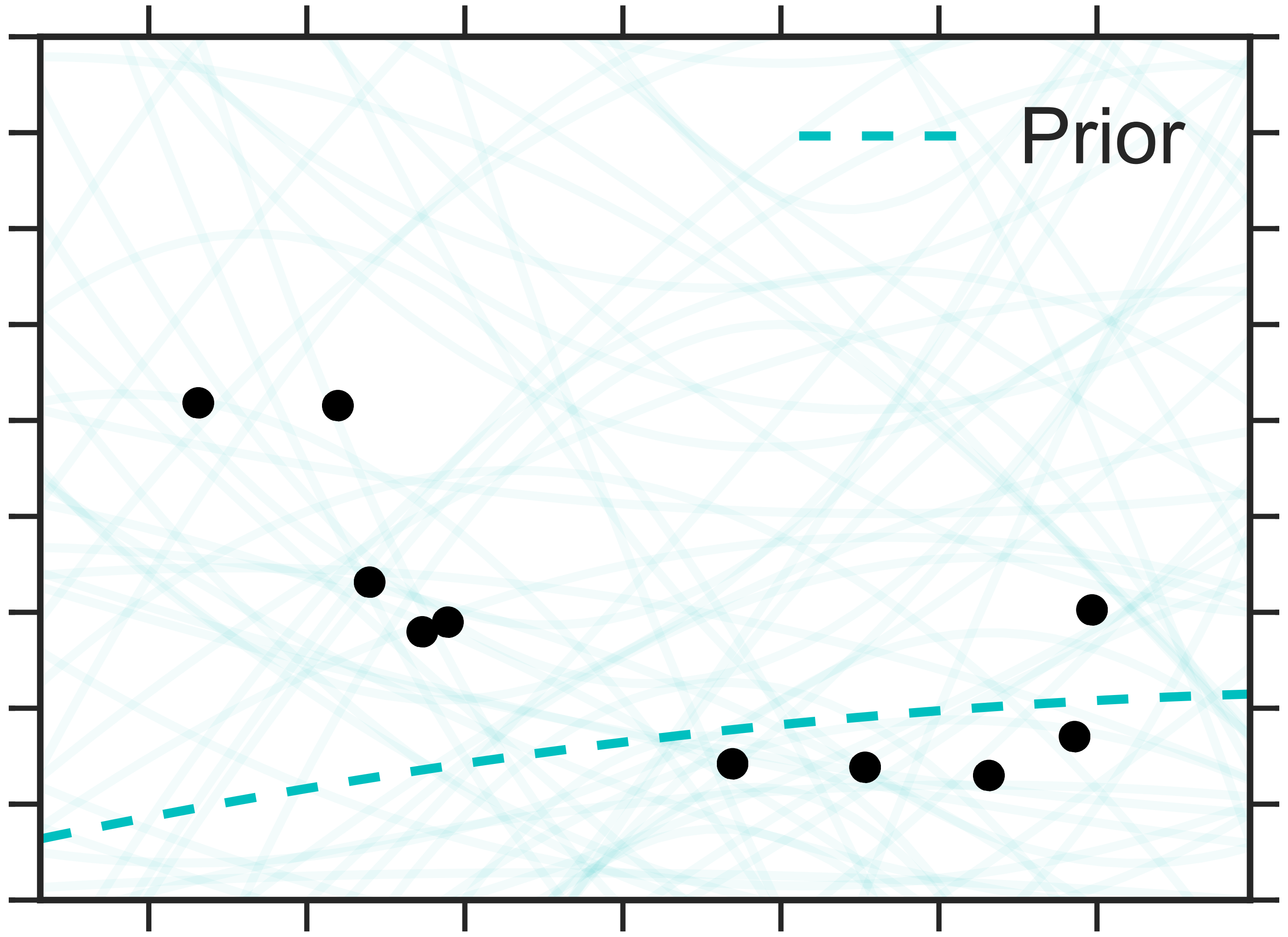
$$\nabla_\eta \mathcal{J}(\eta) = \mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[-\nabla_\eta \log q(\mathbf{x}|\varphi(\eta,\mathbf{y}))\right]$$

Non-conjugate polynomial regression

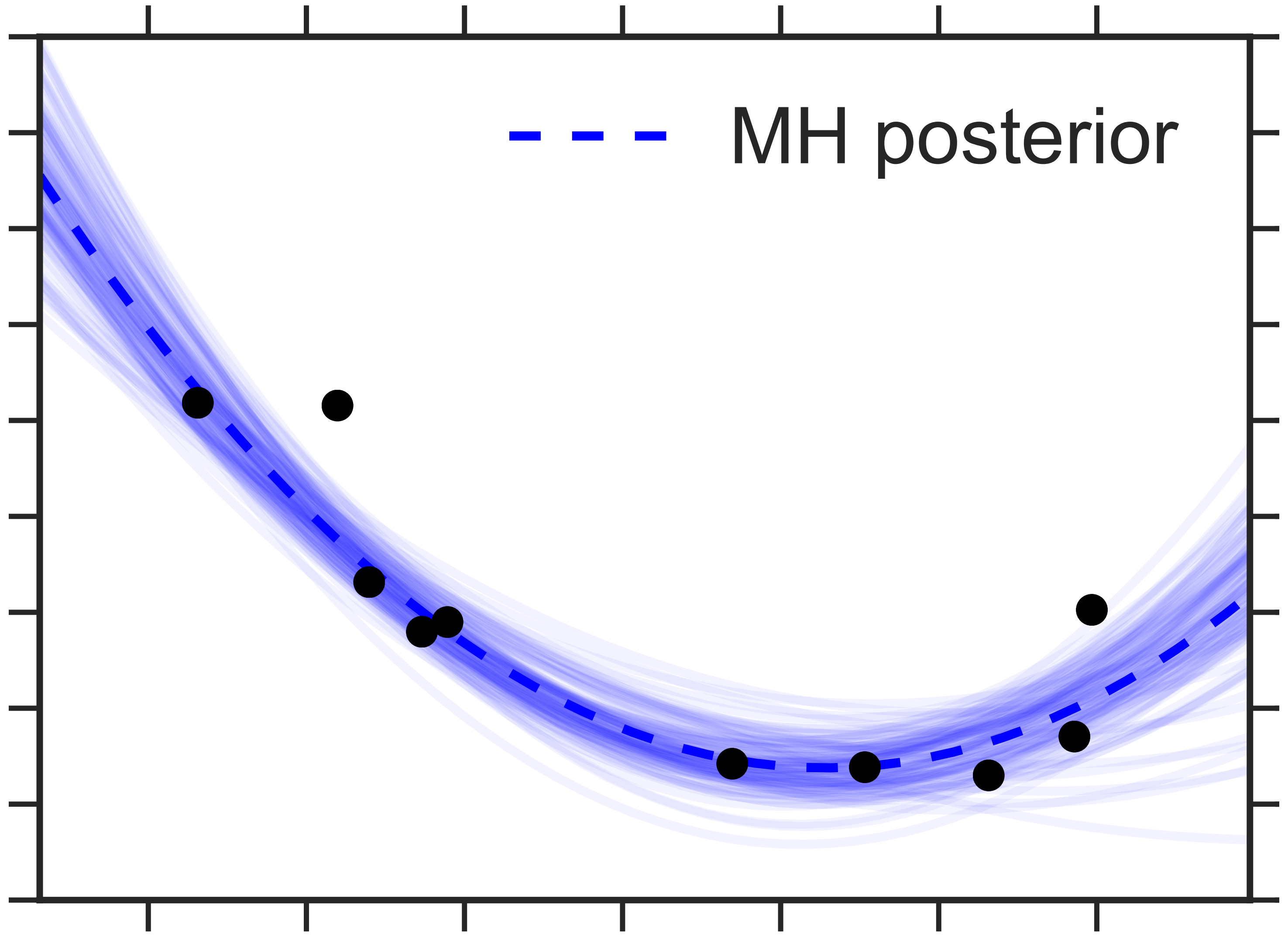


Samples from prior

Non-conjugate polynomial regression

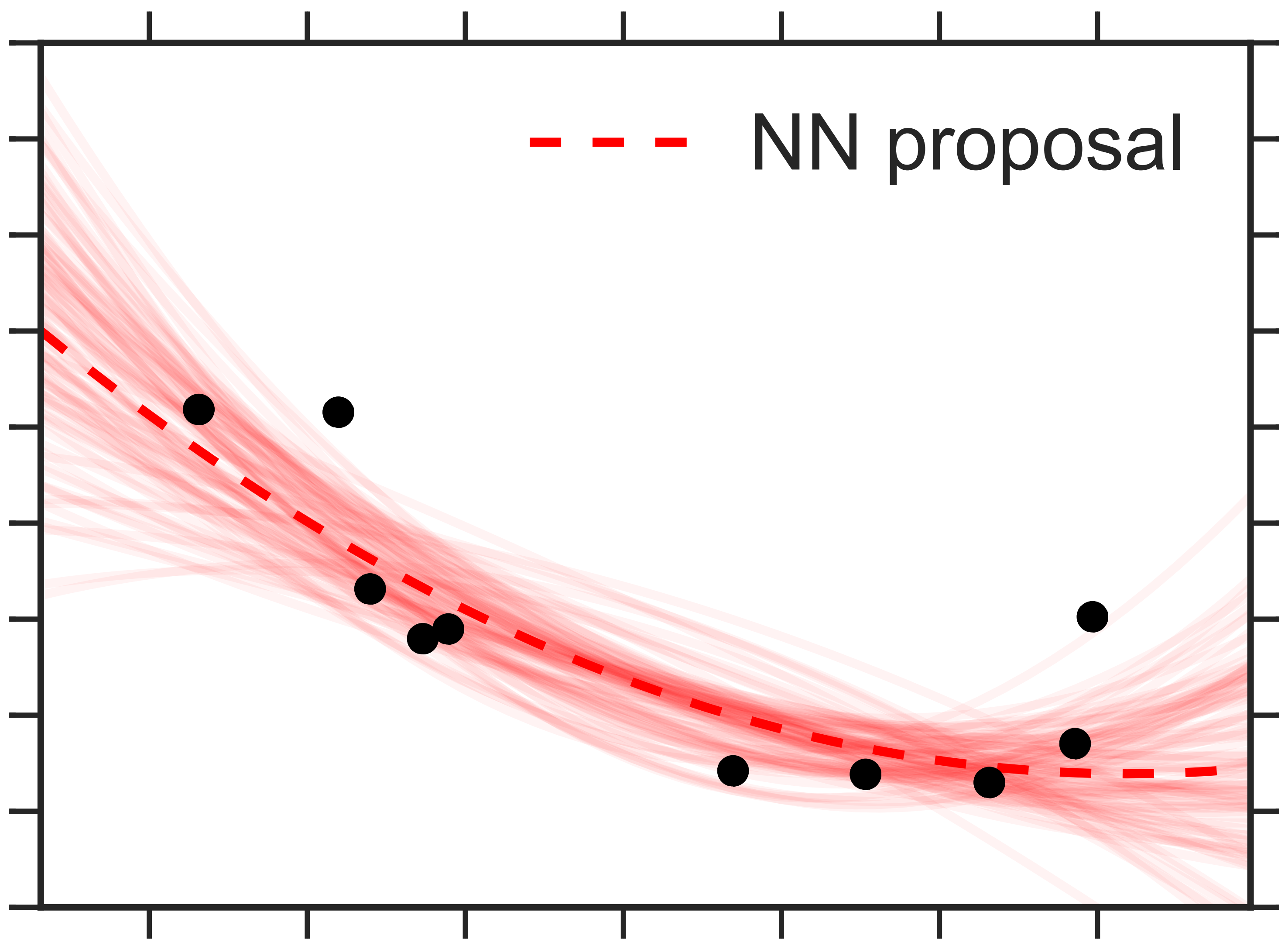


Samples from prior

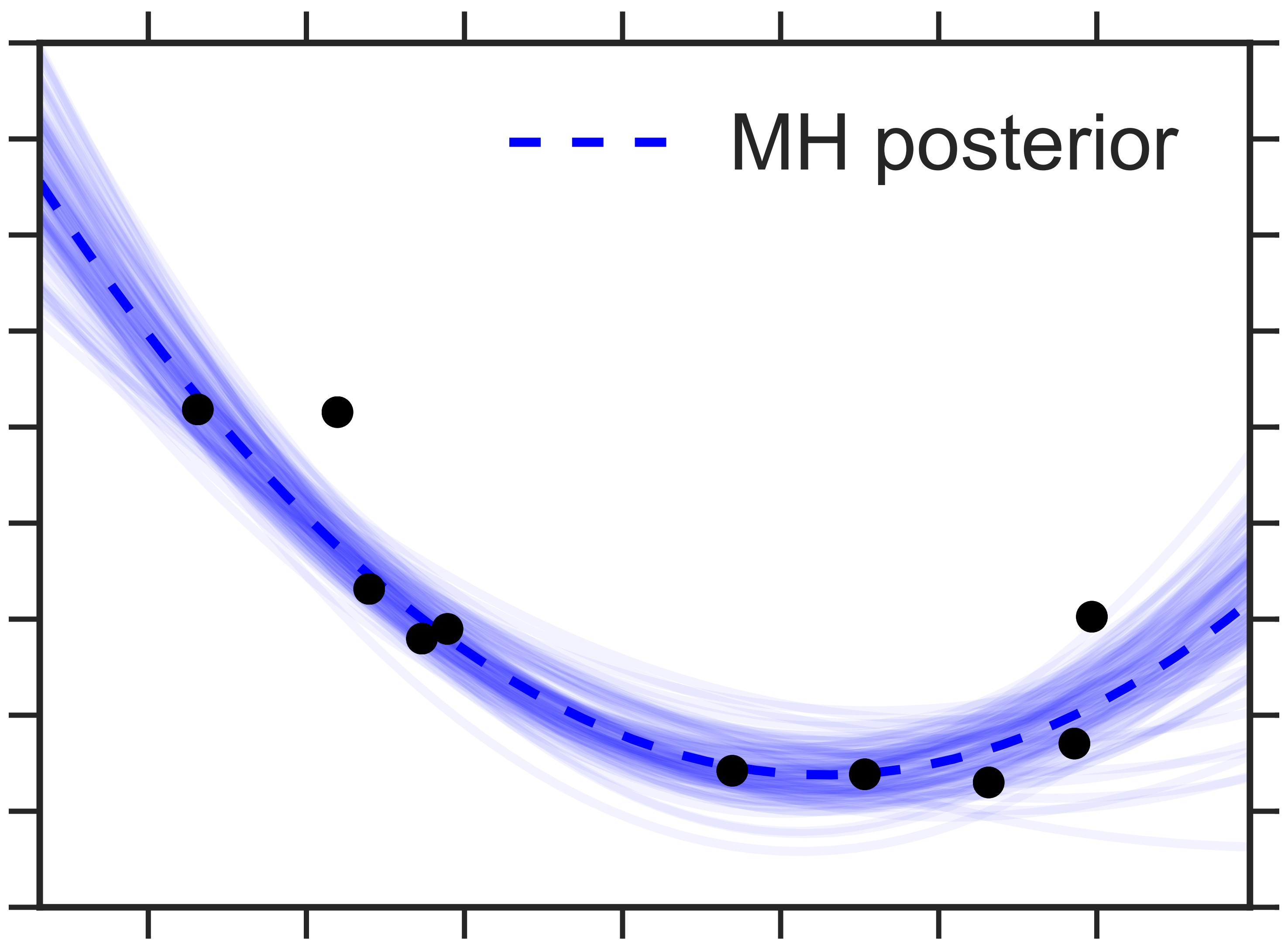


Metropolis-Hastings

Non-conjugate polynomial regression

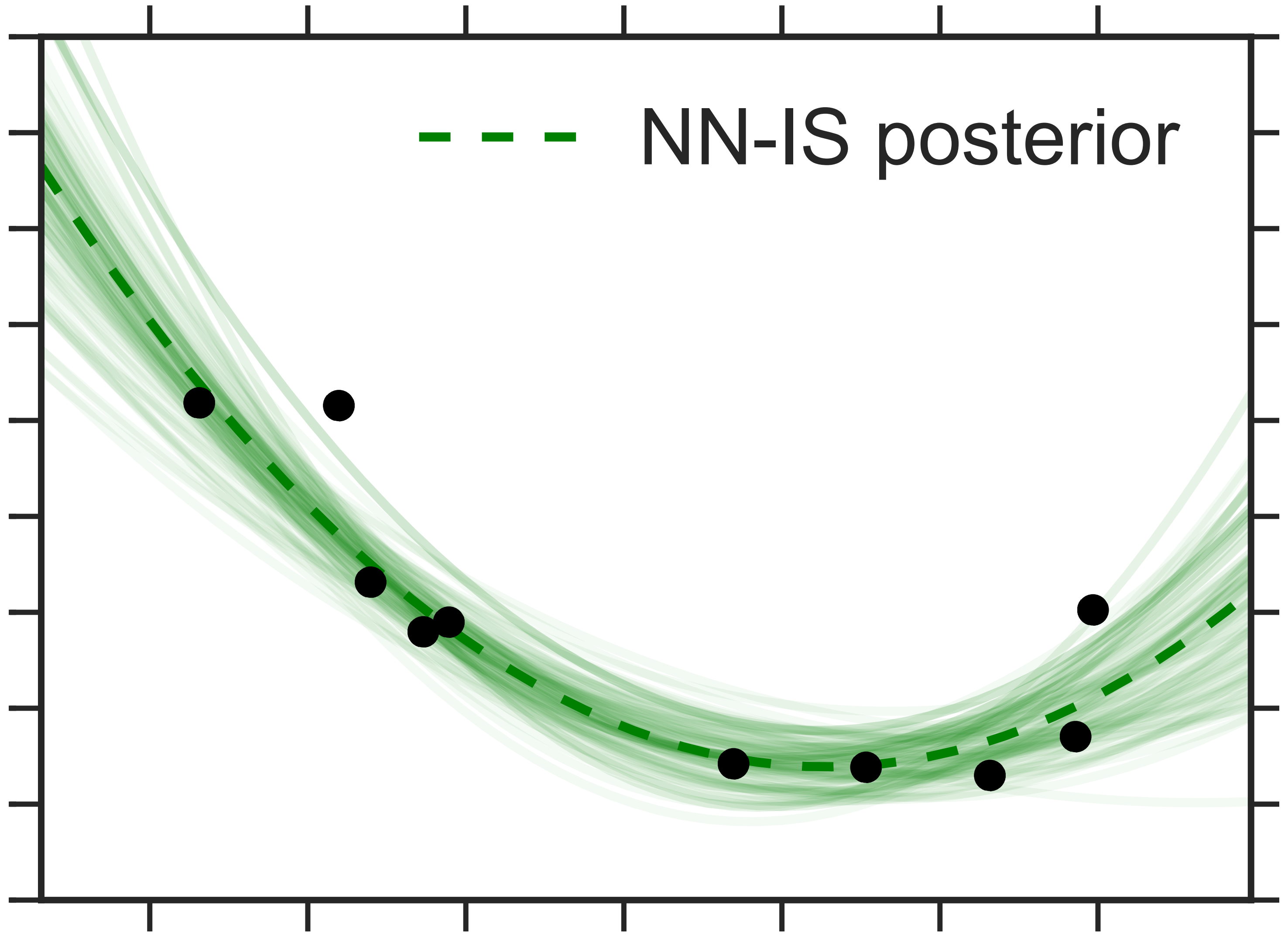


Samples from proposal

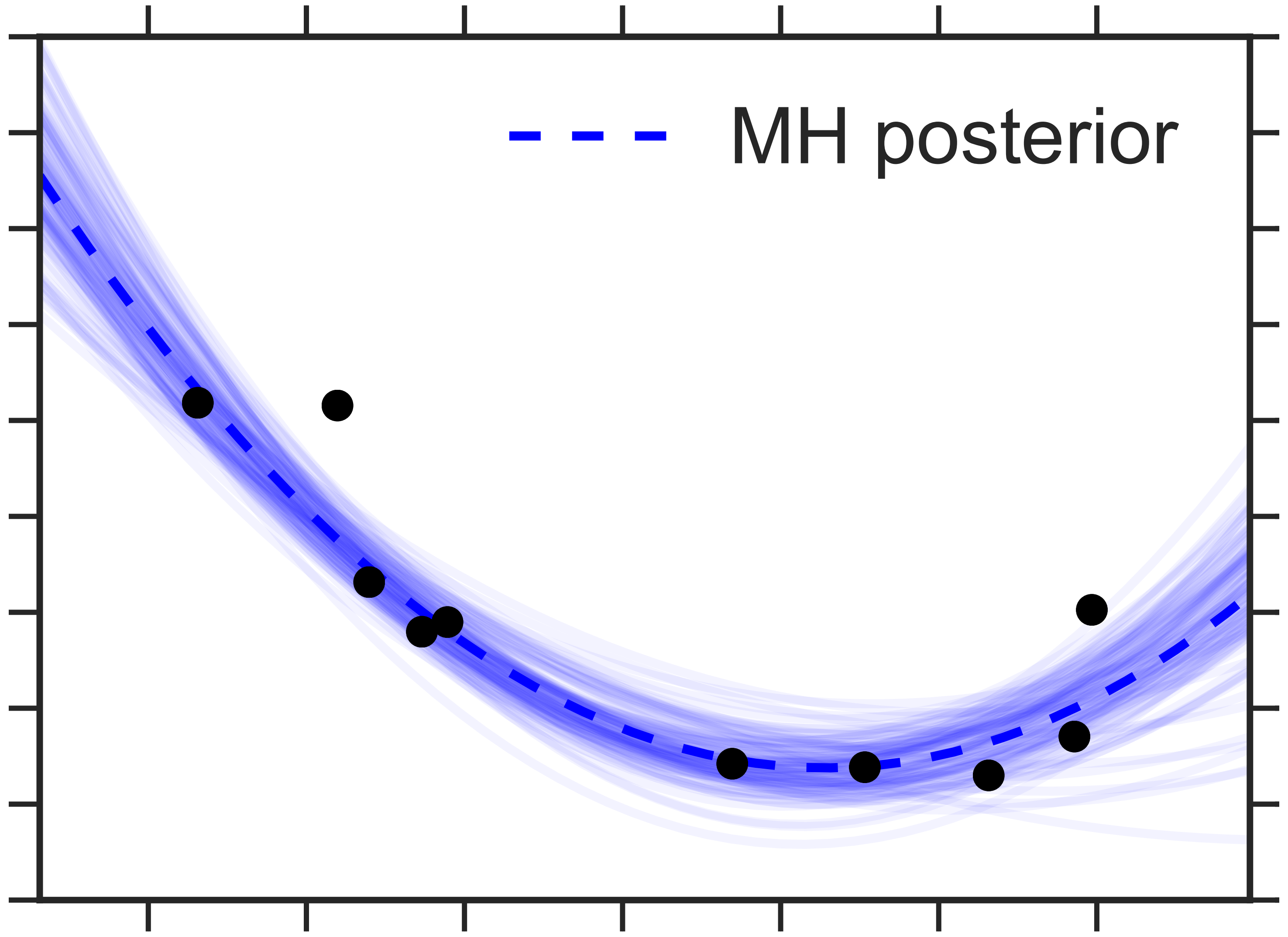


Metropolis-Hastings

Non-conjugate polynomial regression

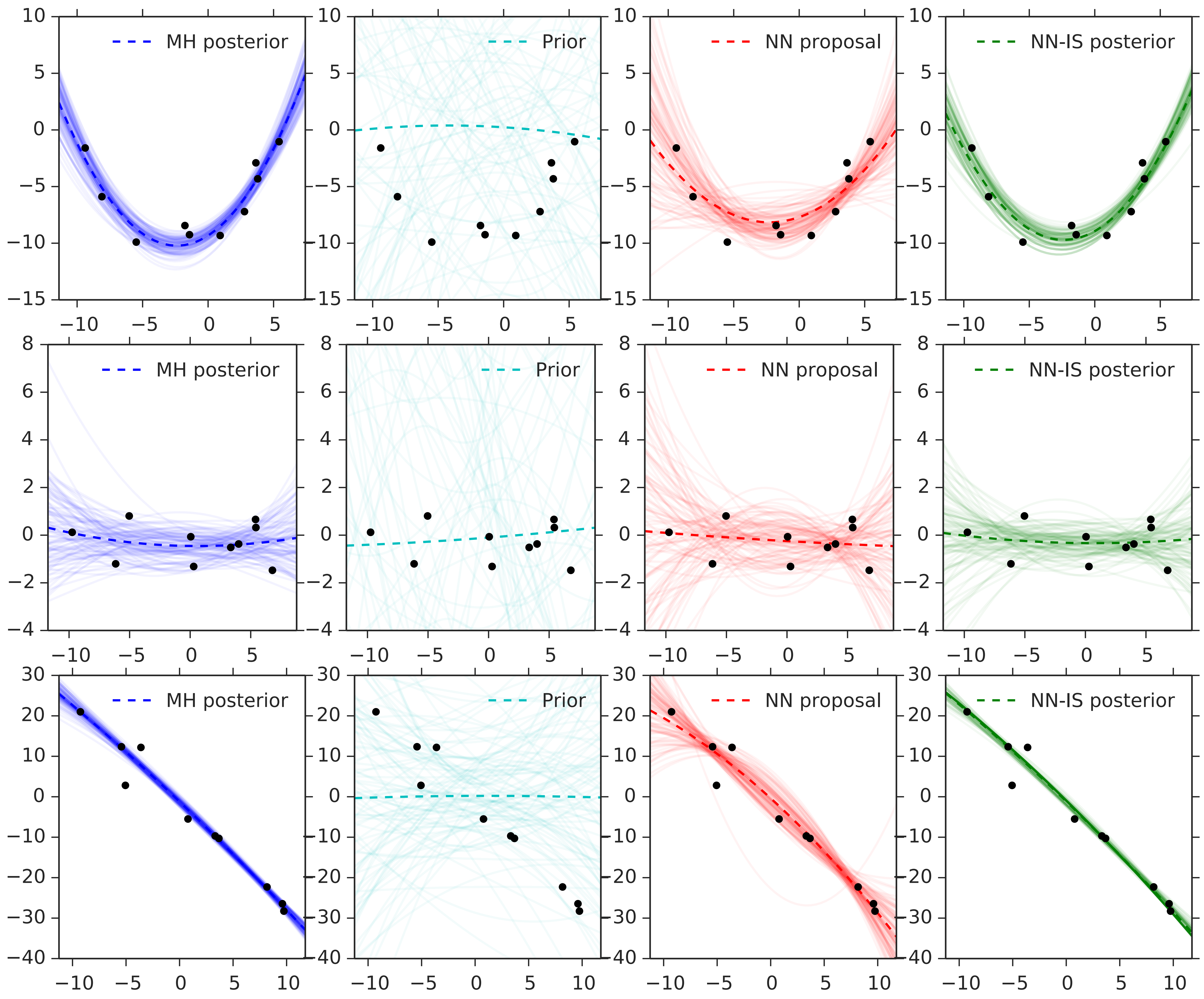


After importance weighting

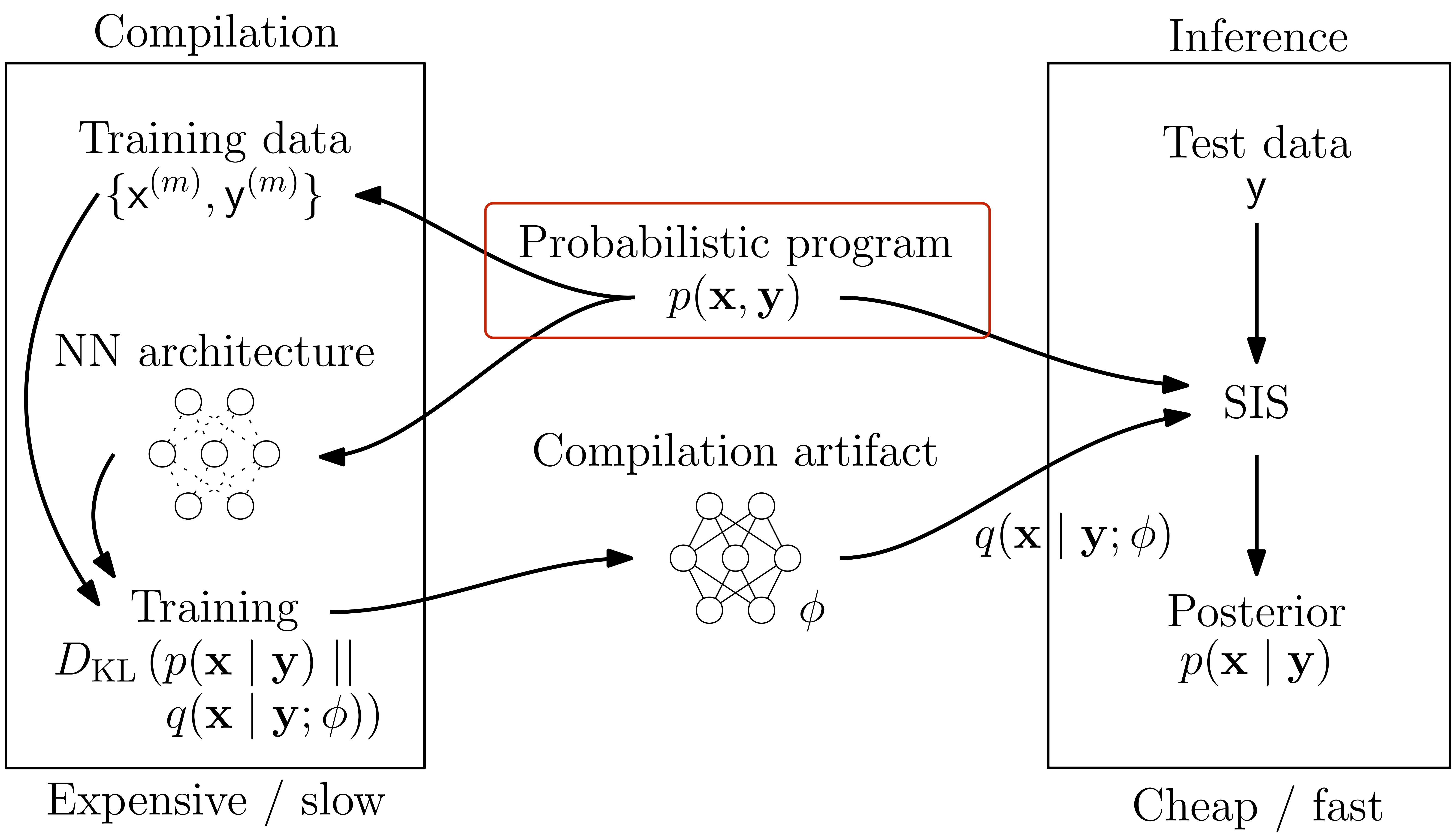


Metropolis-Hastings

Non-conjugate polynomial regression



Inference networks for probabilistic programs



Input: an inference problem denoted in a probabilistic programming language

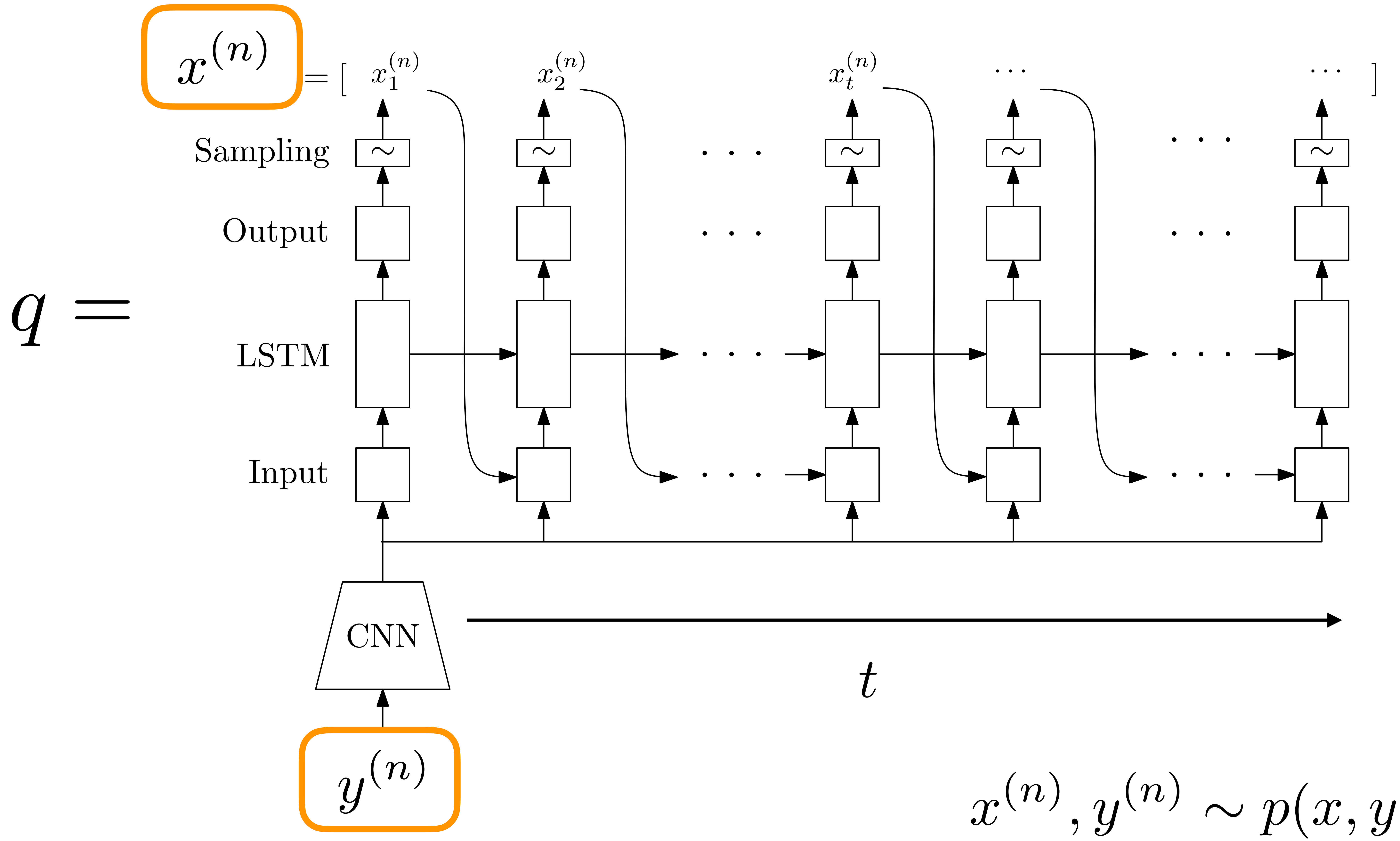
Output: a trained inference network (deep neural network “compilation artifact”)

Le TA, Baydin AG, Wood F. Inference Compilation and Universal Probabilistic Programming. AISTATS. 2017.

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

Generic structured proposal architecture

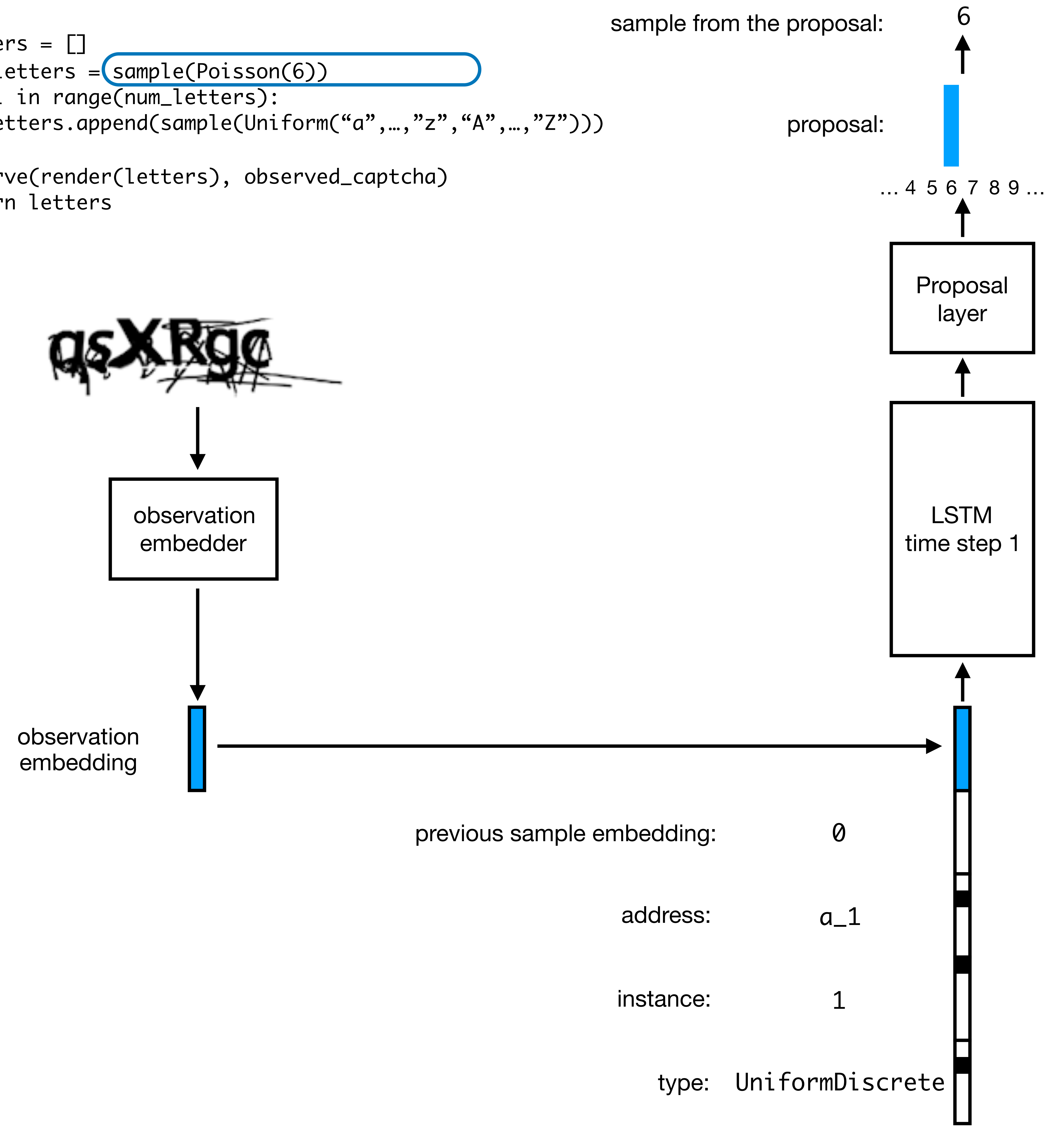


$$x^{(n)}, y^{(n)} \sim p(x, y)$$

synthetic data

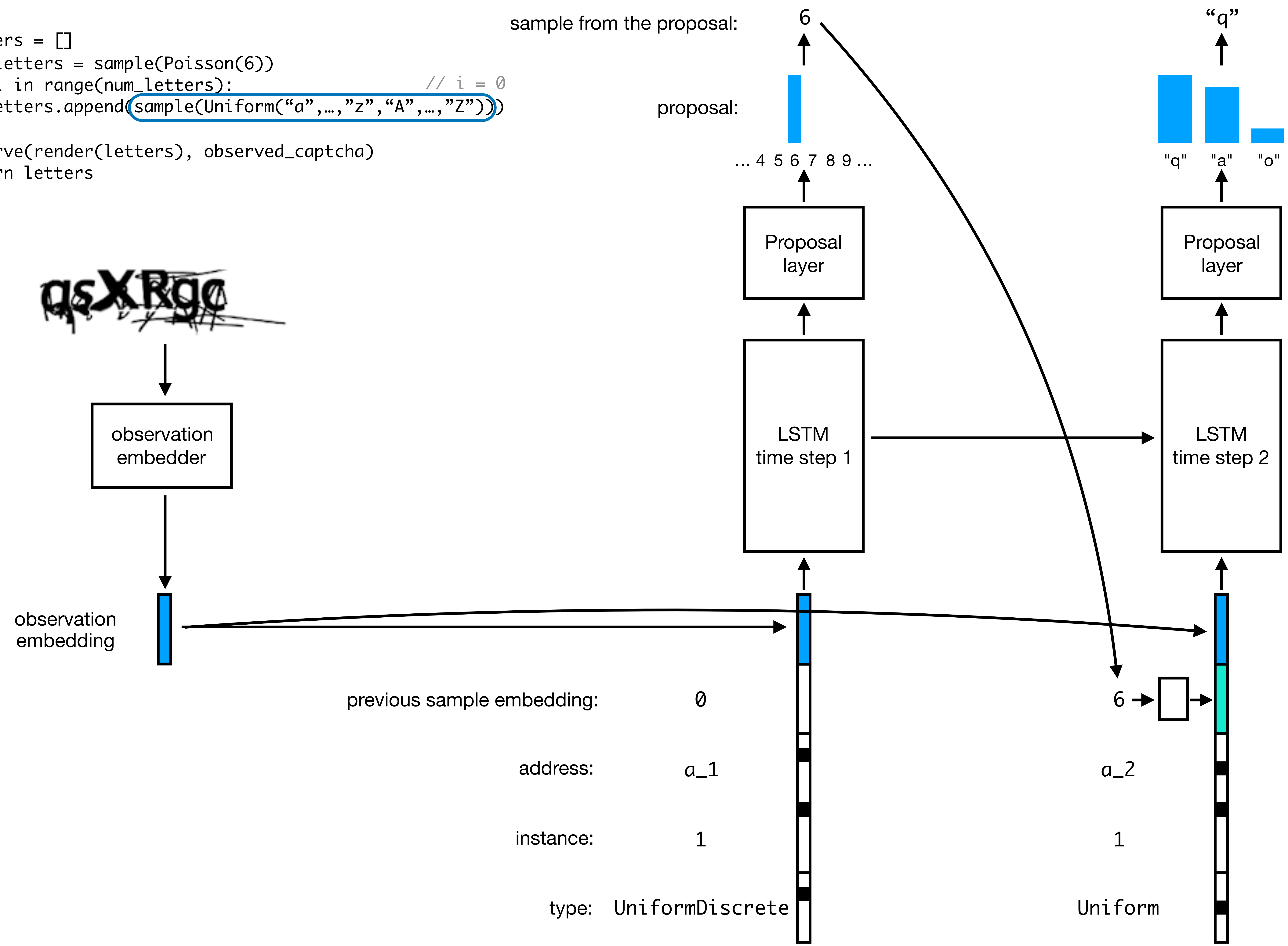
What does this look like for the CAPTCHA example?

```
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
```



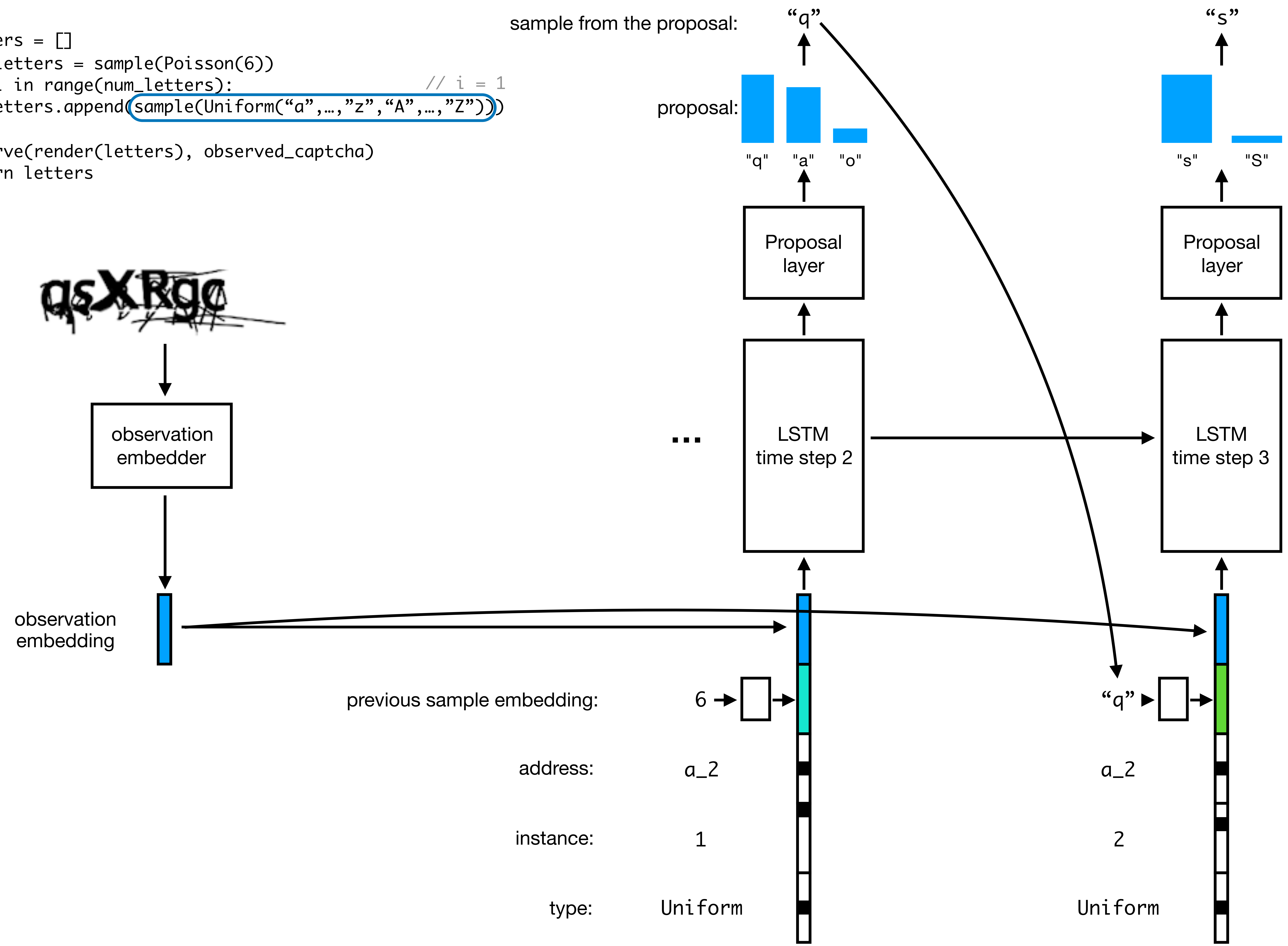
What does this look like for the CAPTCHA example?

```
letters = []
num_letters = sample(Poisson(6))
for i in range(num_letters):                              // i = 0
   letters.append(sample(Uniform(“a”,…,”z”,“A”,…,”Z”)))

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



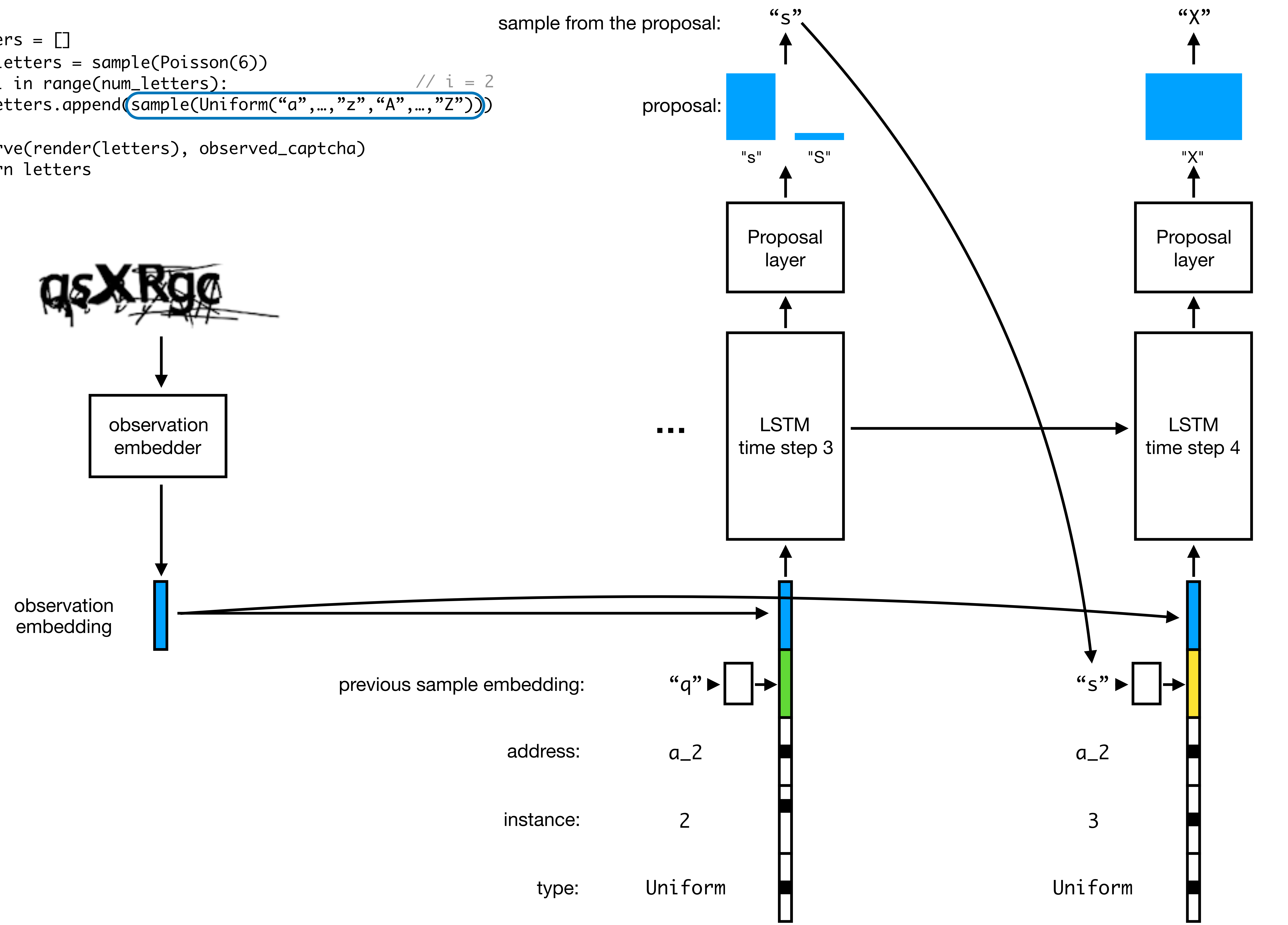
What does this look like for the CAPTCHA example?

```
letters = []
num_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?

```
letters = []
num_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
```



Solving Sudoku with diffusion models

6	3	1	7	1	9	5	1	6
6	**9**	1	7	8	5	5	1	3
6	3	1	5	8	6	6	1	4
9	1	4	1	**5**	7	3	**6**	8
7	1	5	**3**	**6**	8	9	4	2
3	**8**	**6**	9	2	7	7	5	**1**
8	6	9	2	**7**	1	1	3	**5**
1	2	7	8	3	5	**4**	9	6
2	5	**3**	1	**9**	4	1	7	6

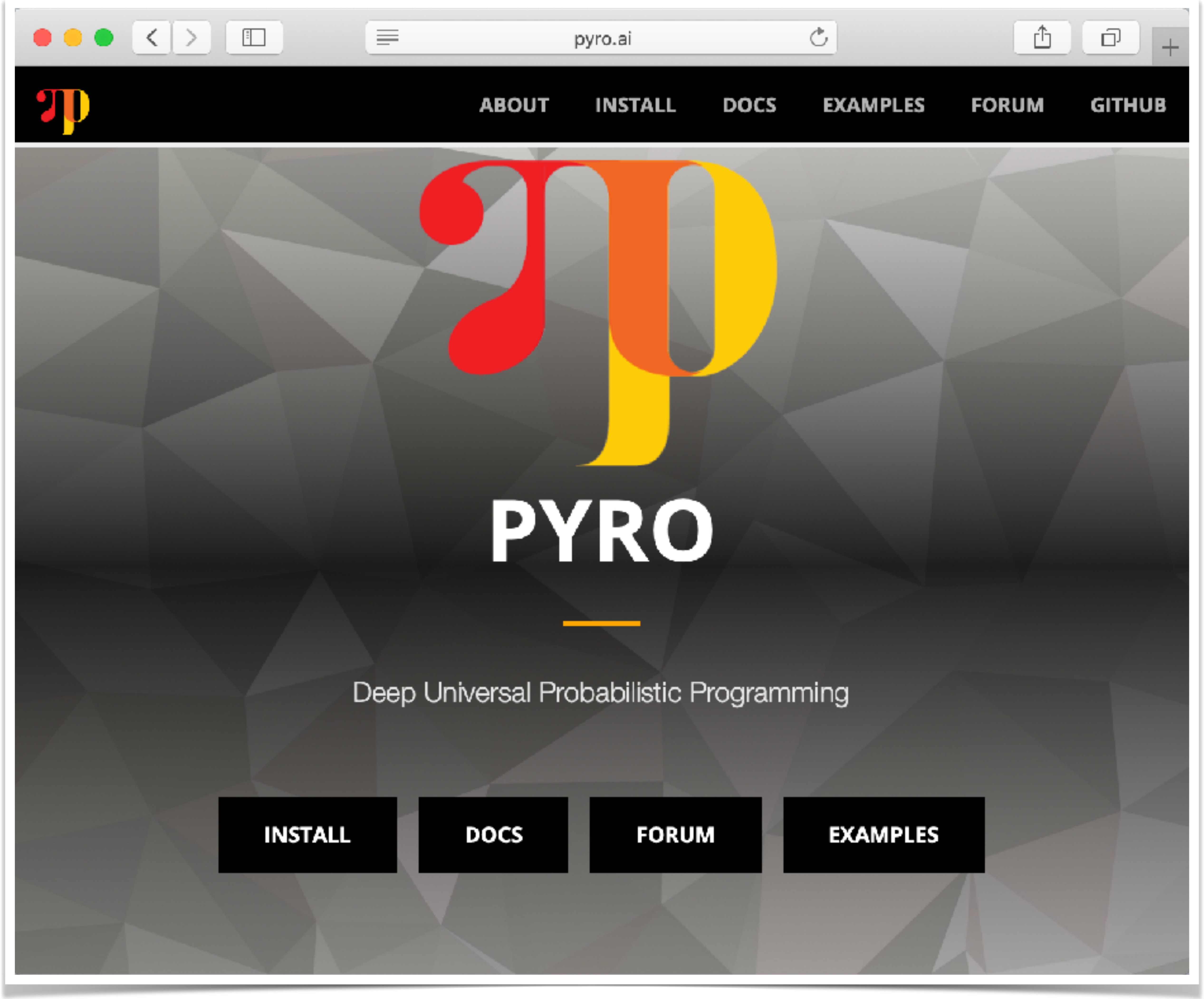
3	3	5	7	1	**9**	6	2	4
9	6	5	7	4	**2**	9	**3**	8
2	7	4	8	**5**	6	9	9	**1**
4	9	8	6	7	1	**5**	8	2
5	3	**7**	9	8	**4**	3	1	**6**
6	6	1	2	7	5	4	7	**9**
8	5	6	4	**2**	**7**	3	5	5
1	4	2	1	**9**	3	7	6	4
7	4	9	**5**	**6**	8	7	9	4

https://plai.cs.ubc.ca/2022/11/16/graphically-structured-diffusion-models/

Writing a good generative model is **hard**

	Yes	No
One-shot	Probabilistic Programming	?
Repeated	Amortized Inference	Un- and Semi-Supervised Deep Learning

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

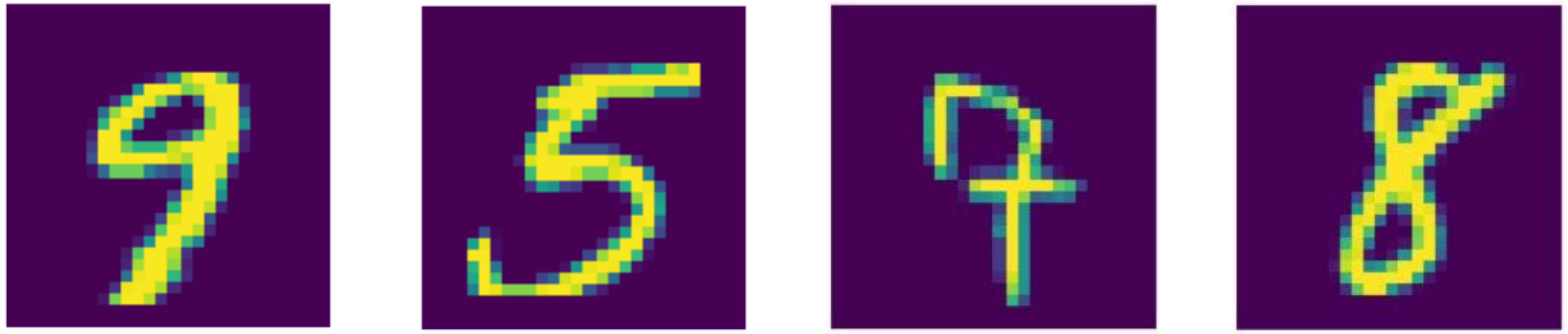
- **Idea:** define a generative model as a program, and a “inference model” as a second program

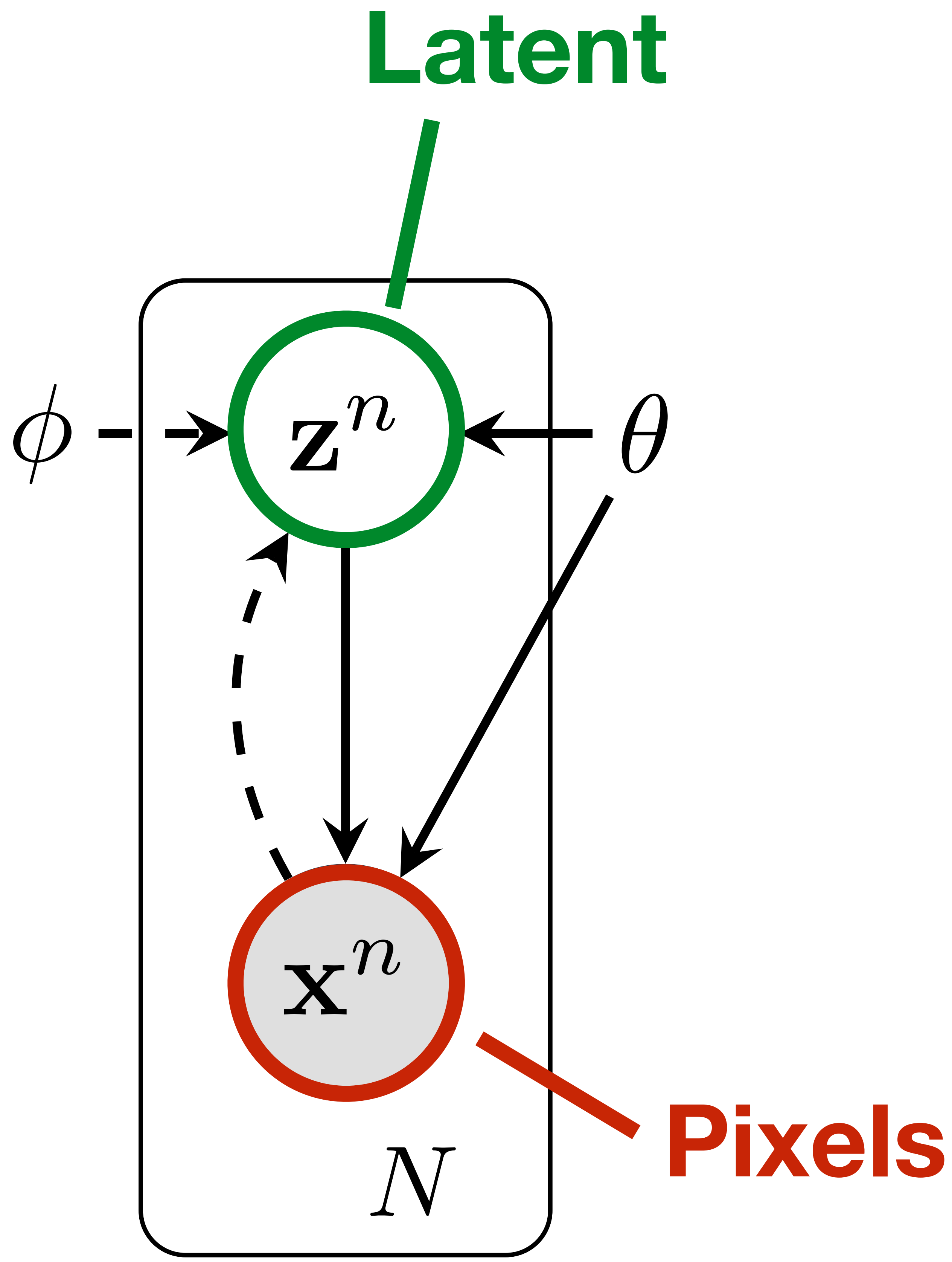
- Assign a “name” to every latent random variable, and make sure that they line up (be careful if support is unbounded…!)

- **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?

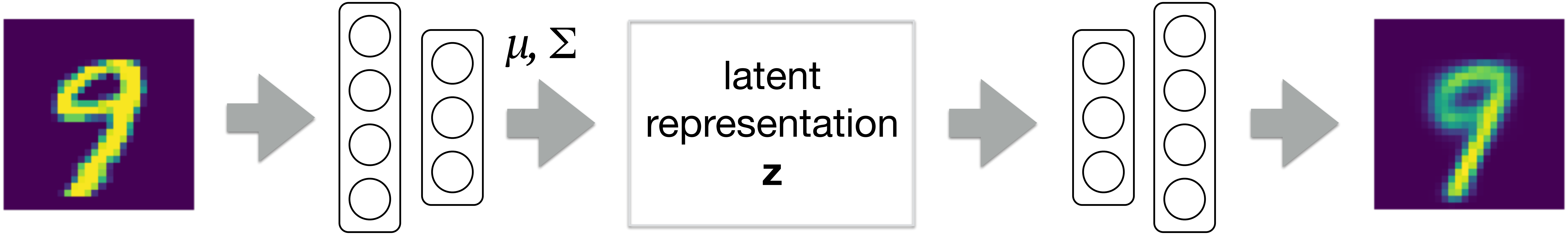




Learning deep generative models

Inference (encoder, guide)

Generative model (decoder)



$q_\phi(\mathbf{z}_n | \mathbf{x}_n)$

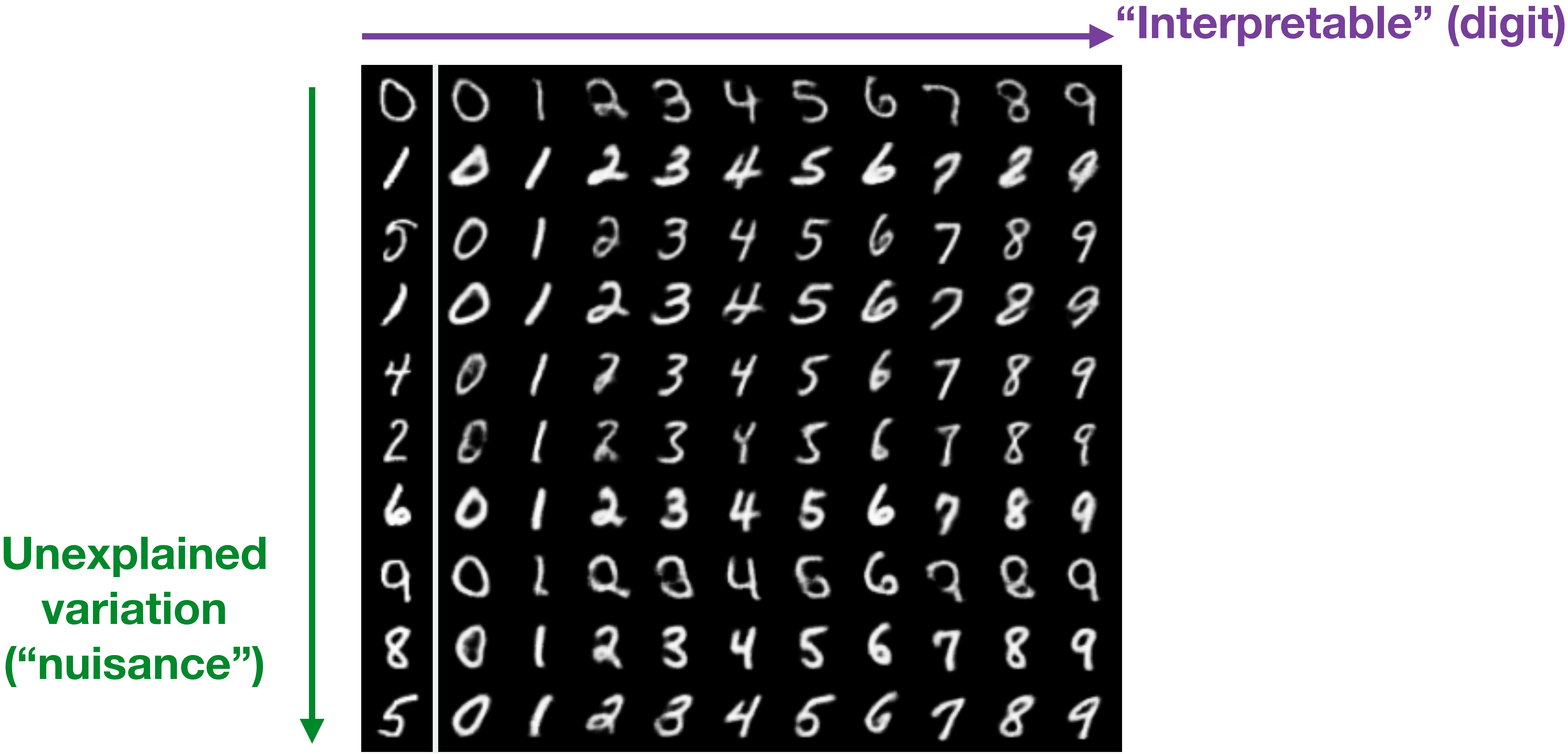
$p(\mathbf{z}_n)$

$p_\theta(\mathbf{x}_n | \mathbf{z}_n)$

Incomprehensible Latent Variable

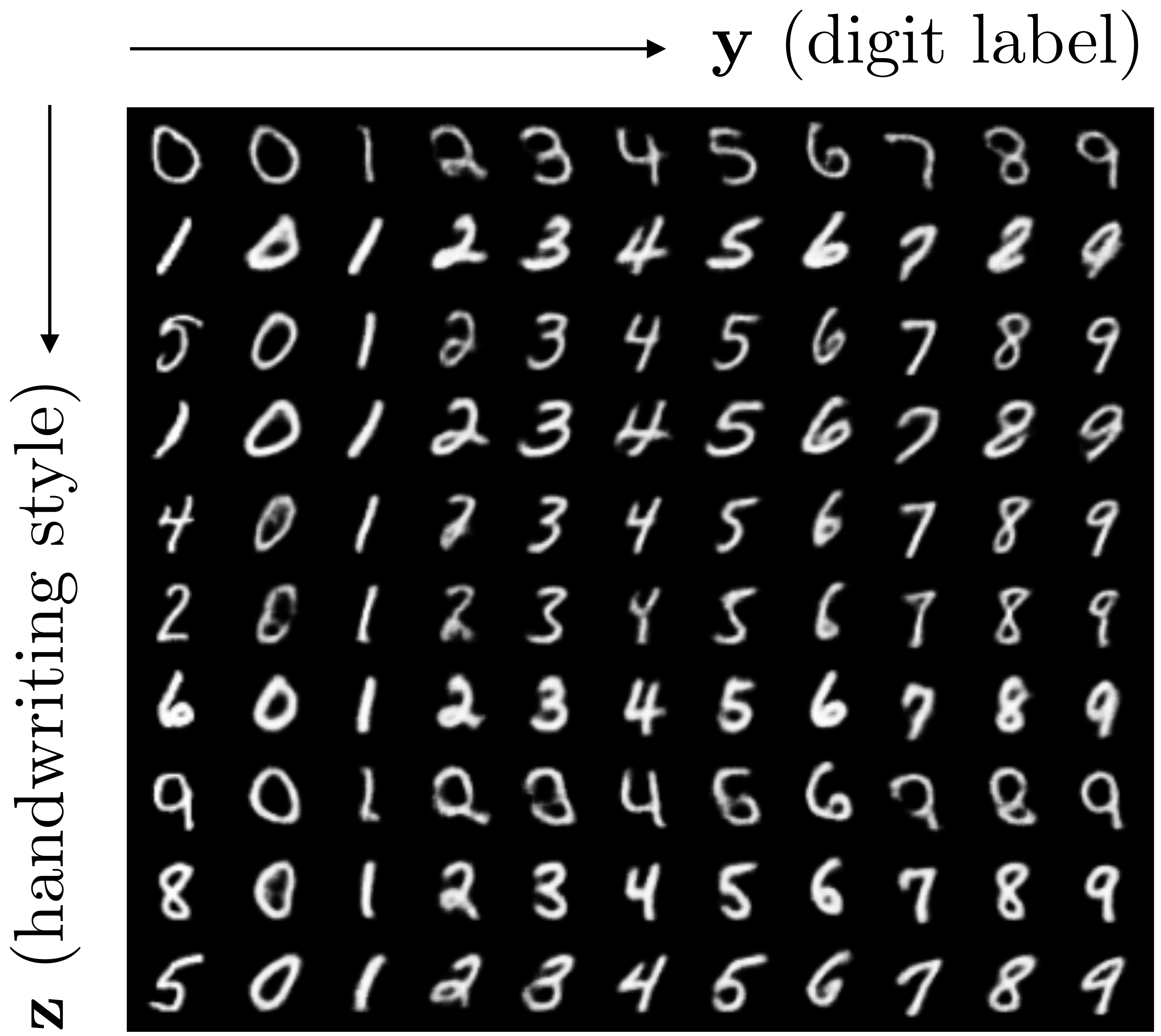
Kingma & Welling 2014; Rezende et al. 2014

Disentangled representations

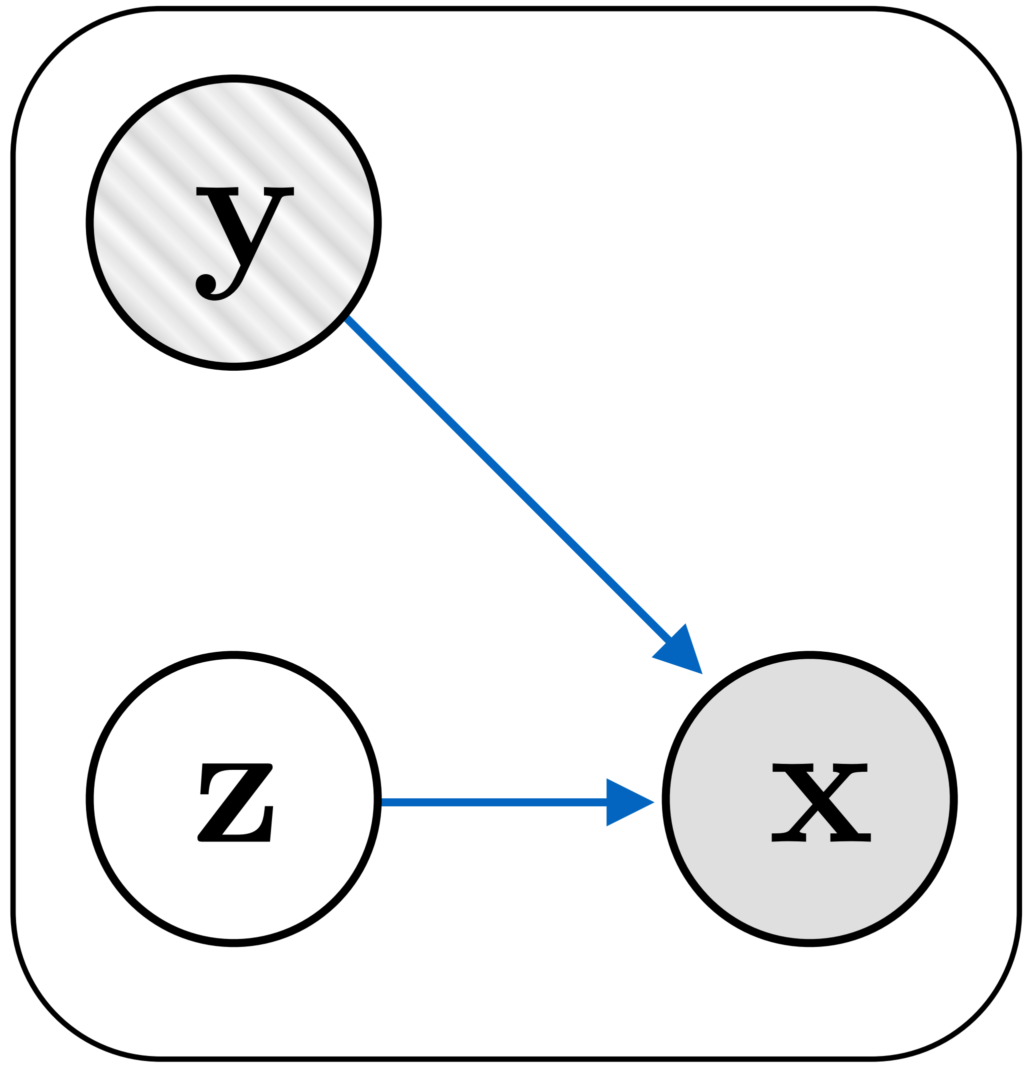


Disentangled representations

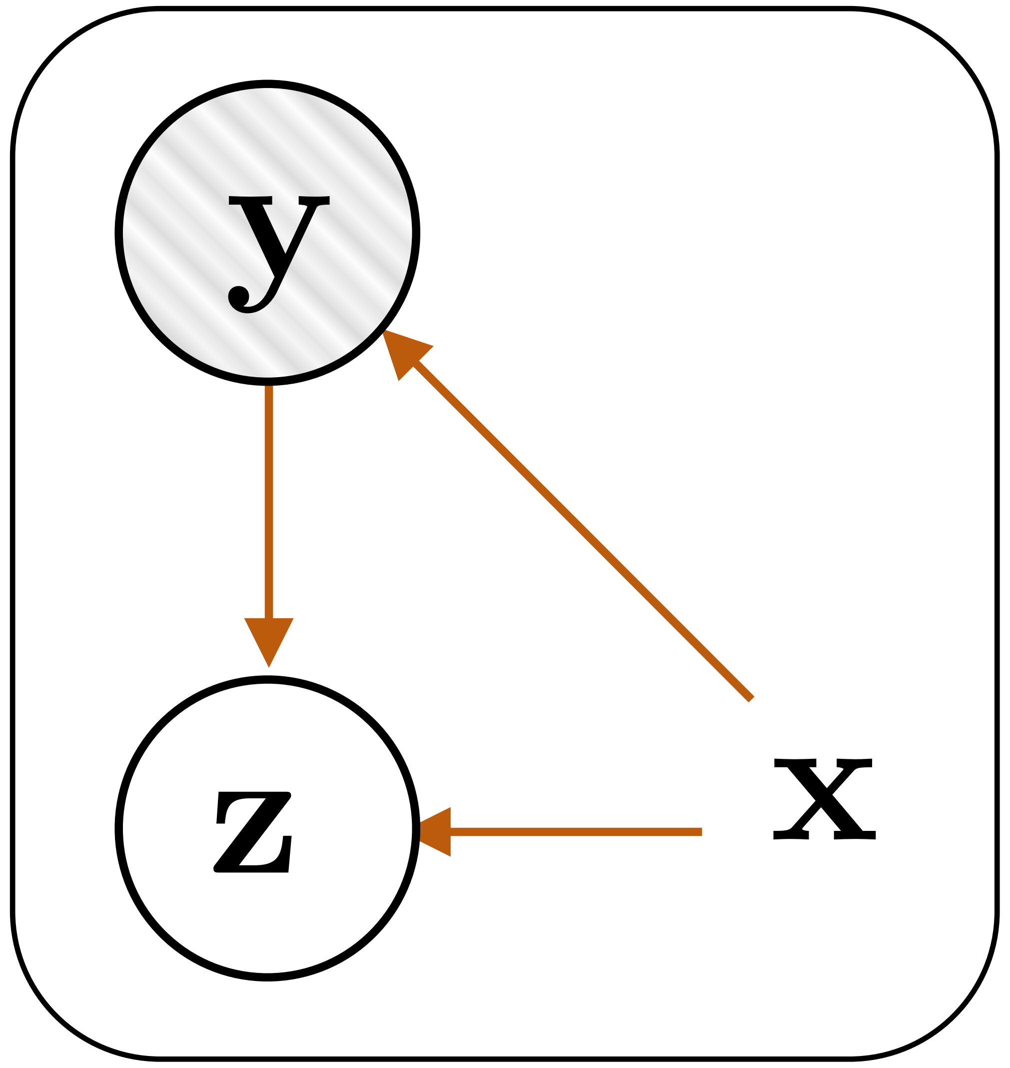
Separate interpretable **y** from nuisance variables **z**



Generative model: predict pixels **x** from **y** and **z**

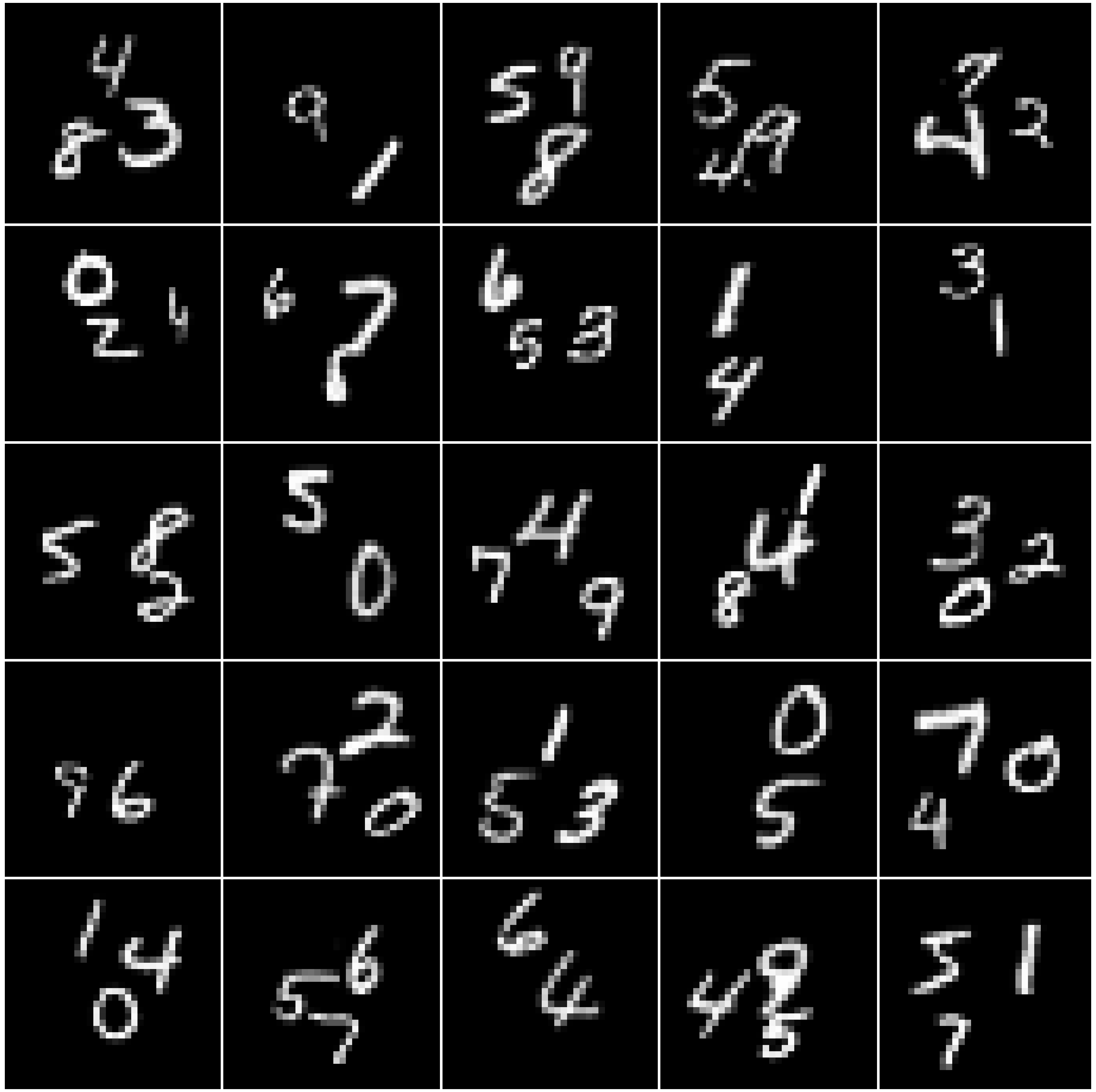


Inference: predict label **y** from pixels **x**, and then predict **z** from **x** and **y**

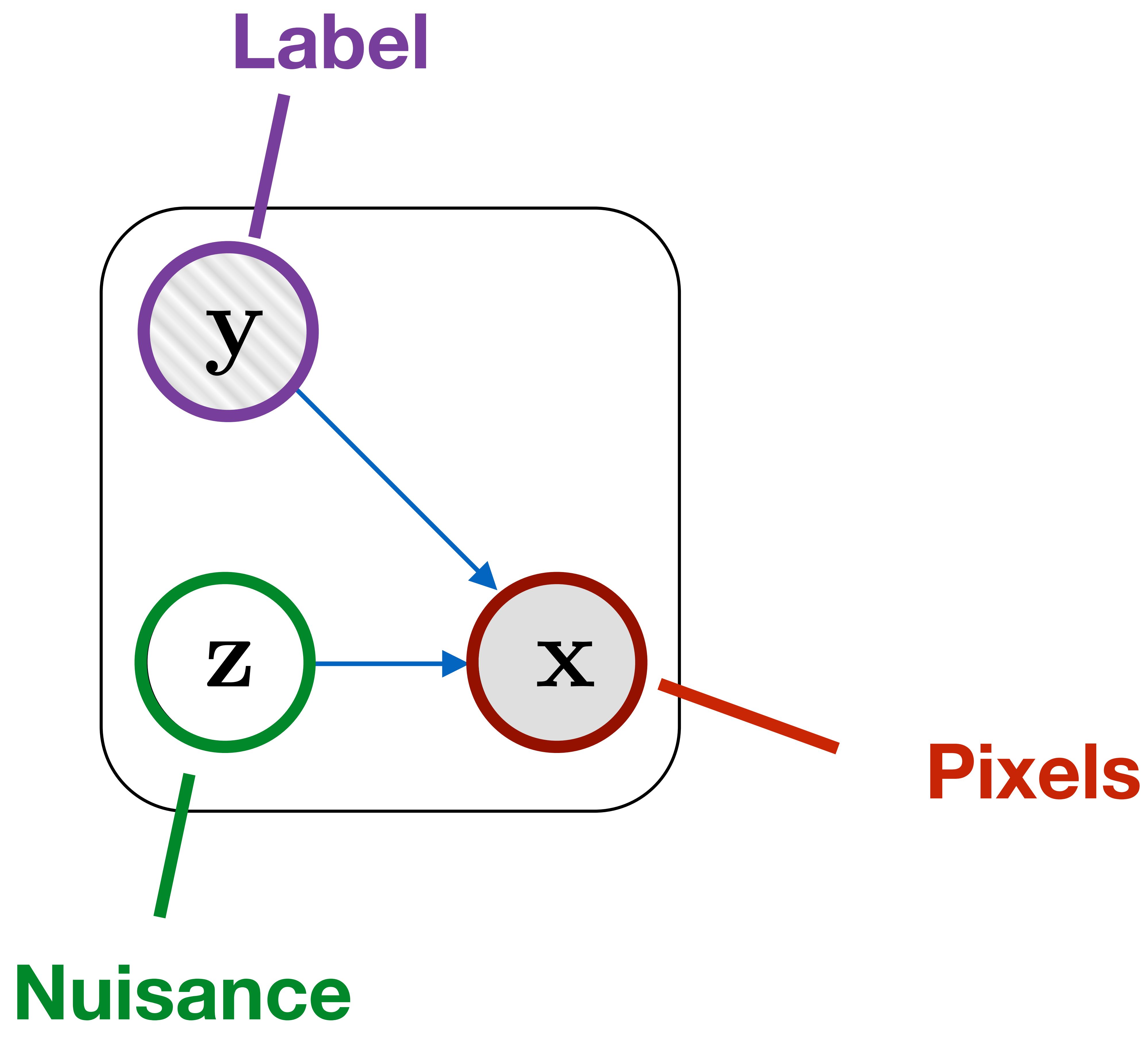


Kingma et al, *Semi-supervised learning with deep generative models*, NIPS 2014

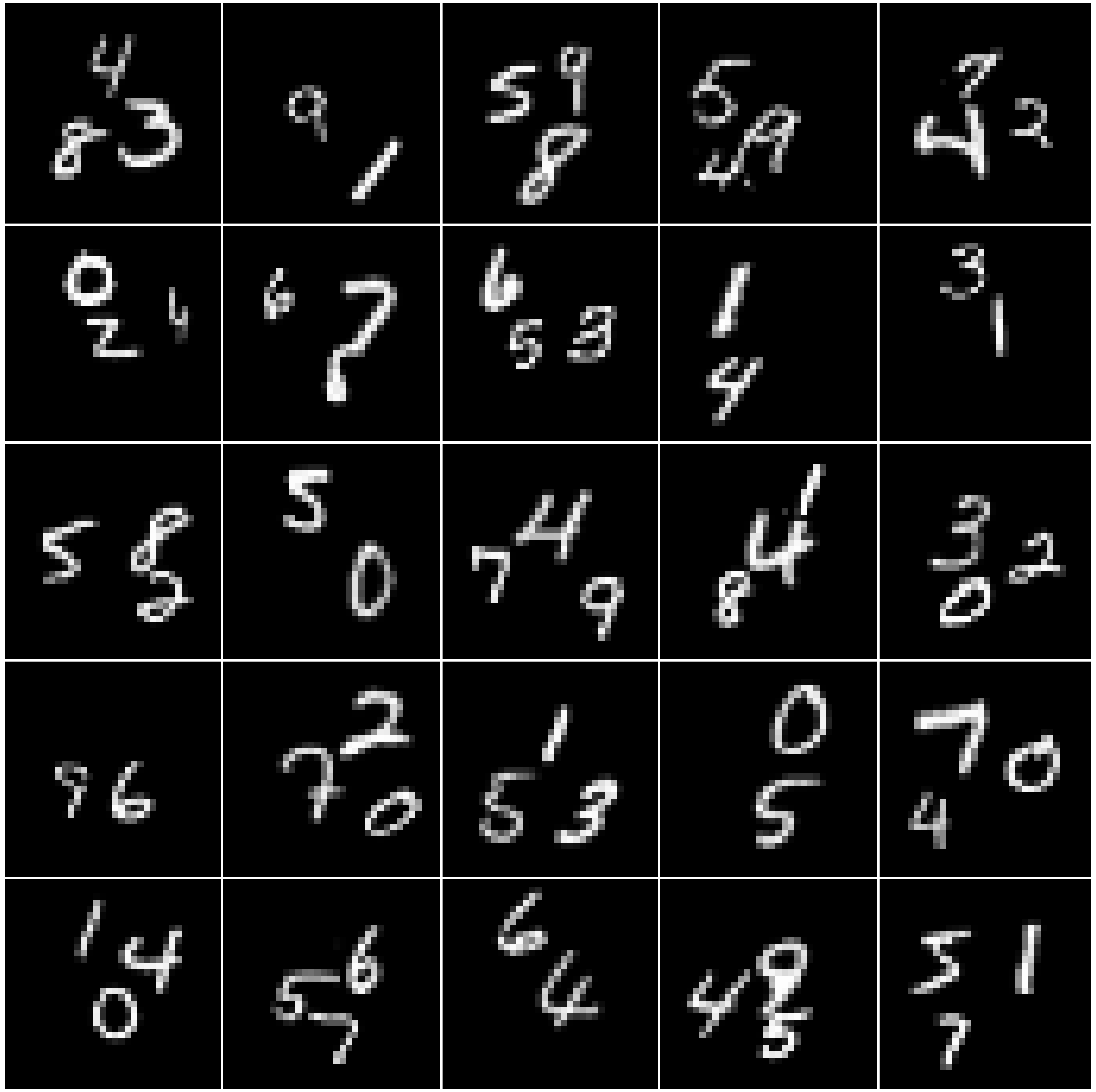
From one digit to many digits

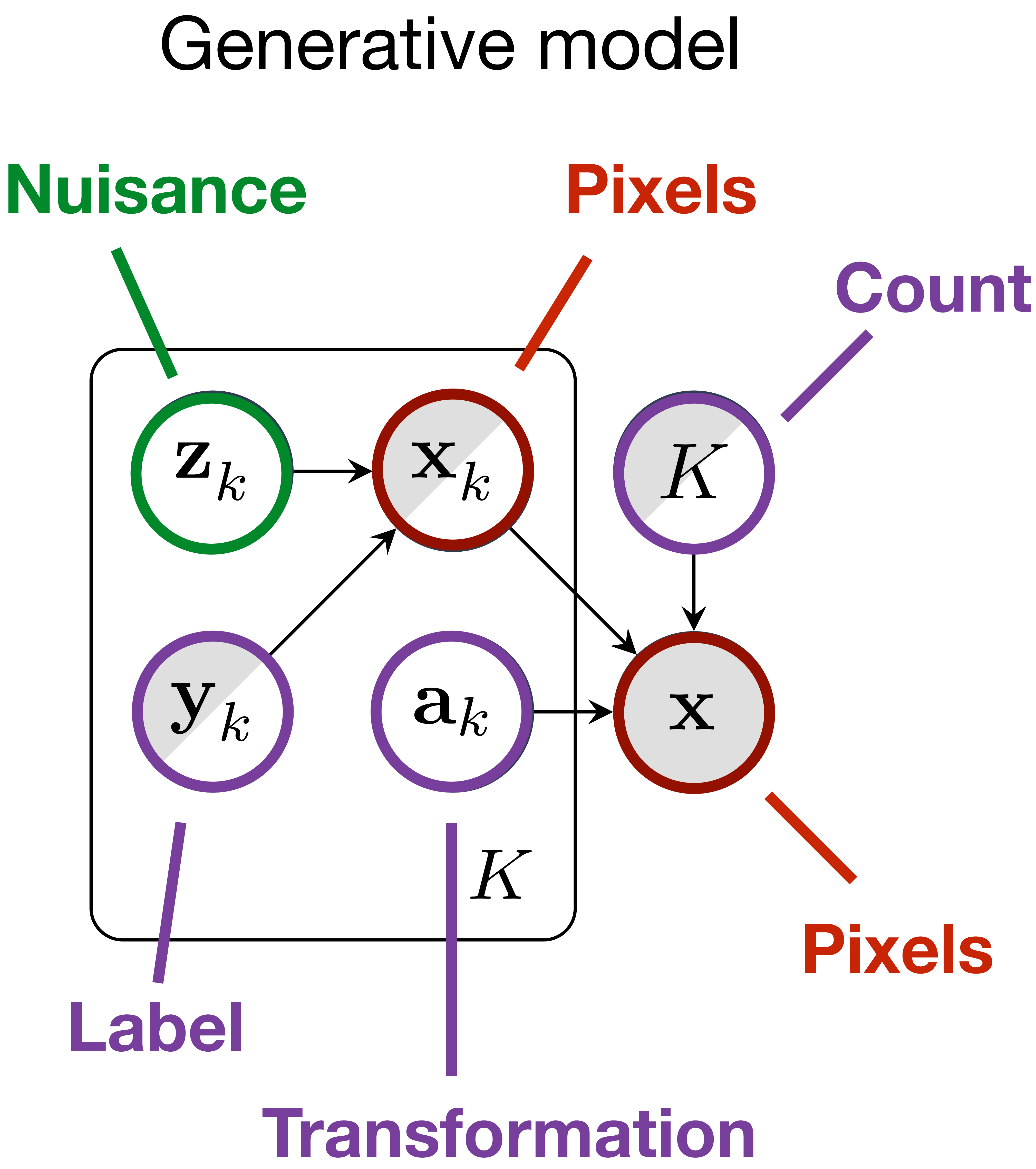


Generative model



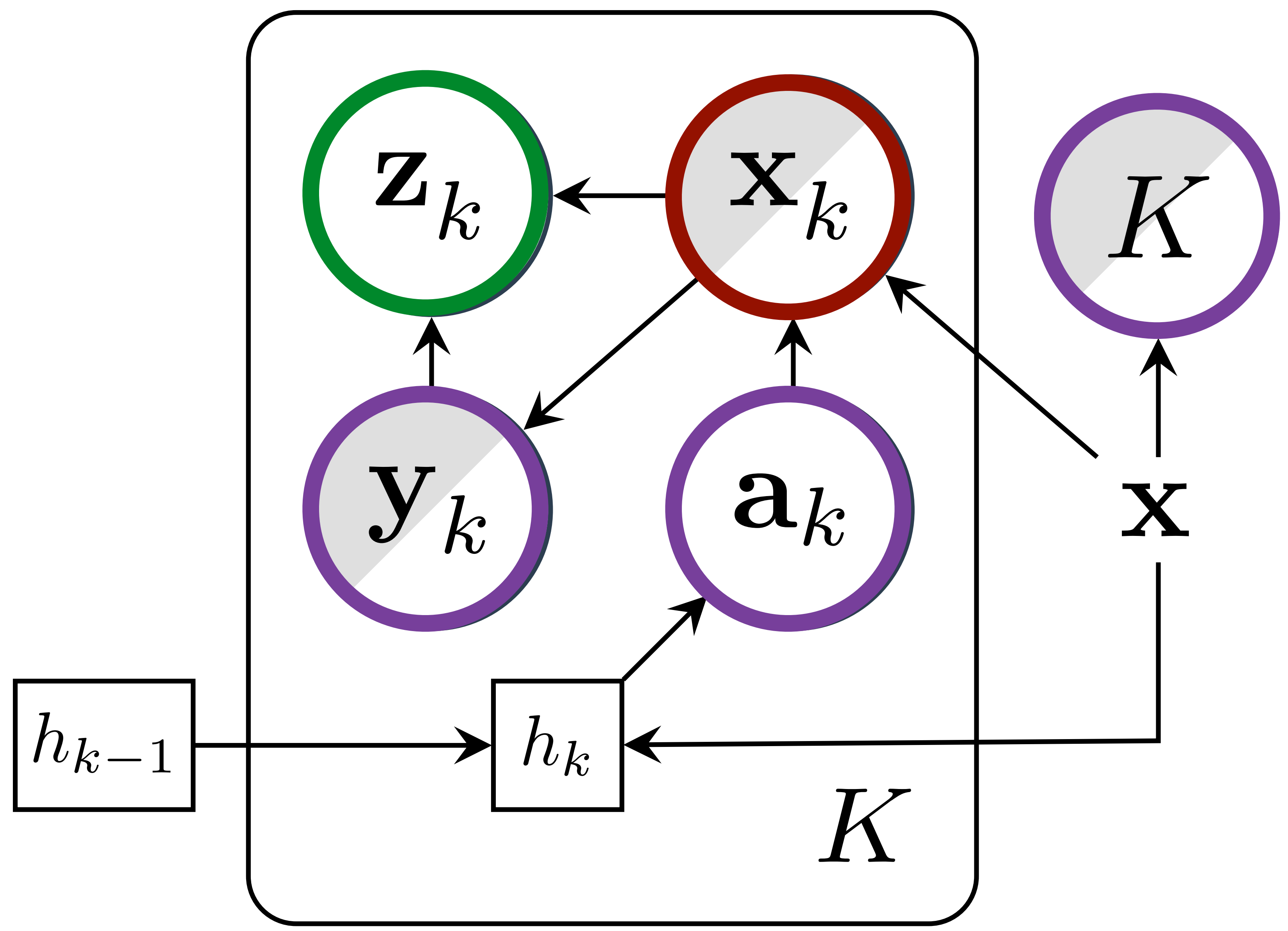
From one digit to many digits



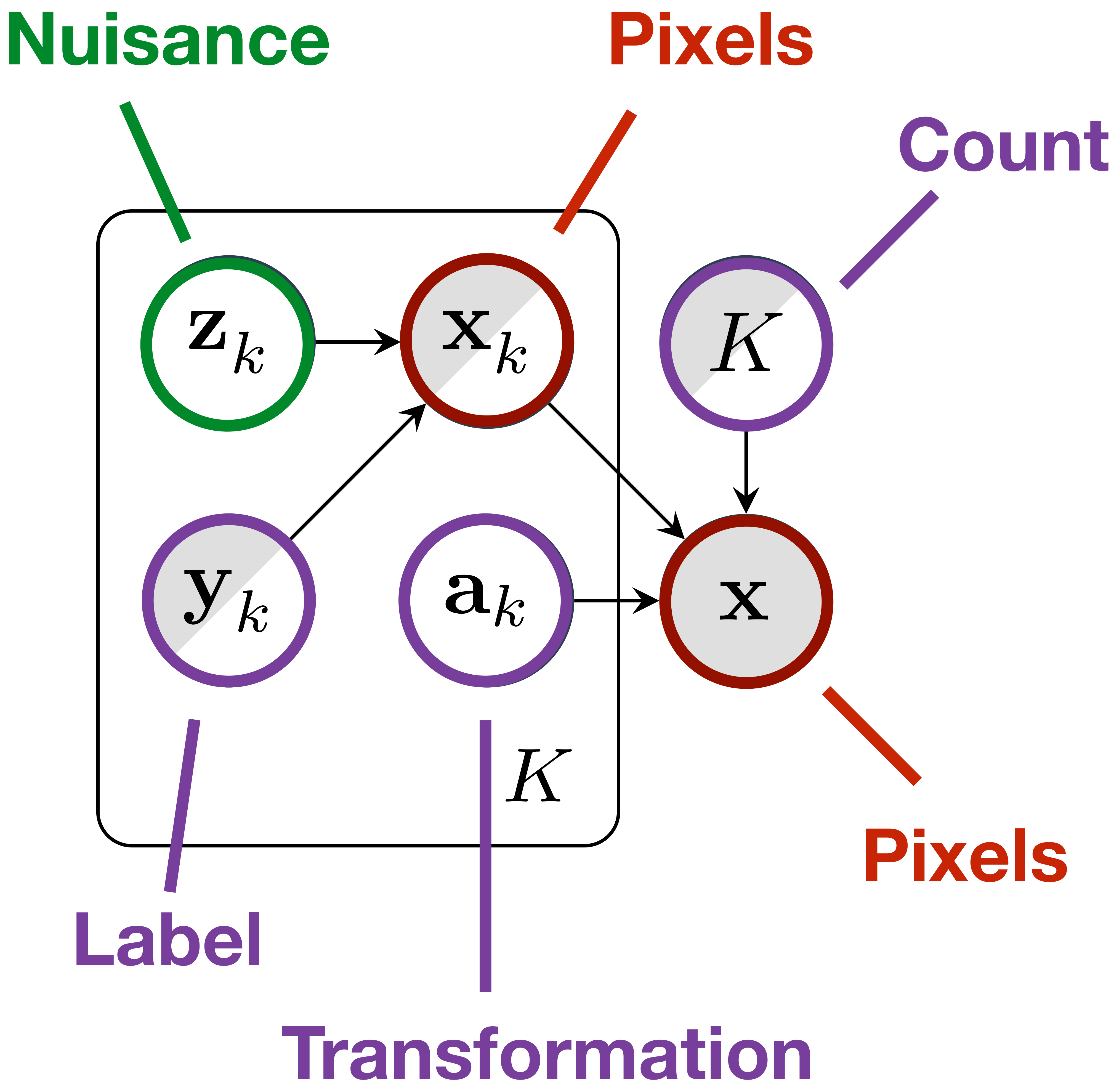


How do we build models?

Inference model
(recurrent neural network)

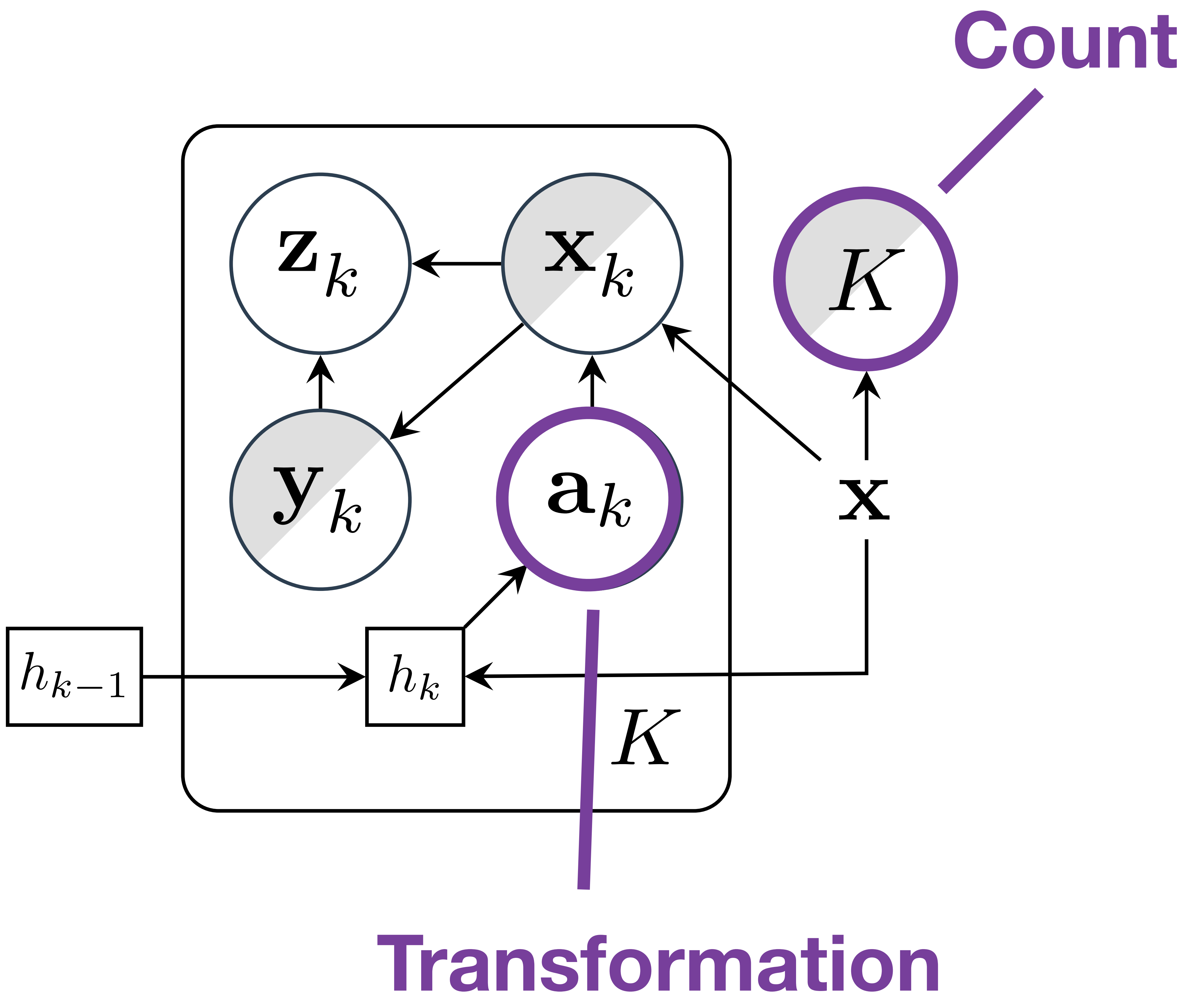


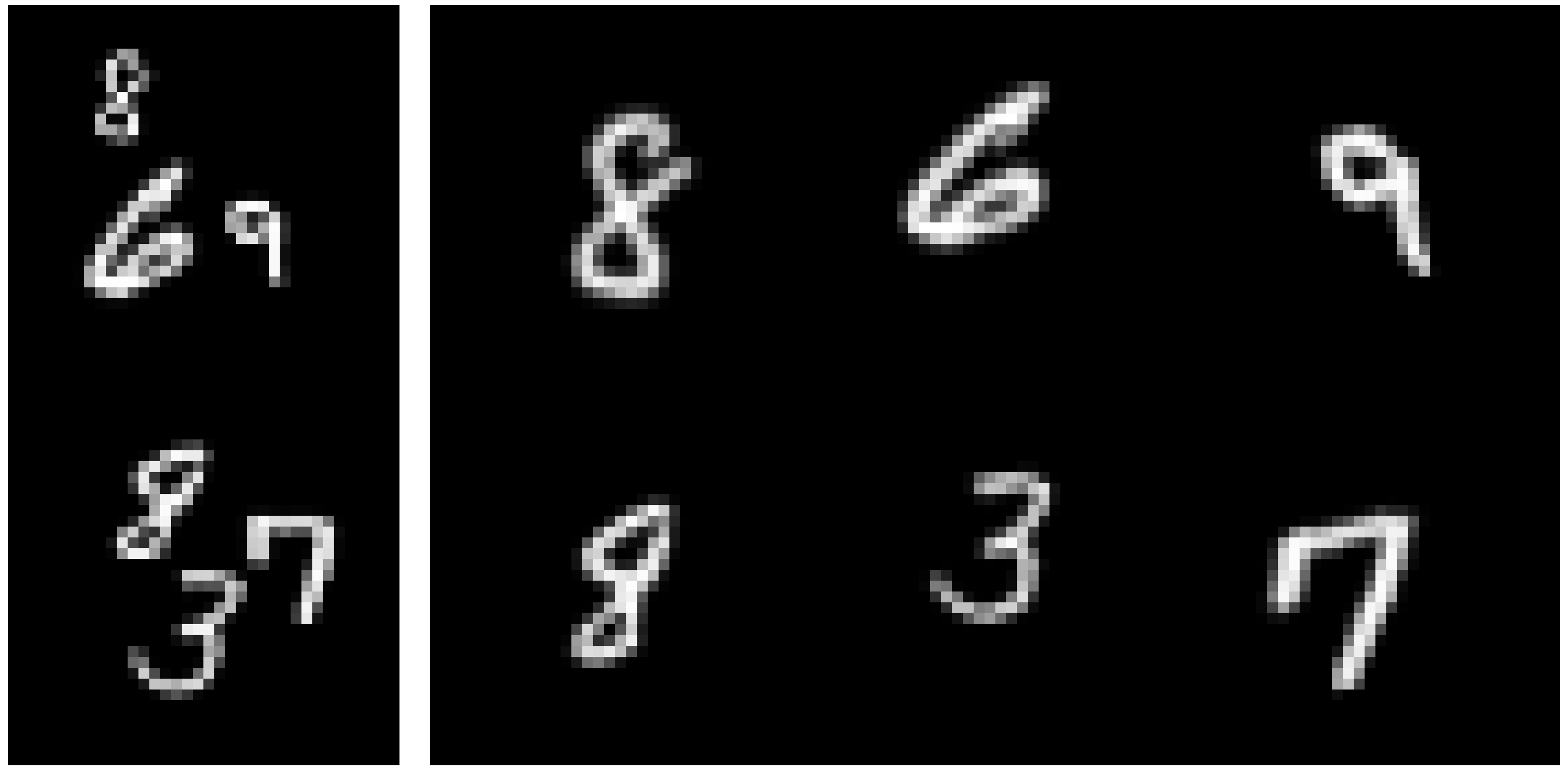
Generative model



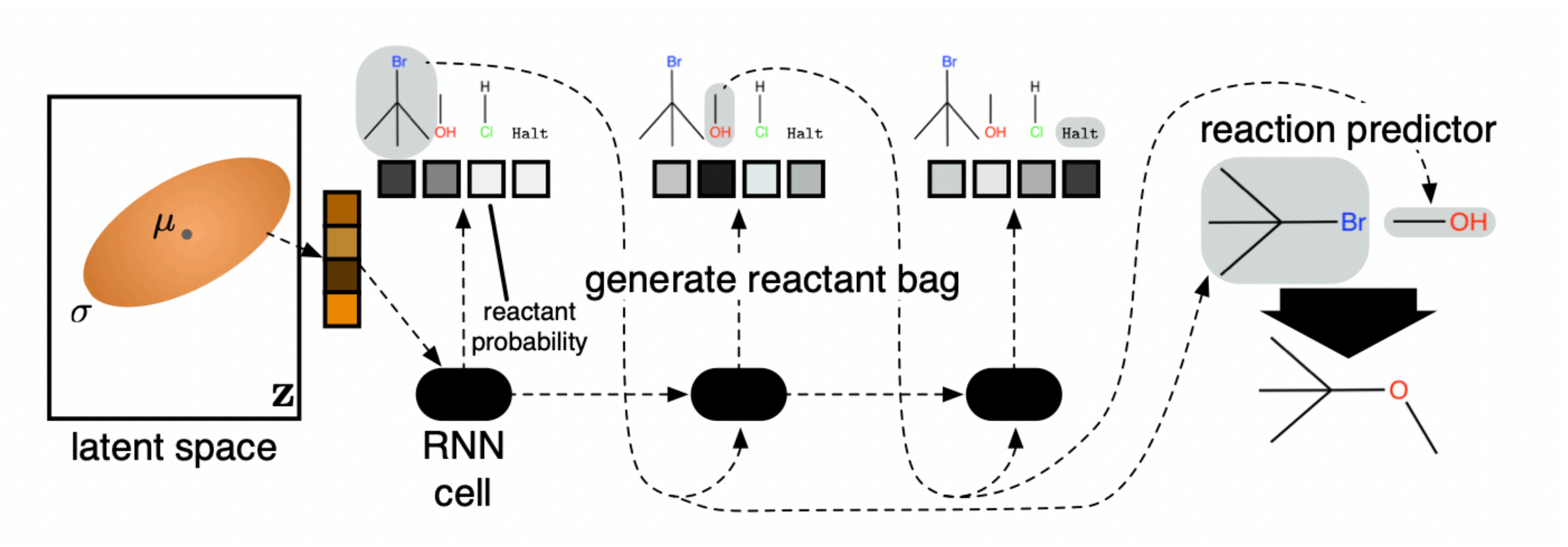
Inference: counting and locating

Inference model
(recurrent neural network)





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
- **An Introduction to Probabilistic Programming** https://arxiv.org/abs/1809.10756
- Frank Wood's graduate course: https://www.cs.ubc.ca/~fwood/CS532W-539W/

Thanks!