Probabilistic Programming

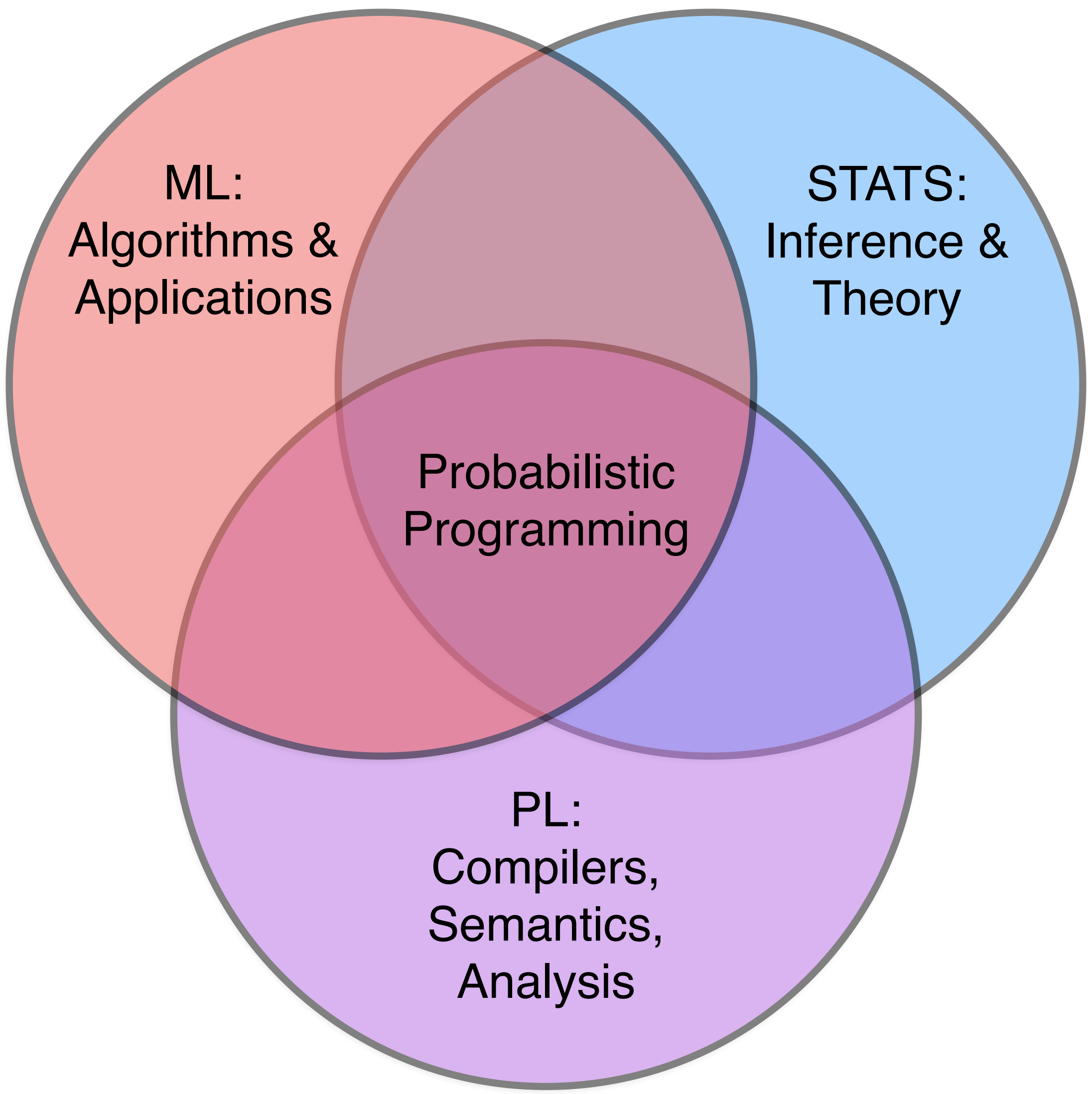
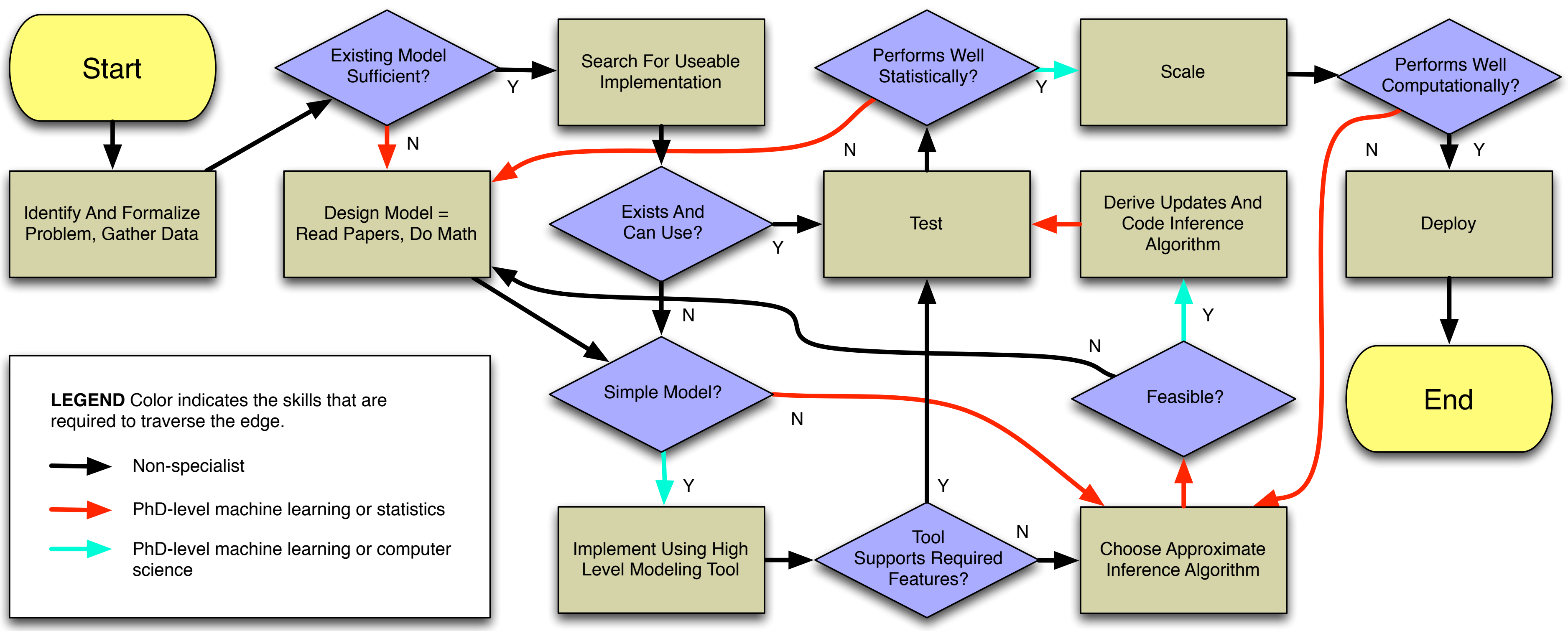


Figure credit: Frank Wood

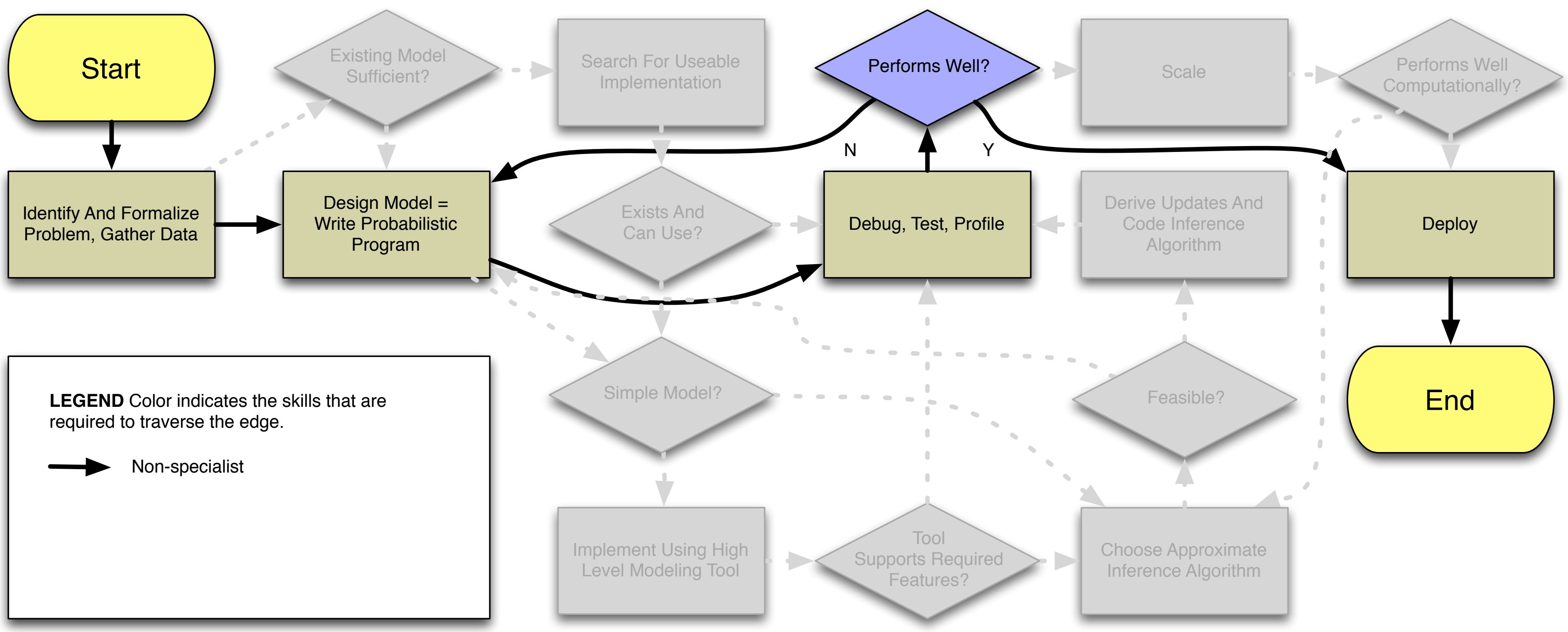
Why Probabilistic Programming?

Simplify Machine Learning…



Slide credits: Frank Wood

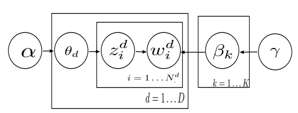
To This

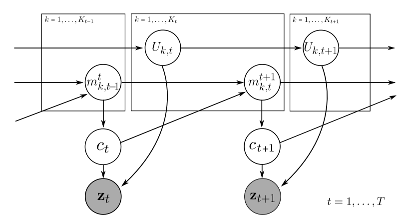


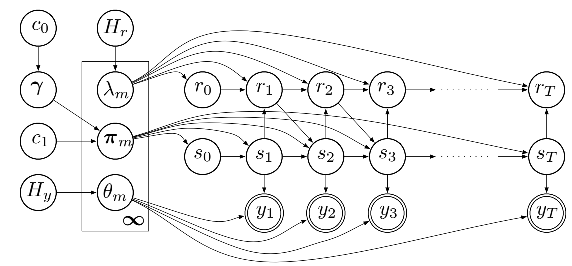
Slide credits: Frank Wood

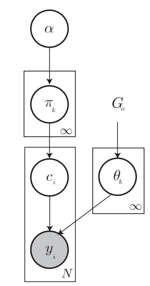
Automate Inference

Models / Stochastic Simulators

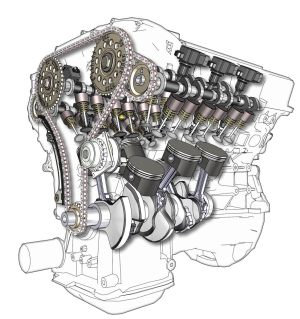








Programming Language Representation / Abstraction Layer



Inference Engine(s)

Slide credits: Frank Wood

What is Probabilistic Programming?

Operative Definition

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

(1) the ability to draw values at random from distributions, and

(2) the ability to condition values of variables in a program via observations."

Gordon et al, 2014

Slide credits: Frank Wood

Probabilistic Programs:
Defining Sampling Processes

```
//create a gaussian distribution:
var g = Gaussian({mu: 0, sigma: 1})

//sample from it:
print( sample(g) )

//can also use the sampling helper (note lower-case name):
print( gaussian(0,1) )

//and build more complex processes!
var foo = function(){return gaussian(0,1)*gaussian(0,1)}
foo()
```

run

Probabilistic Programs:
Defining Sampling Processes

```
//create a gaussian distribution:
var g = Gaussian({mu: 0, sigma: 1})

//sample from it:
print( sample(g) )

//can also use the sampling helper (note lower-case name):
print( gaussian(0,1) )

//and build more complex processes!
var foo = function(){return gaussian(0,1)*gaussian(0,1)}
foo()
```

run

(Distribution objects)

Probabilistic Programs: Defining Sampling Processes

```
//create a gaussian distribution:
var g = Gaussian({mu: 0, sigma: 1})

//sample from it:
print( sample(g) )

//can also use the sampling helper (note lower-case name):
print( gaussian(0,1) )

//and build more complex processes!
var foo = function(){return gaussian(0,1)*gaussian(0,1)}
foo()
```

run

(Distribution objects)

(Distributions support sample)

Probabilistic Programs: Defining Sampling Processes

```
//create a gaussian distribution:
var g = Gaussian({mu: 0, sigma: 1})

//sample from it:
print( sample(g) )

//can also use the sampling helper (note lower-case name):
print( gaussian(0,1) )

//and build more complex processes!
var foo = function(){return gaussian(0,1)*gaussian(0,1)}
foo()
```

run

(Distribution objects)

(Distributions support sample)

(Easy to build complex distributions)

Probabilistic Programs:
Defining Sampling Processes

```
//create a gaussian distribution:
var g = Gaussian({mu: 0, sigma: 1})

//sample from it:
print( sample(g) )

//can also use the sampling helper (note lower-case name):
print( gaussian(0,1) )

//and build more complex processes!
var foo = function(){return gaussian(0,1)*gaussian(0,1)}
foo()
```

run

```
0.4844819841420675

1.4341692553095442

0.08057731257784836
```

x

Probabilistic Programs:
Defining Sampling Processes

```
//create a gaussian distribution:
var g = Gaussian({mu: 0, sigma: 1})

//sample from it:
print( sample(g) )

//can also use the sampling helper (note lower-case name):
print( gaussian(0,1) )

//and build more complex processes!
var foo = function(){return gaussian(0,1)*gaussian(0,1)}
foo()
```

run

```
0.4844819841420675

1.4341692553095442

0.08057731257784836
```

The generative model is now defined by a sampling process

A sampling process implicitly defines a distribution over output values…

Another PPL construct makes this distribution explicit: Infer

Probabilistic Programs:
`Infer` Construct: Convert Implicit Distribution to Explicit Object

(Implicitly Defined Distribution)

```
//a complex function, that specifies a complex sampling process:
var foo = function(){gaussian(0,1)*gaussian(0,1)}

//make the marginal distributions on return values explicit:
var d = Infer({method: 'forward', samples: 10000}, foo)

//now we can use d as we would any other distribution:
print( sample(d) )
viz(d)
```

run

Probabilistic Programs:
`Infer` Construct: Convert Implicit Distribution to Explicit Object

```
//a complex function, that specifies a complex sampling process:
var foo = function(){gaussian(0,1)*gaussian(0,1)}

//make the marginal distributions on return values explicit:
var d = Infer({method: 'forward', samples: 10000}, foo)

//now we can use d as we would any other distribution:
print( sample(d) )
viz(d)
```

(Implicitly Defined Distribution)

(Infer by Forward Sampling)

run

Probabilistic Programs:
`Infer` Construct: Convert Implicit Distribution to Explicit Object

```
//a complex function, that specifies a complex sampling process:
var foo = function(){gaussian(0,1)*gaussian(0,1)}

//make the marginal distributions on return values explicit:
var d = Infer({method: 'forward', samples: 10000}, foo)

//now we can use d as we would any other distribution:
print( sample(d) )
viz(d)
```

run

(Implicitly Defined Distribution)

(Infer by Forward Sampling)

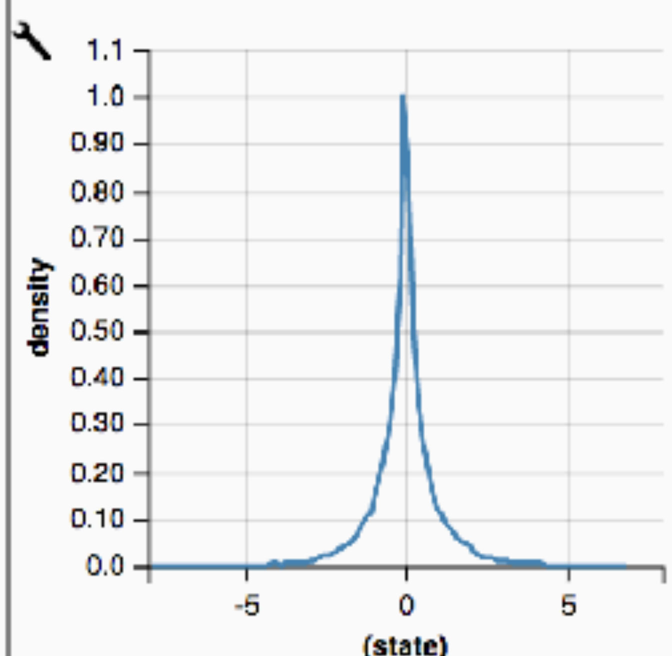
(Now Use like Distribution Object)

Probabilistic Programs:
`Infer` Construct: Convert Implicit Distribution to Explicit Object

```
//a complex function, that specifies a complex sampling process:
var foo = function(){gaussian(0,1)*gaussian(0,1)}

//make the marginal distributions on return values explicit:
var d = Infer({method: 'forward', samples: 10000}, foo)

//now we can use d as we would any other distribution:
print( sample(d) )
viz(d)
```

run

```
-0.11640391203046613
```



Need one more language feature: “mem”
`Random but persistent`: random on first call, cached for subsequent calls
Why needed:

```
var eyeColor = function (person) {
    return uniformDraw(['blue', 'green', 'brown']);
};
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
```

run

Call once

Need one more language feature: “mem”

`Random but persistent`: random on first call, cached for subsequent calls

Why needed:

```
var eyeColor = function (person) {
    return uniformDraw(['blue', 'green', 'brown']);
};
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
```

run

Call once

Call twice

Need one more language feature: “mem”
`Random but persistent`: random on first call, cached for subsequent calls
Why needed:

```
var eyeColor = function (person) {
    return uniformDraw(['blue', 'green', 'brown']);
};
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
```

Call once
Call twice

run

```
["blue","brown","brown"]

["green","green","blue"]
```

Bob’s eye color shouldn’t change…

Need one more language feature: `mem`
`Random but persistent`: random on first call, cached for subsequent calls
Why needed:

```
var eyeColor = mem(function (person) {
    return uniformDraw(['blue', 'green', 'brown']);
});
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
```

Call once
Call twice

run

```
["blue","green","blue"]

["blue","green","blue"]
```

Fixed: value is memoized after first run

Aside:

Dirichlet Process as Probabilistic Program

Recall: Dirichlet as Stick-Breaking Process

$$\{\beta'_k\}_{k=1}^{\infty},\ \beta'_k \sim \mathrm{Beta}(1, \alpha)$$

$$Pr\{k\} = \beta_k = \prod_{i=1}^{k-1}(1 - \beta'_i) \cdot \beta'_k$$

As generative model:

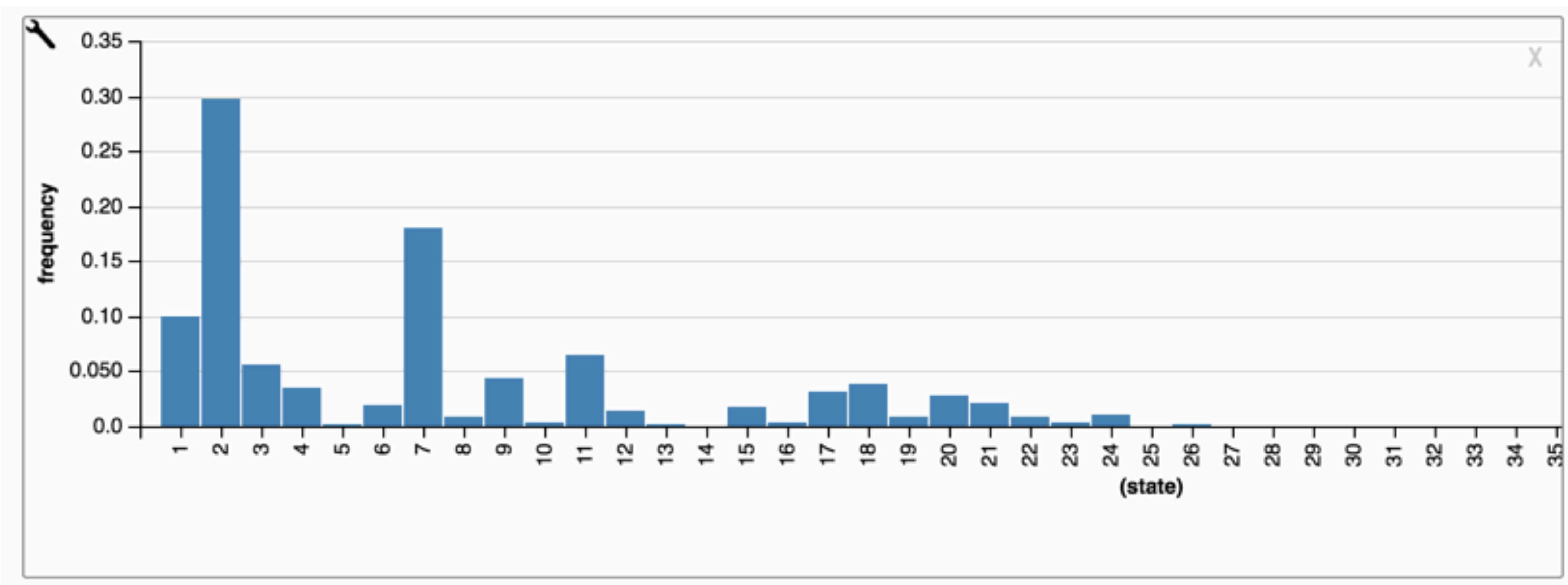
- **Walk down the natural numbers**
- **Flip a biased coin at each number :** $\mathrm{Ber}(\beta'_i)$
- **If FALSE, continue to next number. If TRUE, return the number**

As probabilistic program

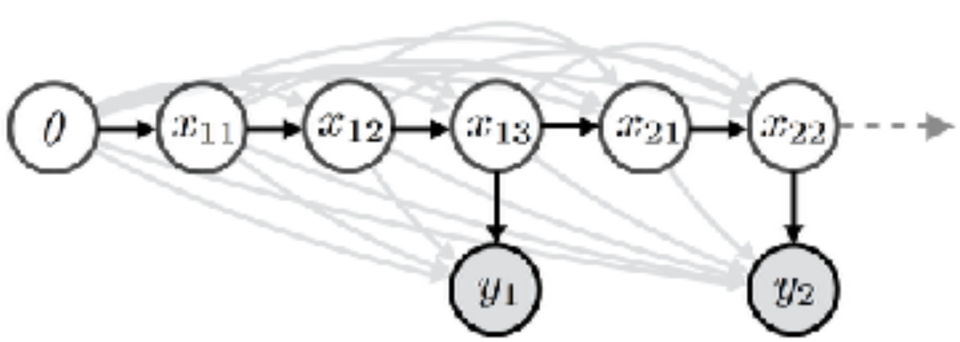
```
var pickStick = function(sticks, J) {
  return flip(sticks(J)) ? J : pickStick(sticks, J+1);
};

var makeSticks = function(alpha) {
  var sticks = mem(function(index) {return beta(1, alpha)});
  return function() {
    return pickStick(sticks,1)
  };
}
var mySticks = makeSticks(1);

viz(repeat(1000, mySticks))
```

As probabilistic program

```
var pickStick = function(sticks, J) {
  return flip(sticks(J)) ? J : pickStick(sticks, J+1);
};

var makeSticks = function(alpha) {
  var sticks = mem(function(index) {return beta(1, alpha)});
  return function() {
    return pickStick(sticks,1)
  };
}
var mySticks = makeSticks(1);

viz(repeat(1000, mySticks))
```



Universal Inference for Probabilistic Programming Languages




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

So far…

- Build complicated probabilistic models with PPLs
- Using **sample** statements: Specify prior generative proc.
- Using **factor** statements: Specify data likelihood
- A prob. program represents posterior over possible execution "traces"

How to develop generic inference algorithms?

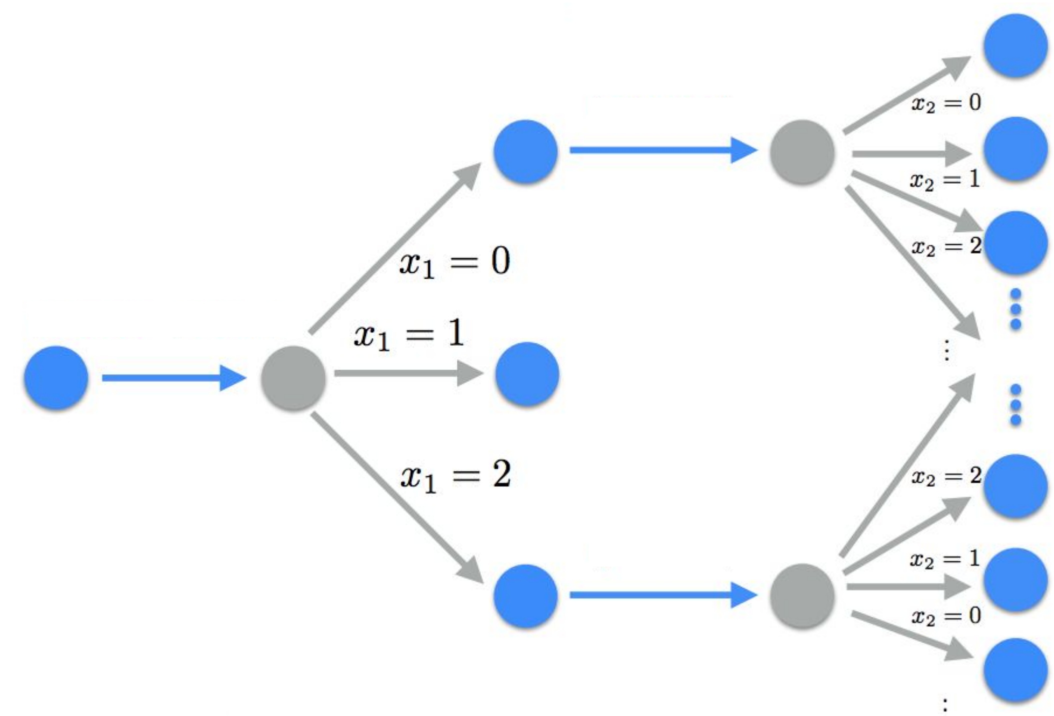
What is a "*Trace*"?

- Sequence of *M* **`sample`** `statements`

$$\{f_j, \theta_j\}_{j=1}^{M}$$

- Sequence of *M* sampled values

$$\{x_j\}_{j=1}^{M}$$



- Sequence of *N* **`factor`** `statements`

$$\{g_i, \phi_i, y_i\}_{i=1}^{N}$$

Inference over traces

- Trace probability:

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

- Posterior over traces:

$$\pi(\mathbf{x}) \triangleq p(\mathbf{x}|\mathbf{y}) = \frac{\gamma(\mathbf{x})}{Z}$$

$$Z = p(\mathbf{y}) = \int \gamma(\mathbf{x}) d\mathbf{x}$$

- What we care about:

$$\mathbb{E}_{\pi(\mathbf{x})}\left[f(\mathbf{x})\right]$$

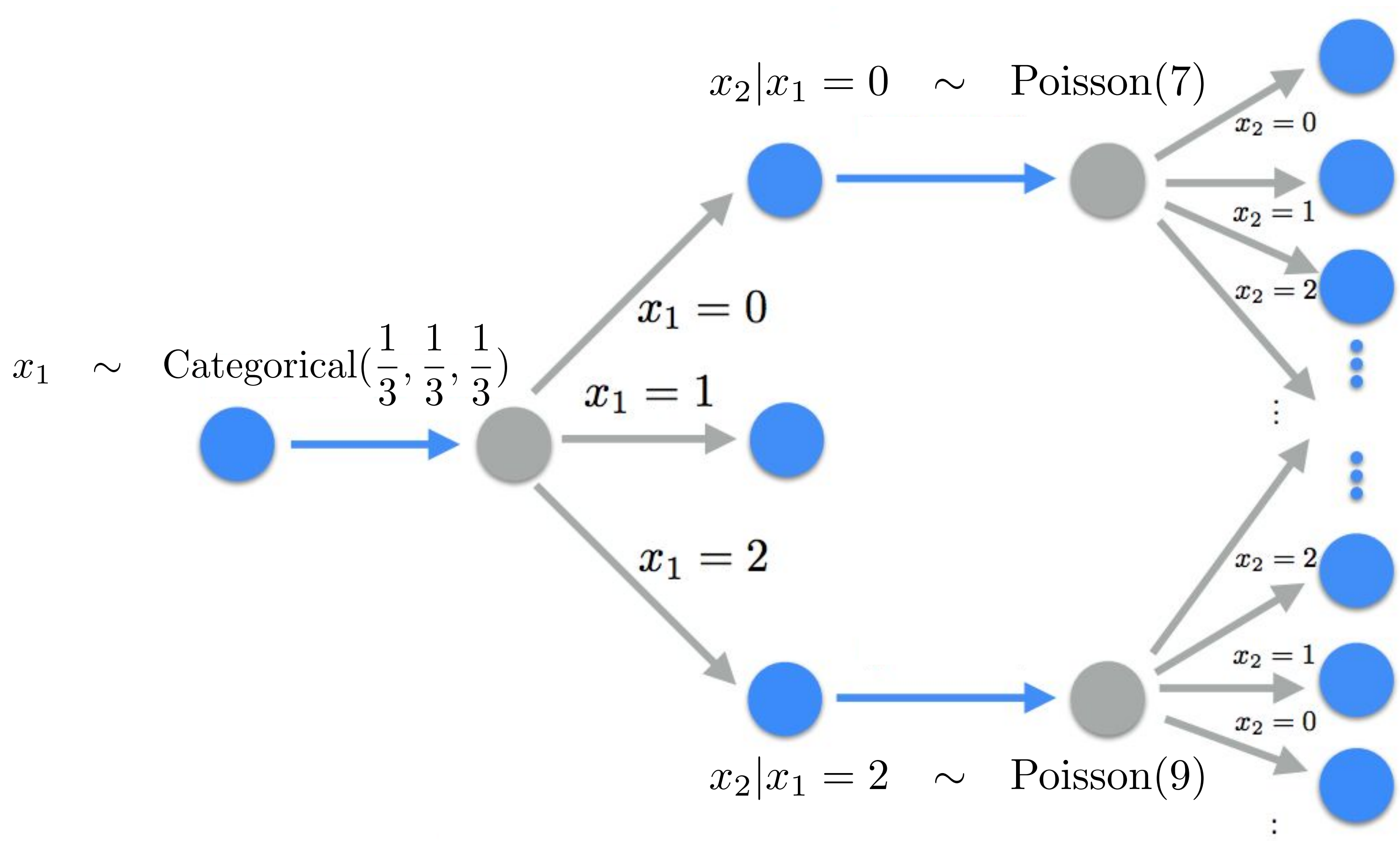
Sampling based inference over “traces”

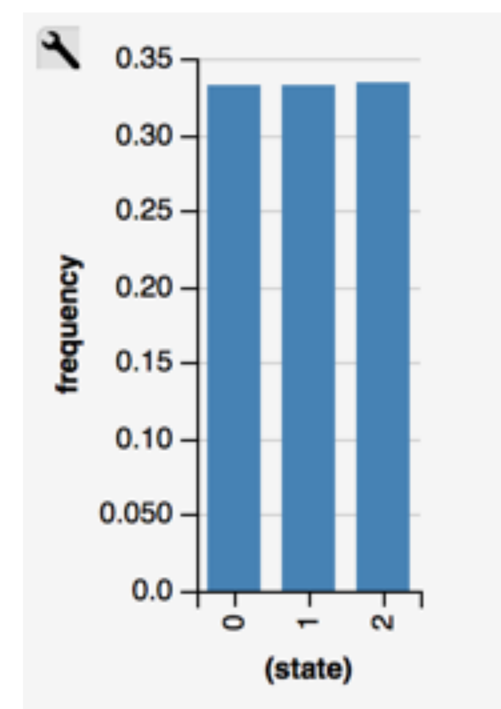
Inference over trace space: exploration–exploitation task

- **Explore** possible execution paths
    - As a side-effect, compute “goodness” of a trace
- **Exploit** good (more probable) traces
- Return projection of the posterior over traces

Inference over execution traces

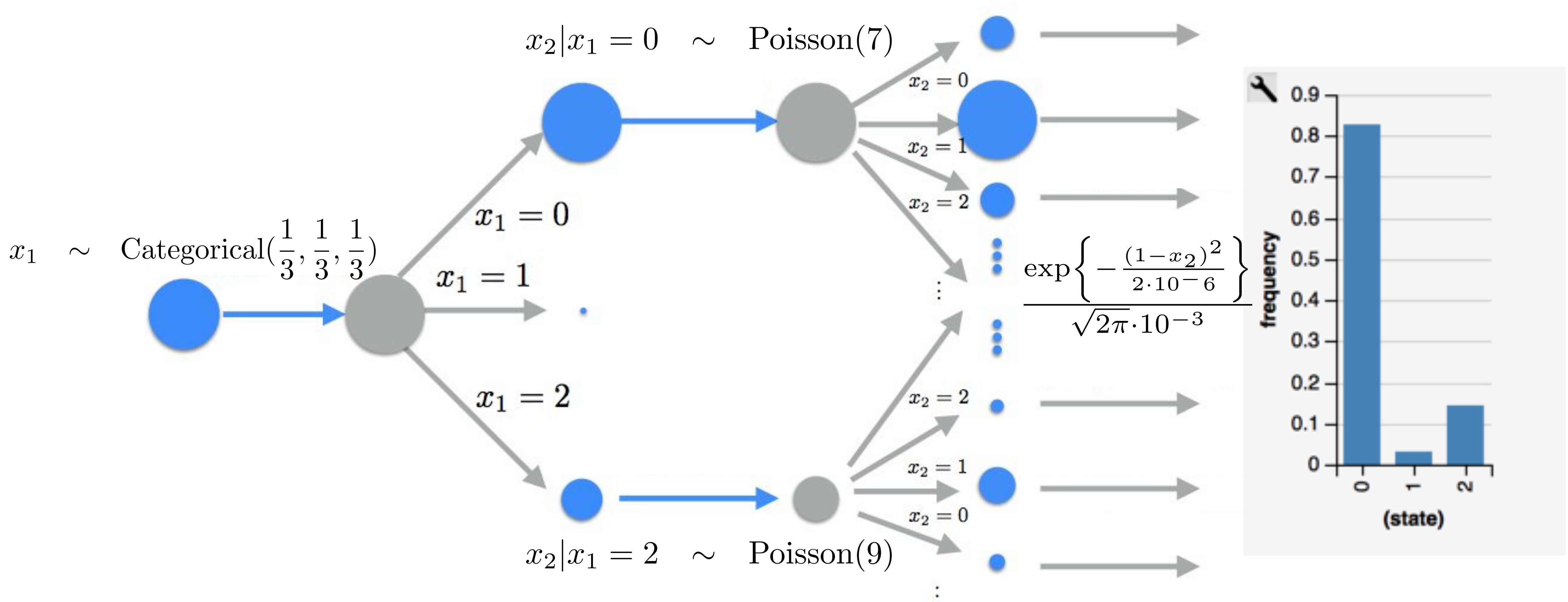
```
var my_model = function(){
  var x_1 = sample(Categorical({vs: [0, 1, 2], ps: [.33, .33, .33]}));
  if (x_1 !== 1){
    var x_2 = poisson(x_1 + 7);
//      factor(Gaussian({mu:x_2,sigma:0.0001}).score(1)) // y ~ N(x_1,0.0001) ; obs y =
  }
  return x_1;
}
var dist = Infer({method: 'MCMC', samples:1000000, burn: 10000}, my_model)
viz(dist)
```





Inference over execution traces

```
var my_model = function(){
  var x_1 = sample(Categorical({vs: [0, 1, 2], ps: [.33, .33, .33]}));
  if (x_1 !== 1){
    var x_2 = poisson(x_1 + 7);
    factor(Gaussian({mu:x_2,sigma:0.0001}).score(1)) // y ~ N(x_2,0.0001) ; obs y = 1
  }
  return x_1;
}
var dist = Infer({method: 'MCMC', samples:1000000, burn: 10000}, my_model)
viz(dist)
```



Importance sampling

- Run ***K*** independent copies of the program simulating from the prior

$$q(\mathbf{x}^k) = \prod_{j=1}^{M^k} f_j(x_j^k|\theta_j^k)$$

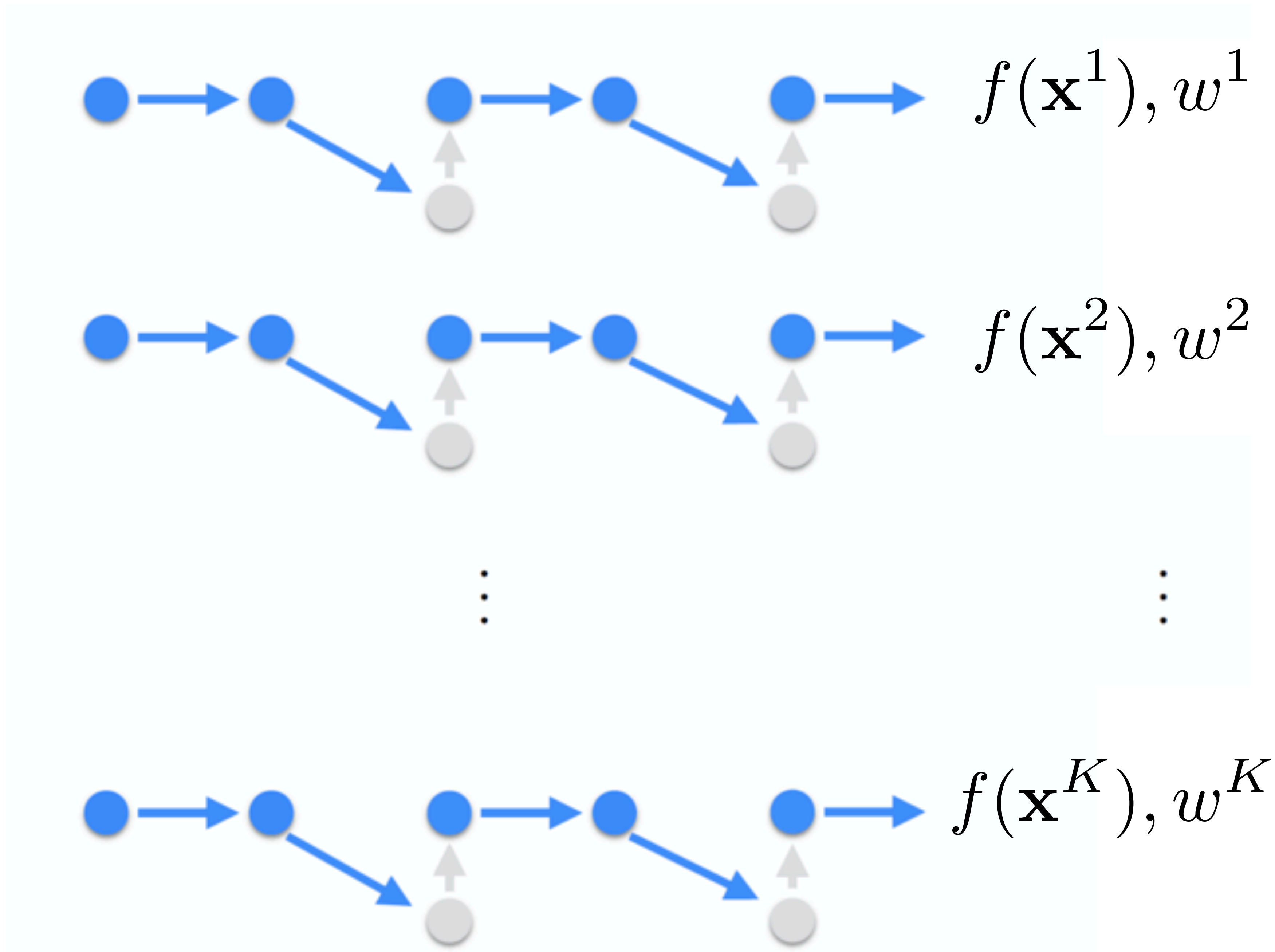
- Calculate importance weights as follows:

$$w(\mathbf{x}^k) = \frac{\gamma(\mathbf{x}^k)}{q(\mathbf{x}^k)} = \prod_{i=1}^{N^k} g_i^k(y_i^k|\phi_i^k) \qquad W^k = \frac{w(\mathbf{x}^k)}{\sum_{\ell=1}^{K} w(\mathbf{x}^\ell)}$$

- Approximate expectation by Monte Carlo integration

$$\mathbb{E}_{\pi(\mathbf{x})}\left[f(\mathbf{x})\right] \approx \sum_{k=1}^{K} W^k f(\mathbf{x^k})$$

Importance sampling



Single-Site *Metropolis–Hastings*

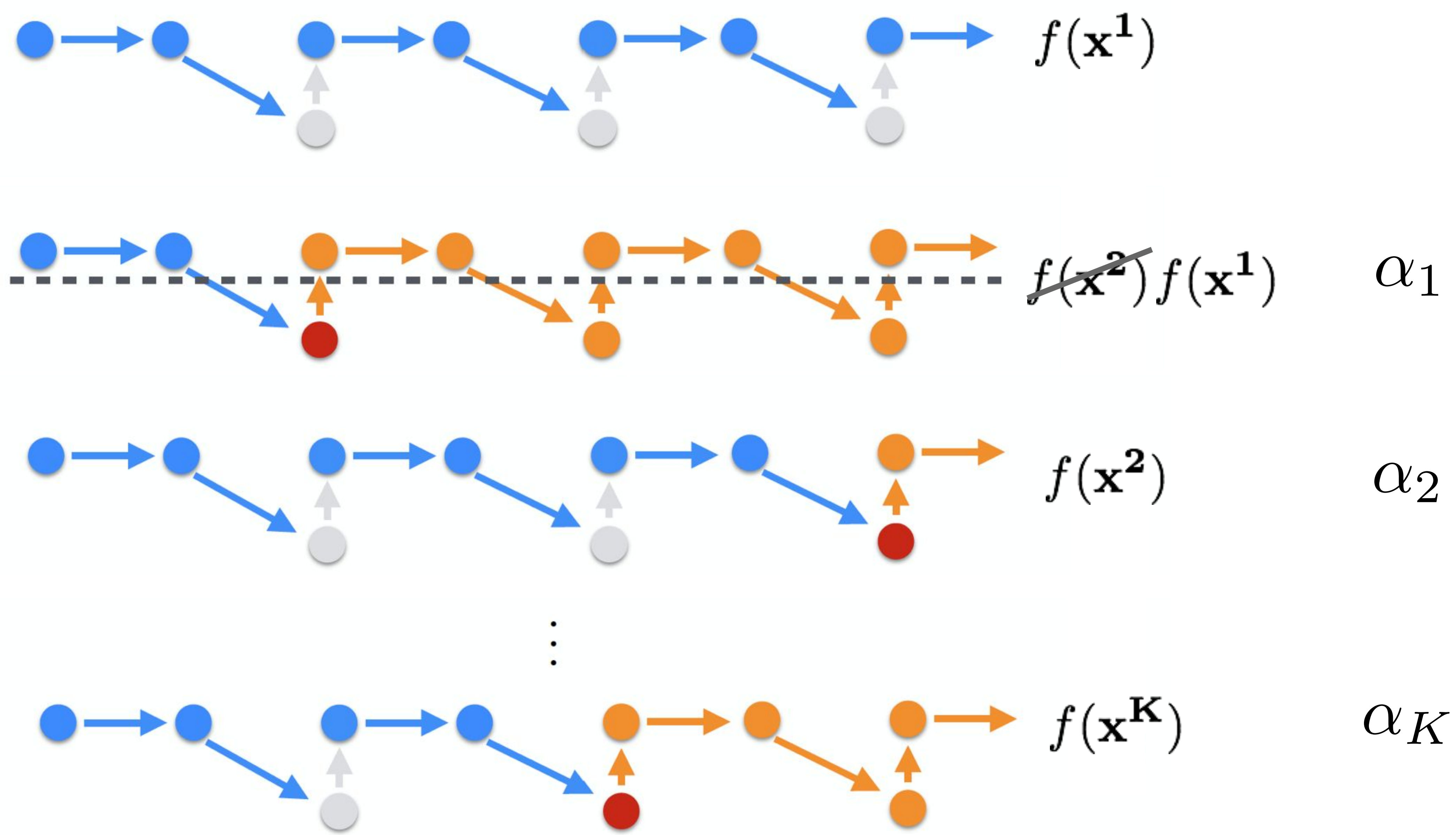
Want samples from $\pi(\mathbf{x}) \triangleq p(\mathbf{x}|\mathbf{y}) = \dfrac{\gamma(\mathbf{x})}{Z}$

- Pick a proposal distribution $q(\mathbf{x}'|\mathbf{x})$ that generates a new trace given current trace

- Use Metropolis–Hastings acceptance

$$\alpha = \min\left(1, \frac{\pi(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{\pi(\mathbf{x})q(\mathbf{x}'|\mathbf{x})}\right)$$

Single-Site Metropolis–Hastings



Single-Site Metropolis–Hastings

$$q(\mathbf{x}'|\mathbf{x}^s) = \frac{1}{M^s}\kappa(x_l'|x_l^s)\prod_{j=l+1}^{M'} f_j'(x_j'|\theta_j')$$

$M^s =$ Number of random elements in old trace

$\kappa(x_l'|x_l^s) =$ Proposal distribution for the *lth random element*

Can set $\kappa(x_m'|x_m) = f_m(x_m'|\theta_m), \theta_m = \theta_m'$

What did we cover?

WebPPL

CHURCH

VENTURE

ANGLICAN

MONAD-BAYES

What did we miss?

PYRO

PROB TORCH

edwardlib.org

That’s all folks!