Probabilistic Programming Languages

Build Your Owl PPL

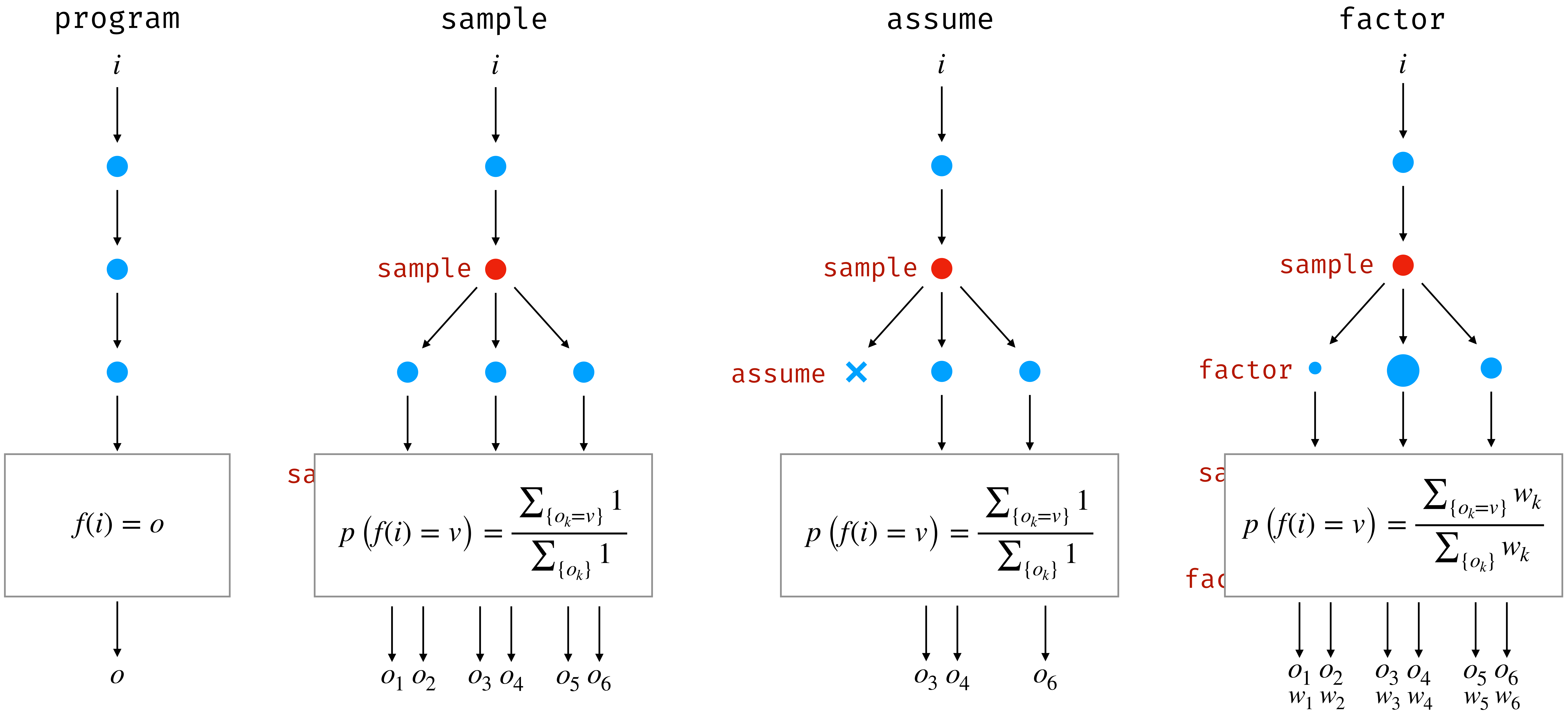
Guillaume Baudart

MPRI 2023-2024

Reminders

BYO-PPL

infer : $(\alpha \to \beta) \to \alpha \to \beta$ dist



Semantics: (un)normalized measures

set of possible outcomes

model

$$\{\!\!\{ e \}\!\!\}^{\phi}_{\gamma} : \Sigma_{[\![t]\!]} \to [0, \infty)$$

score

environments

$$
\begin{aligned}
\{\!\!\{ e \}\!\!\}^{\phi}_{\gamma} &= \lambda U.\ \delta_{[\![e]\!]^{\phi}_{\gamma}}(U) \text{ if } \textit{kindOf}(e) = D \\
\{\!\!\{ \texttt{let}\ p = e_1\ \texttt{in}\ e_2 \}\!\!\}^{\phi}_{\gamma} &= \lambda U.\ \int_{[\![\textit{typeOf}(e_1)]\!]} \{\!\!\{ e_1 \}\!\!\}^{\phi}_{\gamma}(dv)\ \{\!\!\{ e_2 \}\!\!\}^{\phi}_{\gamma+[p \leftarrow v]} \\
\{\!\!\{ \texttt{sample}(e) \}\!\!\}^{\phi}_{\gamma} &= \lambda U.\ [\![e]\!]^{\phi}_{\gamma}(U) \\
\{\!\!\{ \texttt{factor}(e) \}\!\!\}^{\phi}_{\gamma} &= \lambda U.\ [\![e]\!]^{\phi}_{\gamma} \times \delta_{()}(U) \\
\{\!\!\{ \texttt{observe}(e_1, e_2) \}\!\!\}^{\phi}_{\gamma} &= \lambda U.\ \mathrm{pdf}([\![e_1]\!]^{\phi}_{\gamma})([\![e_2]\!]^{\phi}_{\gamma}) \cdot \delta_{()}(U)
\end{aligned}
$$

$$
[\![\texttt{infer}(e)]\!]^{\phi}_{\gamma} = \begin{cases} \dfrac{\lambda U.\ \{\!\!\{ e \}\!\!\}^{\phi}_{\gamma}(U)}{\{\!\!\{ e \}\!\!\}^{\phi}_{\gamma}([\![\textit{typeOf}(e)]\!])} \\[2em] \textit{Error if } \{\!\!\{ e \}\!\!\}^{\phi}_{\gamma}([\![\textit{typeOf}(e) \in \{0, \infty\}]\!] \end{cases}
$$

Exercises

Prove the following properties

- ```
  sample mu (* where mu is defined on [a, b] *)
  ≡
  let x = sample (uniform (a, b)) in
  let () = observe (mu, x) in
  x
  ```

- ```
  observe (mu, x) (* where mu is a discrete distribution *)
  ≡
  let y = sample mu in
  assume x = y
  ```

- ```
  sample (bernoulli (0.5))
  ≡
  let x = sample (gaussian (0., 1.)) in
  x > 0
  ```
  ```

Build Your Owl PPL

BYO-PPL

Outline

For a given inference algoritm, how to implement `sample`, `assume`, `factor`, `observe`, and `infer`?

I - Problem with basic inference
- Curse of dimensionality
- Resampling and checkpoints

II - Continuation Passing Style (CPS) models

III - Revisiting inference with CPS programming
- Sample generation
- Importance sampling
- Particle filter

IV - Inference formalization
- Weighted samplers
- Big-step semantics with checkpoints

HMM: Hidden Markov Model

Track the position of an agent from noisy observations

- The current position should not be too far from the previous position
- The observations should not be too far from the current position

Probabilistic model: $\forall t \in \mathbb{N}$

- $x_t \sim \mathcal{N}(x_{t-1}, \textit{speed})$
- $y_t \sim \mathcal{N}(x_t, \textit{noise})$

HMM: Hidden Markov Model

hmm.ml

```
open Basic.Importance_sampling

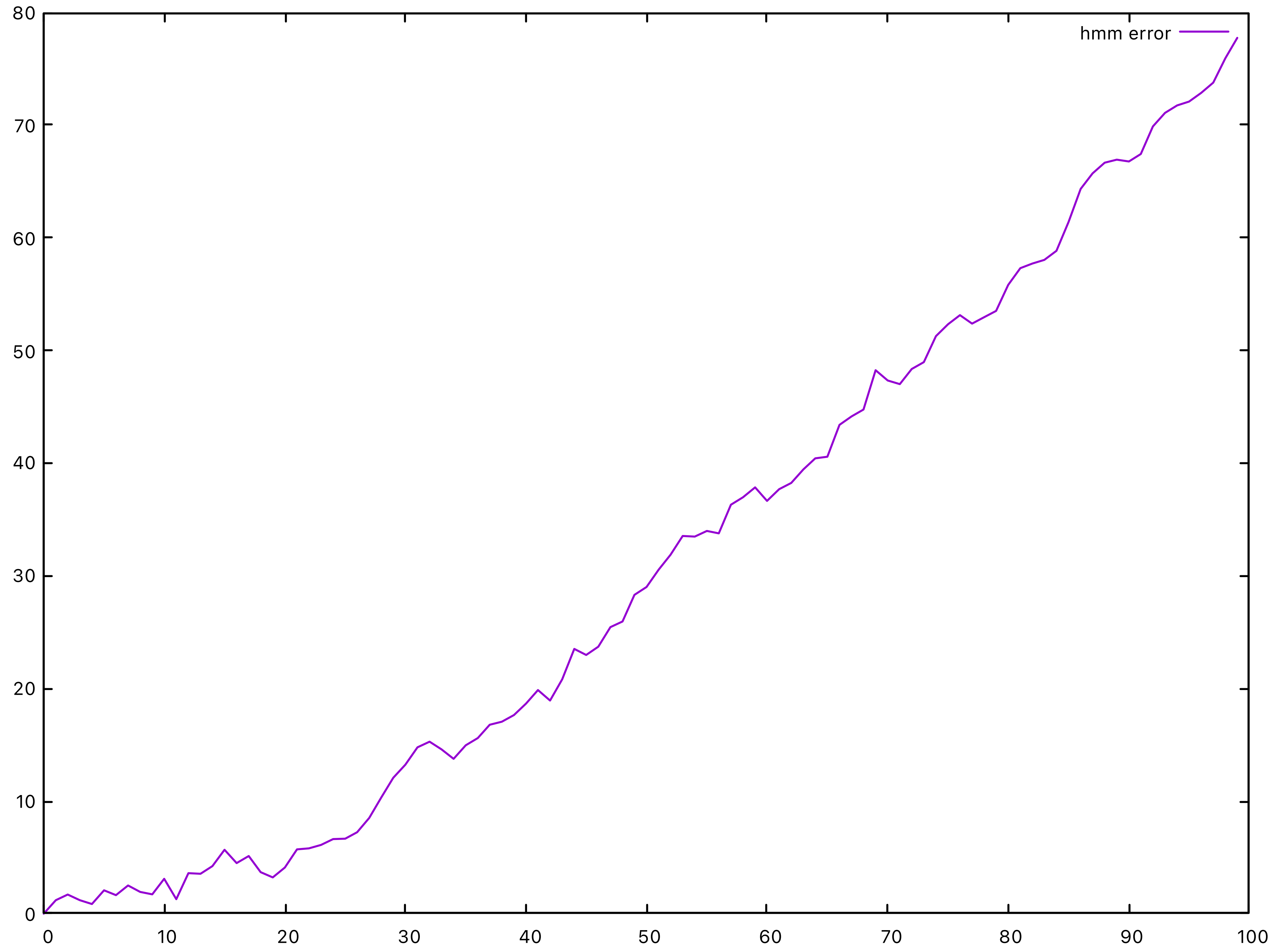
let hmm prob data =
  let rec gen states data =
    match (states, data) with
    | [], y :: data -> gen [ y ] data
    | states, [] -> states
    | pre_x :: _, y :: data ->
        let x = sample prob (gaussian ~mu:pre_x ~sigma:1.0) in
        let () = observe prob (gaussian ~mu:x ~sigma:1.0) y in
        gen (x :: states) data
  in
  gen [] data

let _ =
  let data = Owl.Arr.linspace 0. 20. 20 |> Owl.Arr.to_array |> Array.to_list in
  let dist = Distribution.split_list (infer hmm data) in
  let m_x = List.rev (List.map Distribution.mean dist) in
  List.iter2 (Format.printf "%f >> %f@.") data m_x
```

HMM: Hidden Markov Model

```
❯ dune exec ./examples/hmm.exe

0.000000 >> 0.000000
1.052632 >> 0.278989
2.105263 >> 2.923428
3.157895 >> 2.812035
4.210526 >> 2.328341
5.263158 >> 1.742109
6.315789 >> 2.518105
7.368421 >> 3.958375
8.421053 >> 5.946233
9.473684 >> 7.329554
10.526316 >> 9.293653
11.578947 >> 10.181831
12.631579 >> 8.549409
13.684211 >> 9.323073
14.736842 >> 9.280692
15.789474 >> 9.352218
...
```



HMM: Importance sampling

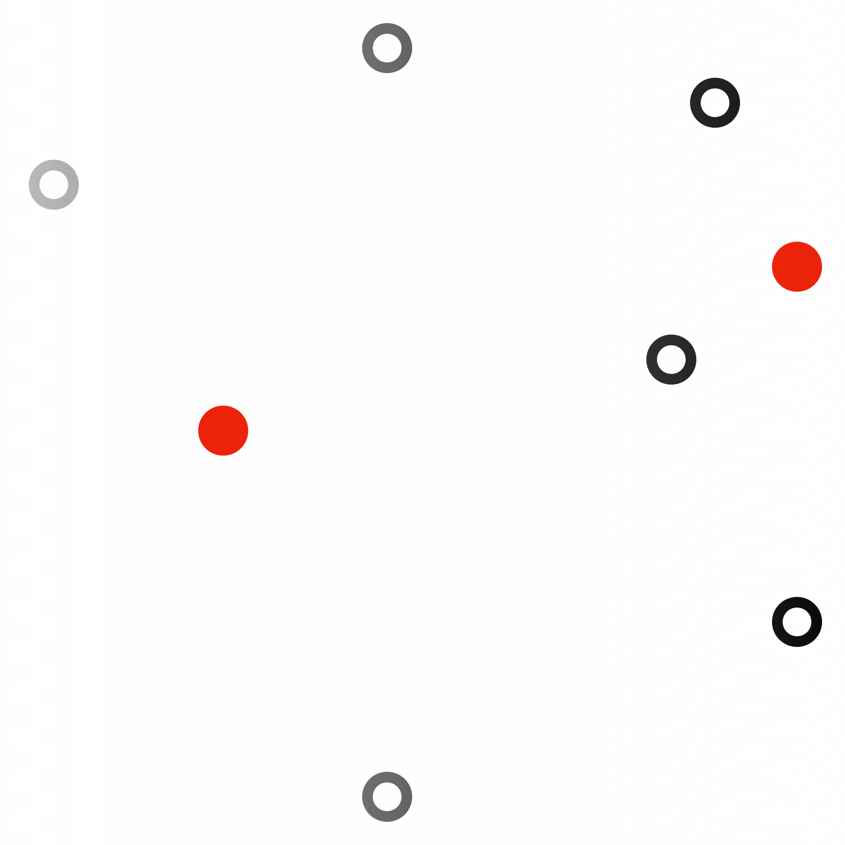
Problem:

- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement

HMM: Importance sampling

Problem:

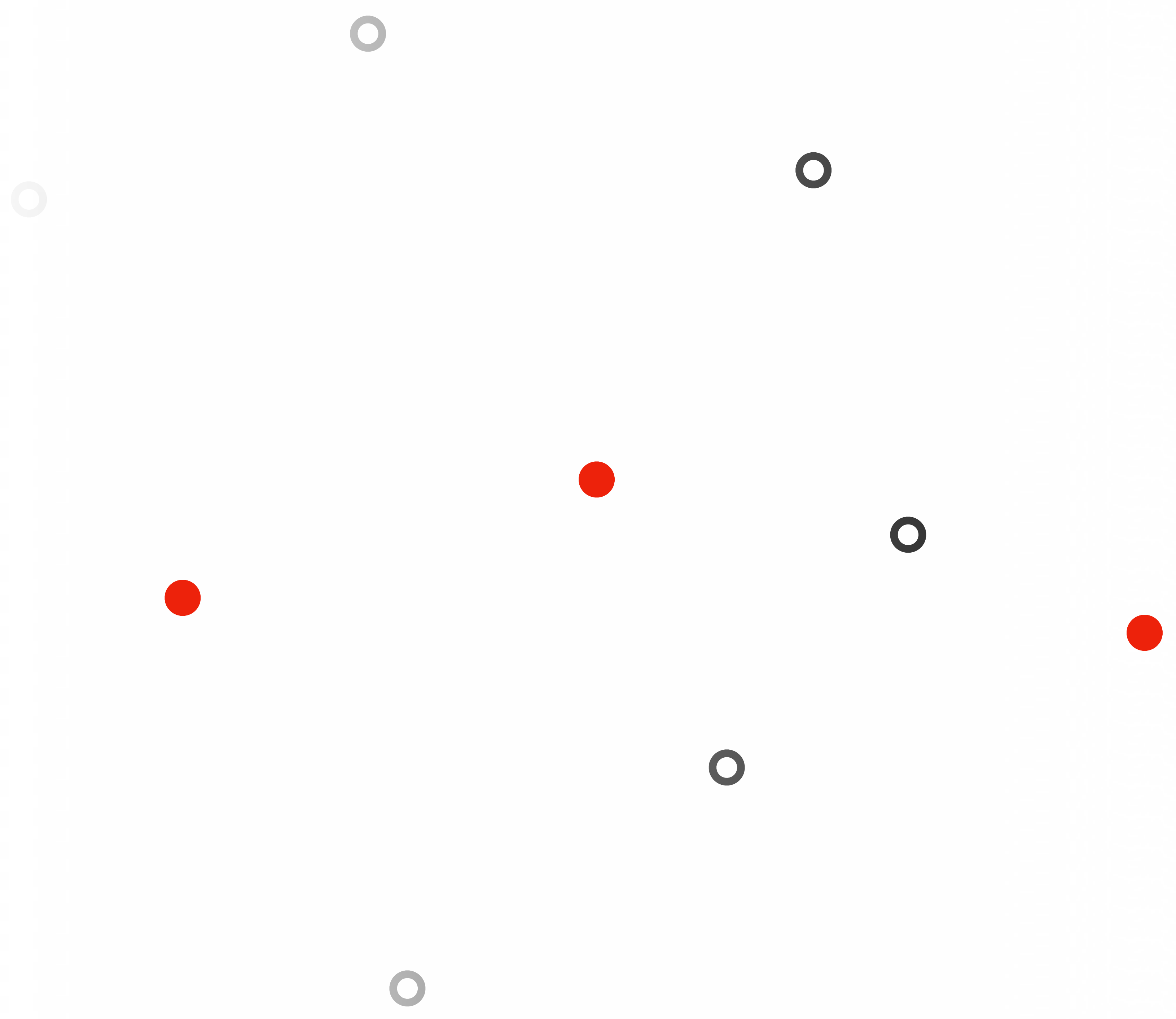
- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement



HMM: Importance sampling

Problem:

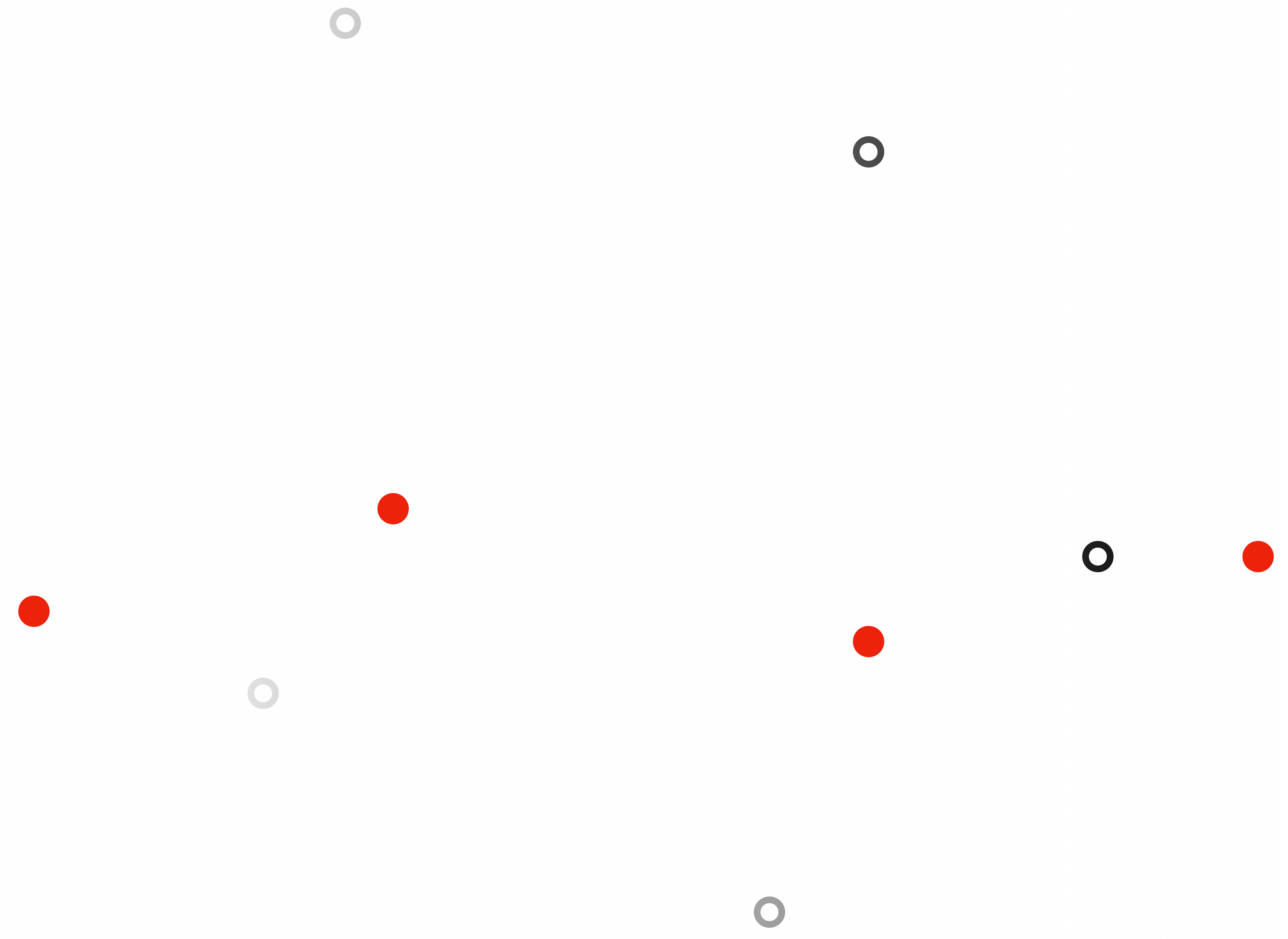
- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement



HMM: Importance sampling

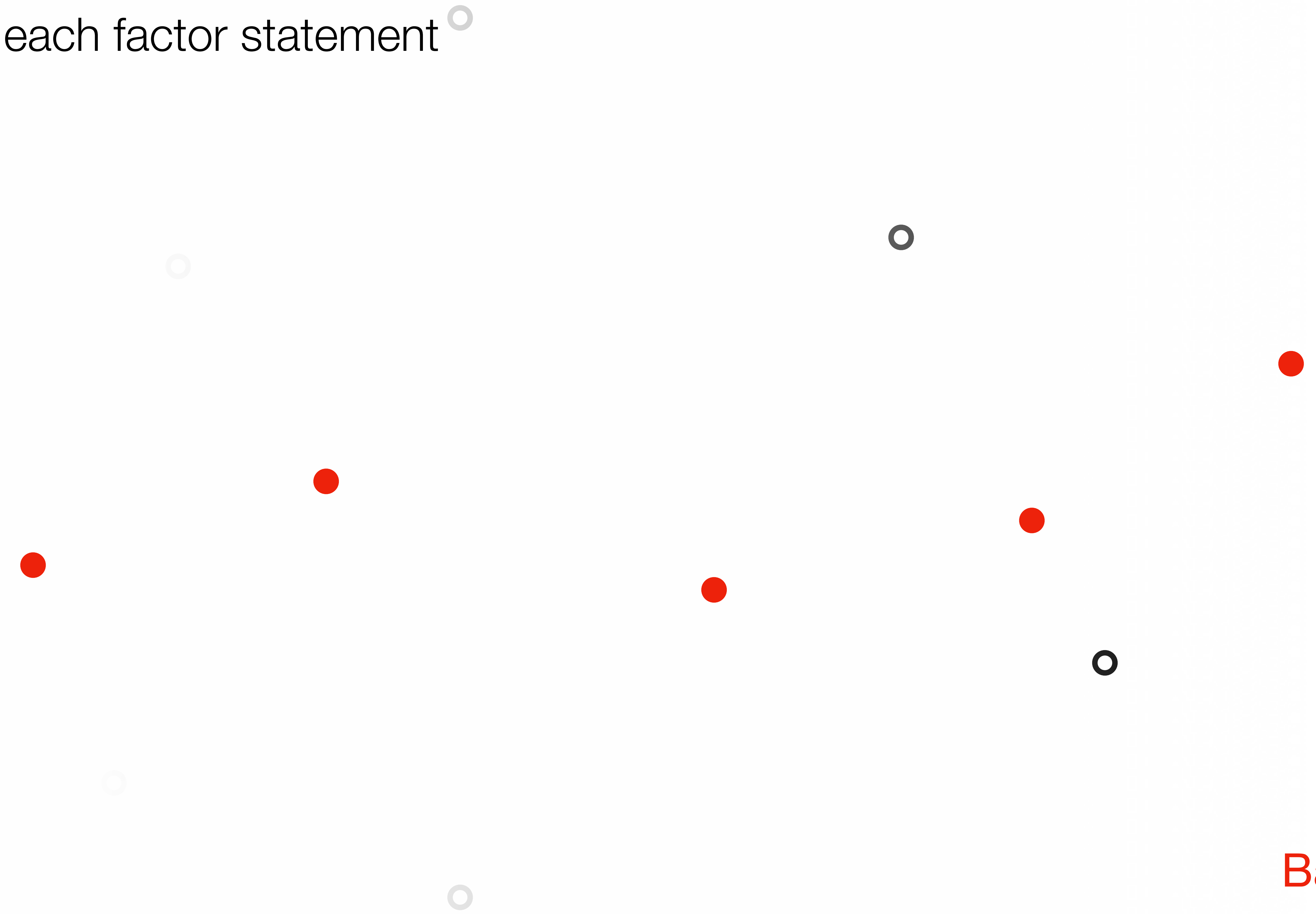
Problem:

- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement



HMM: Importance sampling

Problem:
- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement



Bad estimation

The curse of dimensionality

Problem becomes harder as the dimension increases

Basic inference: rejection sampling, importance sampling
- Performances decrease exponentially when the dimension increases
- Only use for low-dimension models

How to mitigate this problem?
- Make assumptions about the posterior distributions
- Break the problem into simpler, smaller problems

17h45mn

Sven Sachsalber, *Looking for a needle in the Haystack*, 2014

HMM: Particle filter

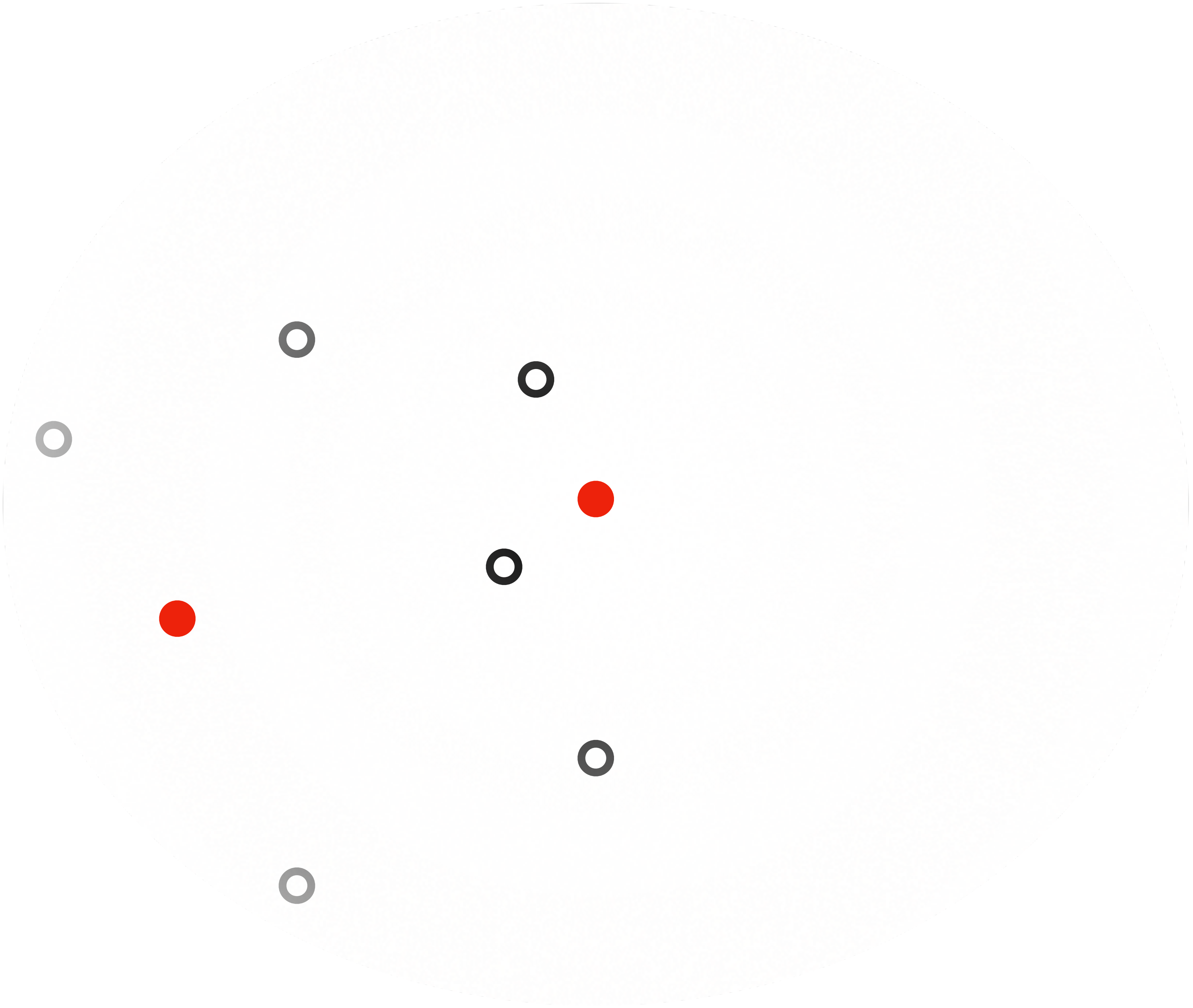
Add a resampling step

- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution

HMM: Particle filter

Add a resampling step

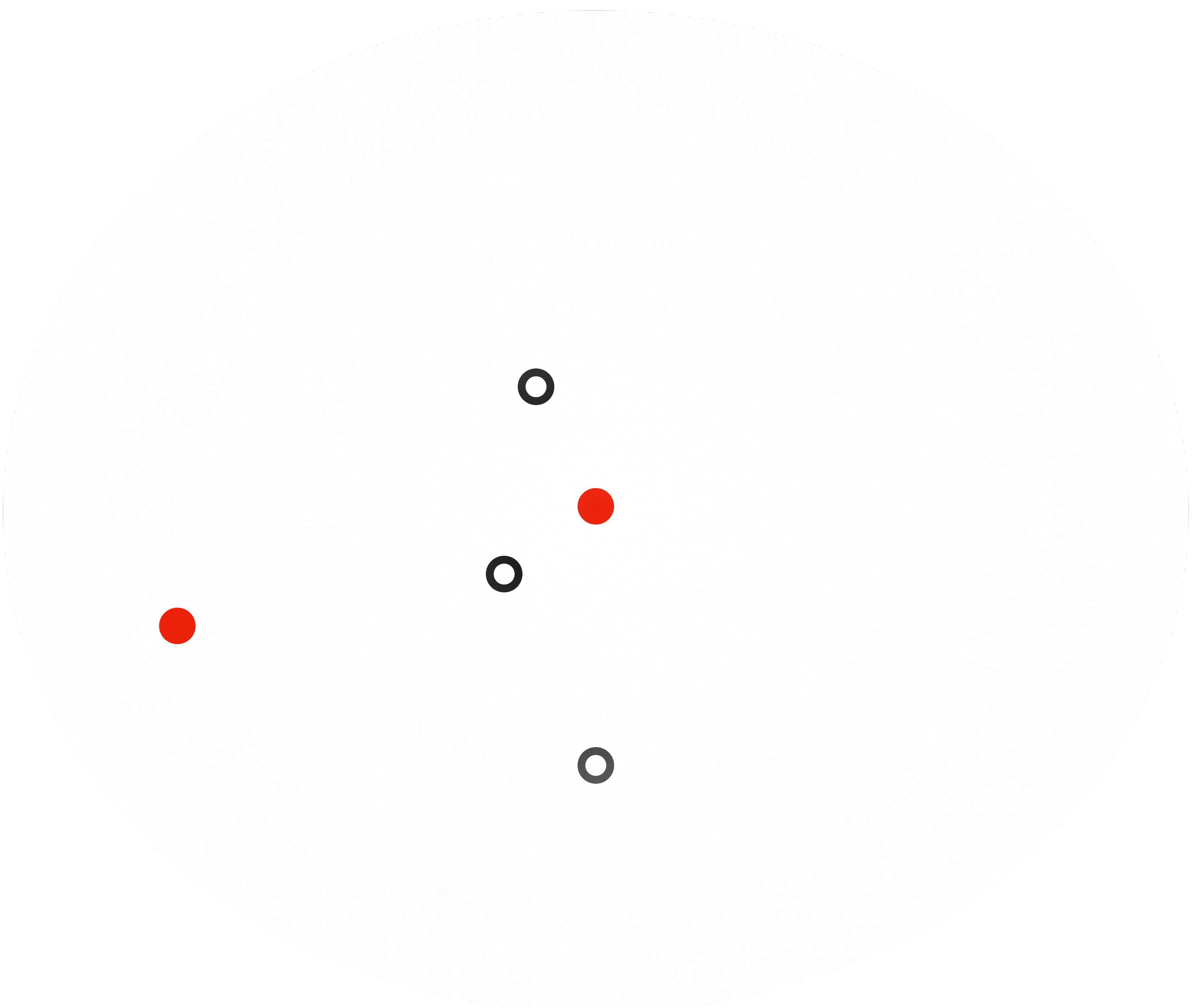
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



HMM: Particle filter

Add a resampling step

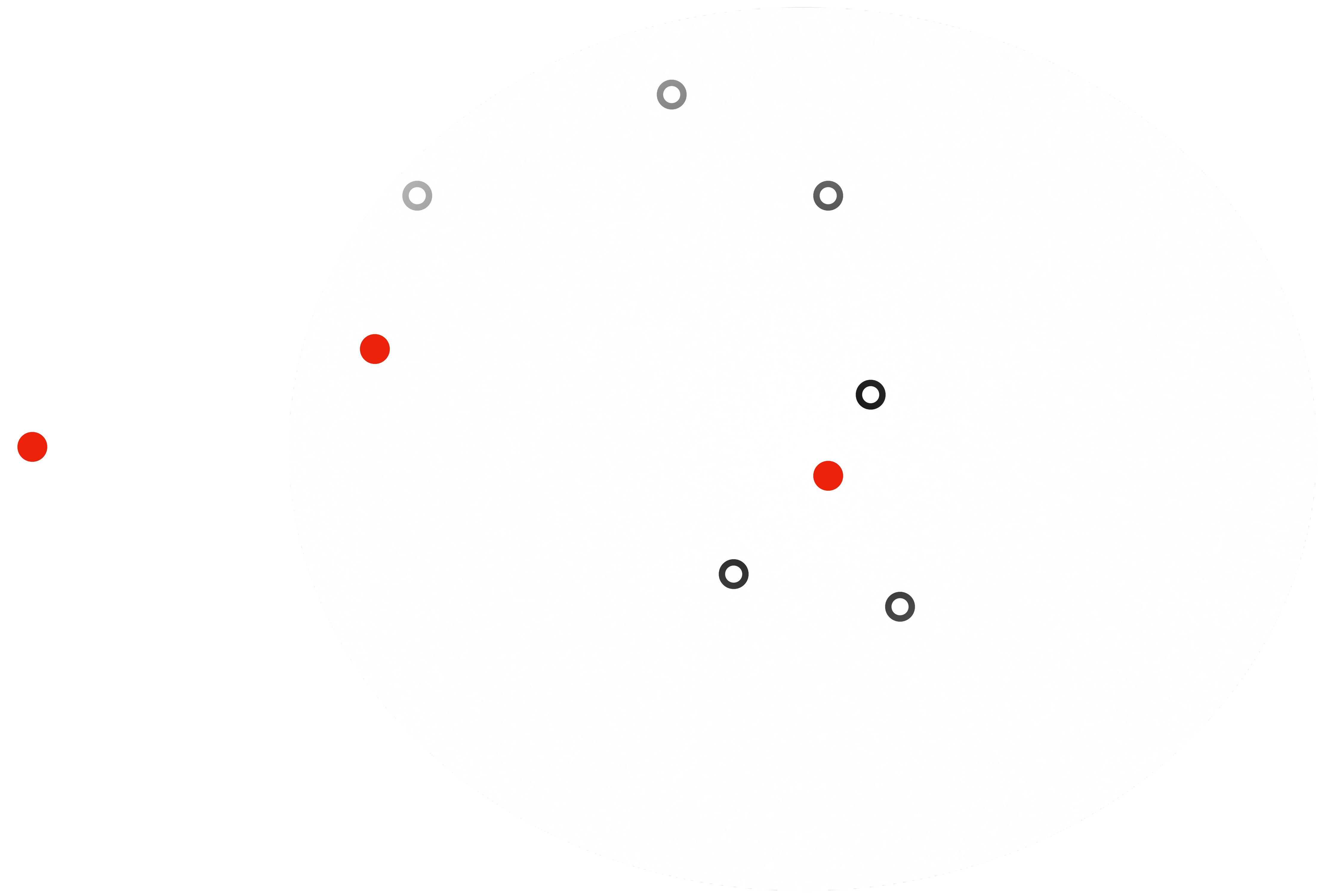
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



HMM: Particle filter

Add a resampling step

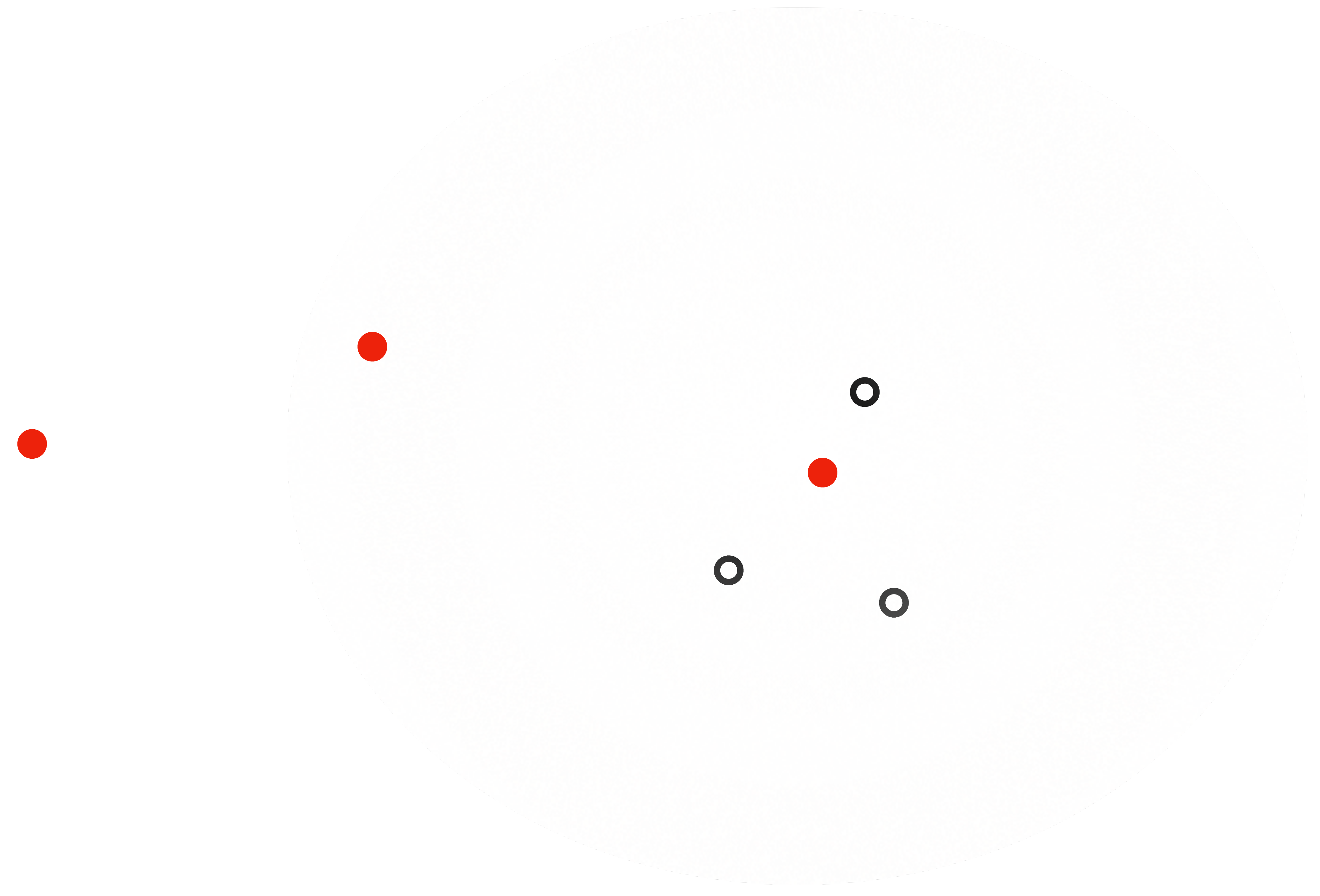
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



HMM: Particle filter

Add a resampling step

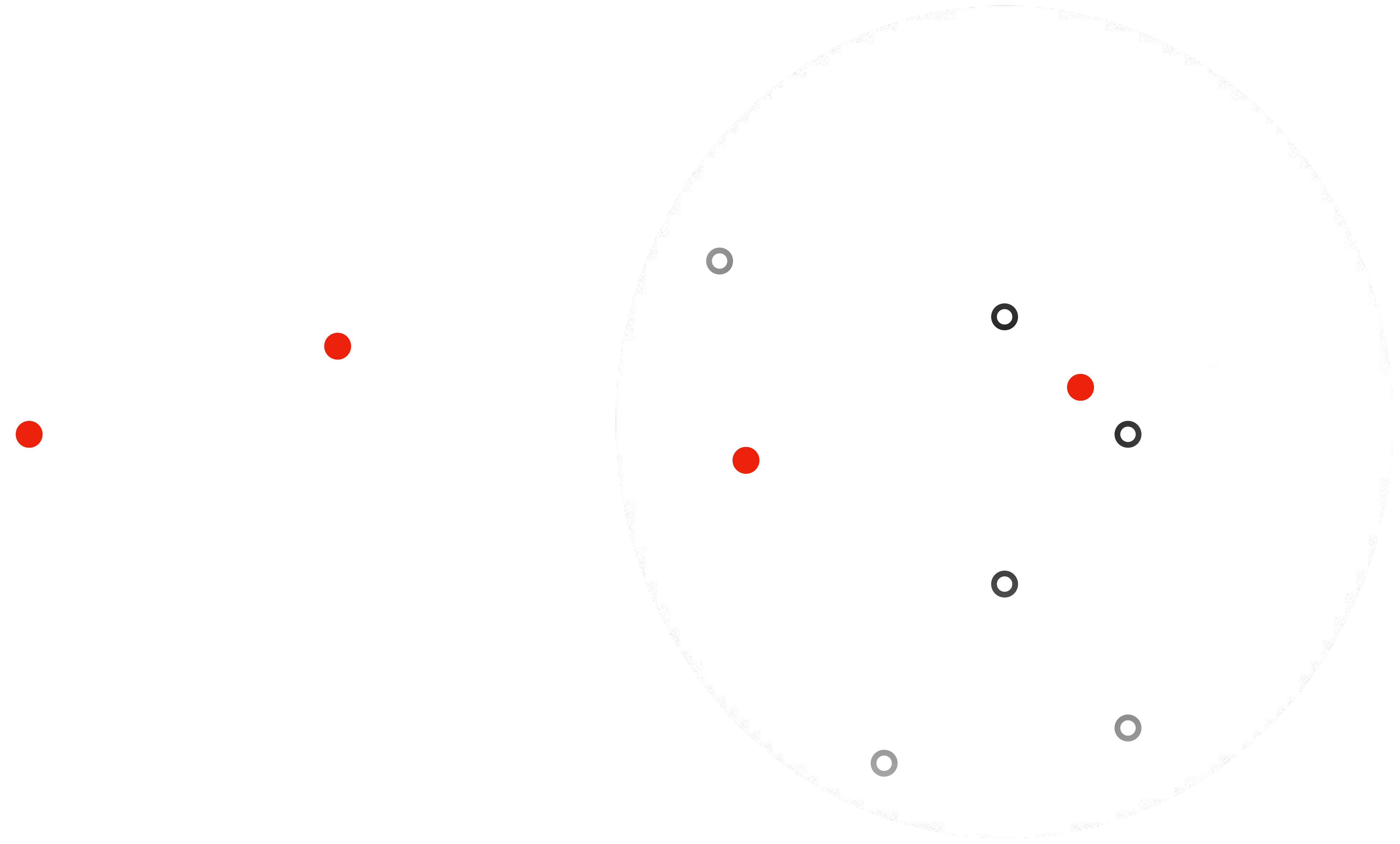
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



HMM: Particle filter

Add a resampling step

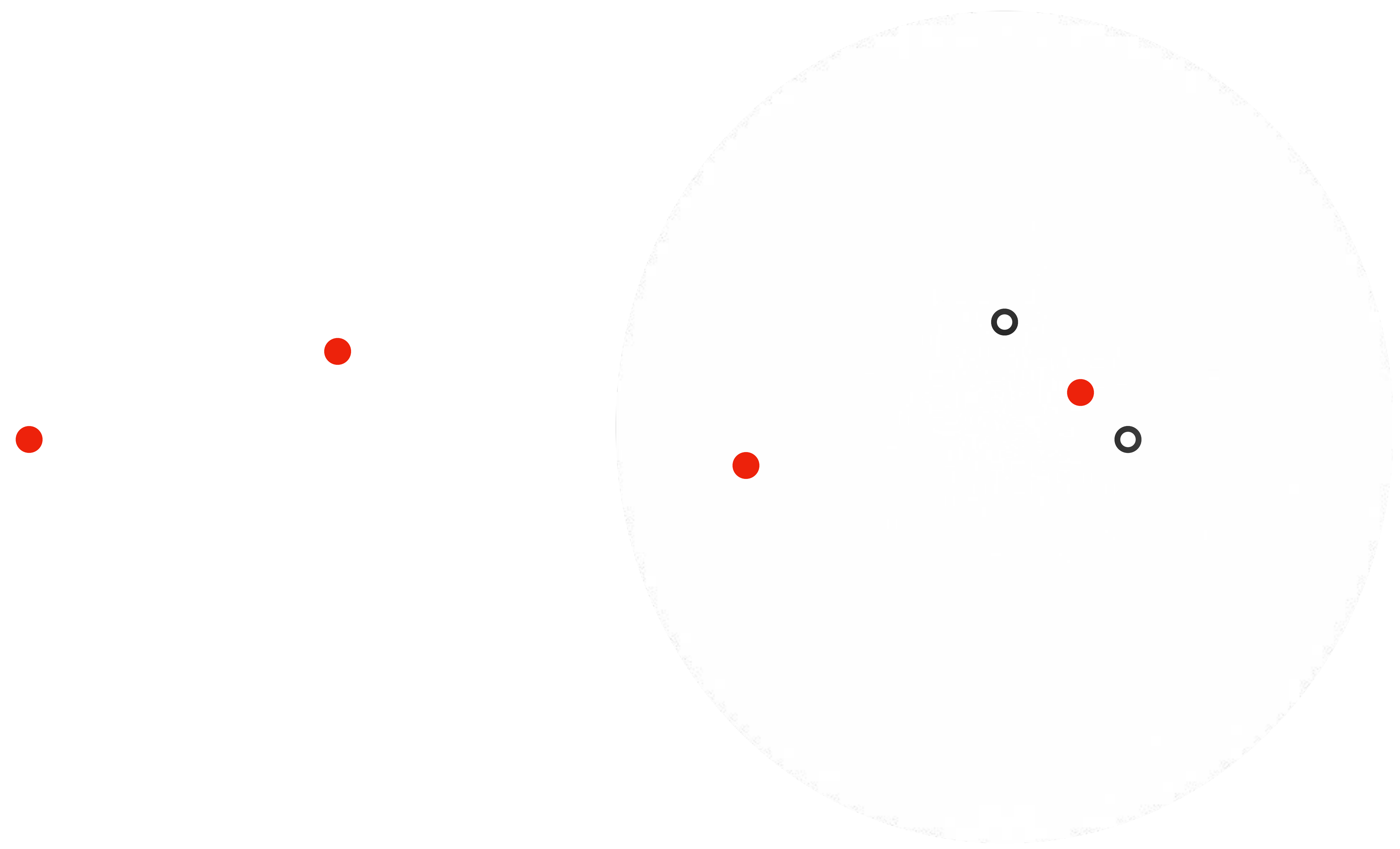
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



HMM: Particle filter

Add a resampling step

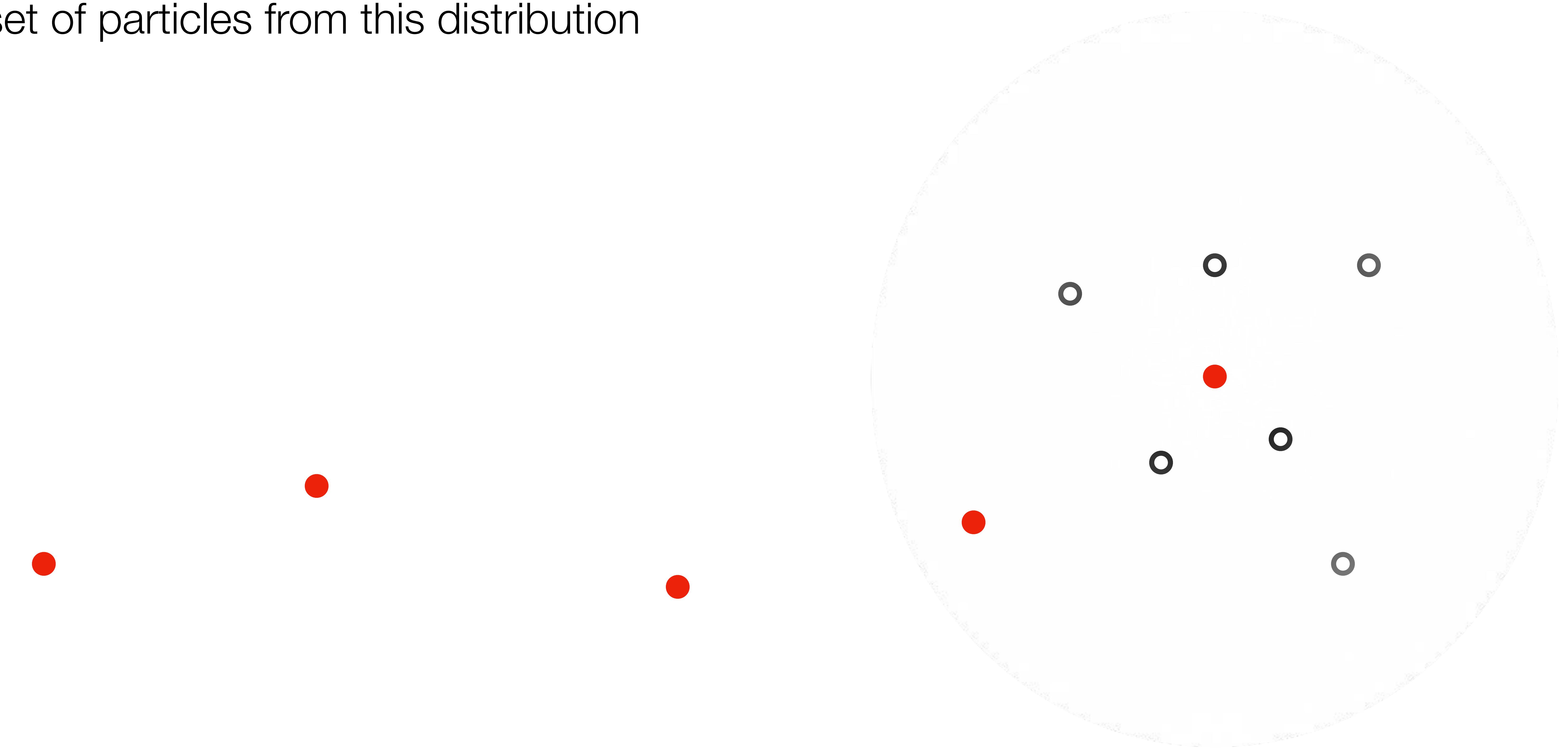
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



HMM: Particle filter

Add a resampling step

- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



Better estimation

Problem: Duplications

How can we duplicate a particle during execution?

- Rerun the particle from the start?
- Force reuse sampled values?
- Clone the memory state?

Continuation Passing Style

- Functions take an extra argument `k`: the continuation
- `k` implements what should be done with the result of the function
- In our context, we can use continuation to interrupt/restart the execution of a model

Continuation Passing Style (CPS)

BYO-PPL

Reminders: CPS

tree.ml

```
let rec tree_height t =
  match t with
  | Empty → 0
  | Node (_, l, r) → 1 + max (tree_height l) (tree_height r)
```

Reminders: CPS

tree.ml

```
let rec tree_height t =
  match t with
  | Empty → 0
  | Node (_, l, r) → 1 + max (tree_height l) (tree_height r)
```

```
let rec tree_height t =
  match t with
  | Empty → 0
  | Node (_, l, r) →
    let hl = tree_height l in
    let hr = tree_heigh r in
    (1 + max hl hr)
```

1. Add intermediate values

Reminders: CPS

tree.ml

```
let rec tree_height t =
  match t with
  | Empty → 0
  | Node (_, l, r) → 1 + max (tree_height l) (tree_height r)
```

```
let rec tree_height t k =
  match t with
  | Empty → k 0
  | Node (_, l, r) →
    let hl = tree_height l in
    let hr = tree_heigh r in
    k (1 + max hl hr)
```

1. Add intermediate values
2. Add call to continuation

Reminders: CPS

tree.ml

```
let rec tree_height t =
  match t with
  | Empty → 0
  | Node (_, l, r) → 1 + max (tree_height l) (tree_height r)
```

```
let rec tree_height t k =
  match t with
  | Empty → k 0
  | Node (_, l, r) →
    tree_height l (fun hl →
      tree_heigh r (fun hr →
        k (1 + max hl hr)))
```

1. Add intermediate values
2. Add call to continuation
3. Turn let/in into nested function call

Funny bernoulli CPS

funny_bernoulli.ml

```
let funny_bernoulli () =
  let a = sample (bernoulli ~p:0.5) in
  let b = sample (bernoulli ~p:0.5) in
  let c = sample (bernoulli ~p:0.5) in
  let () = assume (a = 1 || b = 1) in
  a + b + c
```

1. Add intermediate values
2. Add call to continuation
3. Turn let/in into nested function call

Funny bernoulli CPS

funny_bernoulli.ml

```
let funny_bernoulli () =
  let a = sample (bernoulli ~p:0.5) in
  let b = sample (bernoulli ~p:0.5) in
  let c = sample (bernoulli ~p:0.5) in
  let () = assume (a = 1 || b = 1) in
  a + b + c
```

```
let funny_bernoulli () k =
  sample (bernoulli ~p:0.5) (fun a →
    sample (bernoulli ~p:0.5) (fun b →
      sample (bernoulli ~p:0.5) (fun c →
        assume (a = 1 || b = 1) (fun () →
          k (a + b + c))))
```

1. Add intermediate values
2. Add call to continuation
3. Turn let/in into nested function call

CPS monadic operators

cps_operators.ml

```
let return e k = k e

let ( let* ) e f k = e (fun x → f x k)    (* let* x = e in f(x) *)
```

CPS monadic operators

cps_operators.ml

```
let return e k = k e

let ( let* ) e f k = e (fun x → f x k)    (* let* x = e in f(x) *)
```

```
let funny_bernoulli () k =
  sample (bernoulli ~p:0.5) (fun a →
    sample (bernoulli ~p:0.5) (fun b →
      sample (bernoulli ~p:0.5) (fun c →
        assume (a = 1 || b = 1) (fun () →
          k (a + b + c)))))
```

```
let funny_bernoulli () =
  let* a = sample (bernoulli ~p:0.5) in
  let* b = sample (bernoulli ~p:0.5) in
  let* c = sample (bernoulli ~p:0.5) in
  let* () = assume (a = 1 || b = 1) in
  return (a + b + c)
```

Sample generation (CPS)

BYO-PPL

CPS models


```
module Gen : sig
  type 'a prob
  and 'a next = 'a prob → 'a prob
  and ('a, 'b) model = 'a → ('b → 'b next) → 'b next

  val sample : 'a Distribution.t → ('a → 'b next) → 'b next
  val factor : float → (unit → 'b next) → 'b next
  val draw: ('a, 'b) model → 'a → 'b
end = struct ... end
```

Type 'a prob

- Store all information required for inference (e.g., particles array)
- Type `('a, 'b) model` capture input/output types

Models where probabilistic constructs are CPS functions

- Two arguments: input `'a` and a continuation on the return value `('b → 'b next)`.
- The return value is a continuation `'a next` that updates a probabilistic state of type `'a prob`.

Sample generation

```
let model data =
  ...
  let x = sample ... in
  ...
  let () = factor ... in
  ...
  output
```

exit

o

Sample generation

infer.ml

```
module Gen = struct
  type 'a prob = 'a option
  and 'a next = 'a prob → 'a prob
  and ('a, 'b) model = 'a → ('b → 'b next) → 'b next

  let exit v _prob = Some v

  let sample d k prob =
    let v = Distribution.draw d in
    k v prob

  let factor _s k prob = k () prob

  let draw m data =
    let v = (m data) exit None in
    Option.get v
end
```

Funny bernoulli

funny_bernoulli.ml

```
open Infer.Gen

let funny_bernoulli () =
  let* a = sample (bernoulli ~p:0.5) in
  let* b = sample (bernoulli ~p:0.5) in
  let* c = sample (bernoulli ~p:0.5) in
  let* () = assume (a = 1 || b = 1) in
  return (a + b + c)

let _ =
  for _ = 1 to 10 do
    let v = draw funny_bernoulli () in
    Format.printf "%d " v
  done
```

```
❯ dune exec ./examples/funny_bernoulli.exe

1 1 2 2 2 2 2 1 3 2
```

Importance sampling (CPS)

BYO-PPL

Importance sampling

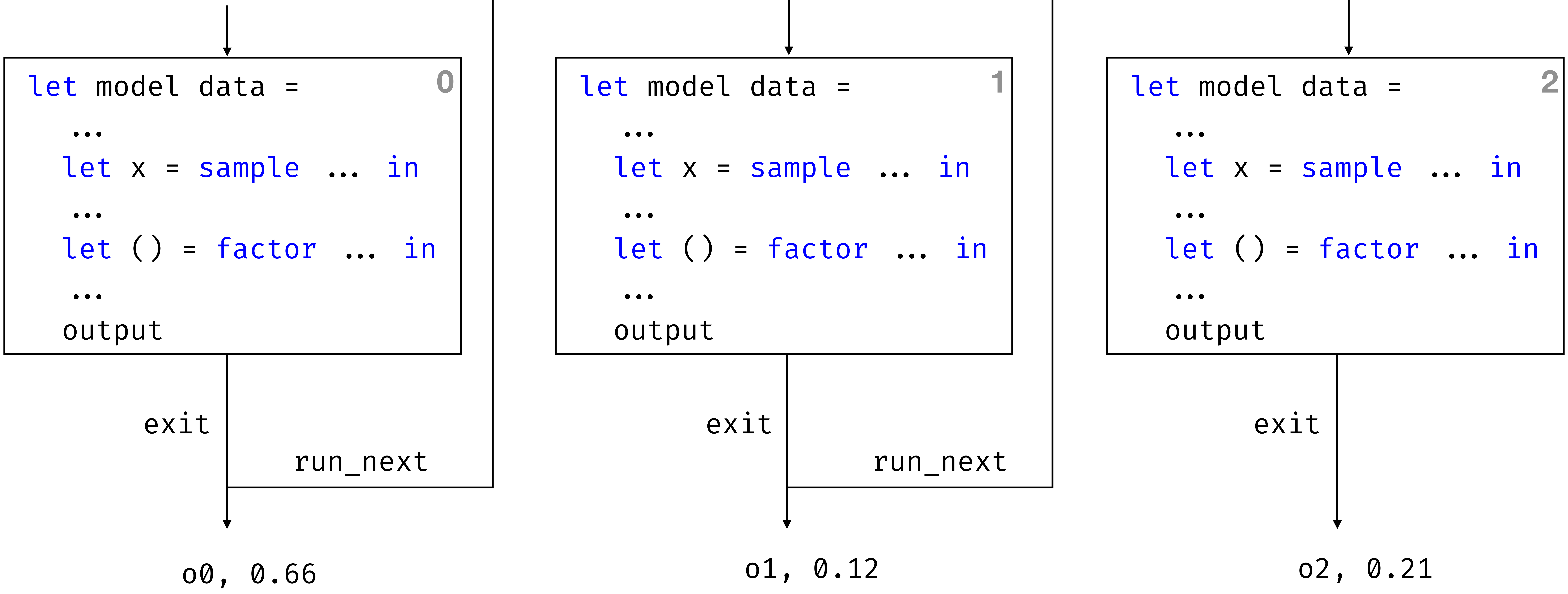

```
module Importance_sampling : sig
  type 'a prob
  and 'a next = 'a prob → 'a prob
  and ('a, 'b) model = 'a → ('b → 'b next) → 'b next

  val sample : 'a Distribution.t → ('a → 'b next) → 'b next
  val factor : float → (unit → 'b next) → 'b next
  val infer : ('a, 'b) model → 'a → 'b Distribution.t
end = struct ... end
```

Inference algorithm

- Run a set of n independent executions
- `sample`: draw a sample from a distribution
- `factor`: associate a score to the current execution
- Gather output values and score to approximate the posterior distribution

Importance sampling



Importance sampling


```
module Importance_sampling = struct
  type 'a prob =  ...

  let sample d k prob = assert false
  let factor s k prob = assert false

  let infer ?(n = 1000) m data = assert false
end
```

Importance sampling


```
module Importance_sampling = struct
  type 'a prob = { id : int; particles : 'a particle array }
  and 'a particle = { k : 'a next; value : 'a option; score : float }
  ...

  let sample d k prob =
    let v = Distribution.draw d in
    k v prob

  let factor s k prob =
    let particle = prob.particles.(prob.id) in
    prob.particles.(prob.id) ← { particle with score = s +. particle.score };
    k () prob

  ...
end
```

Importance sampling

infer.ml

```
module Importance_sampling = struct
  type 'a prob = { id : int; particles : 'a particle array }
  and 'a particle = { k : 'a next; value : 'a option; score : float }
  …

  let exit v prob =
    let particle = prob.particles.(prob.id) in
    prob.particles.(prob.id) ← { particle with value = Some v };
    prob

  let infer ?(n = 1000) model data =
    let particles = Array.make n { value = None; score = 0.; k = (model data) exit } in
    Array.iteri (fun i particle → ignore (particle.k { id = i; particles })) particles;
    let values = Array.map (fun p → Option.get p.value) particles in
    let logits = Array.map (fun p → p.score) particles in
    Distribution.support ~values ~logits
end
```

Coin

coin.ml

```
open Infer.Importance_sampling

let coin x =
  let* z = sample (uniform ~a:0. ~b:1.) in
  let* () = Cps_list.iter (observe (bernoulli ~p:z)) x in
  return z

let _ =
  let dist = infer coin [1; 1; 0; 0; 0; 0; 0; 0; 0; 0] in
  let m, s = Distribution.stats dist in
  Format.printf "Coin bias, mean:%f, std:%f@." m s
```

```
❯ dune exec ./examples/coin.exe

Coin bias, mean:0.247876, std:0.118921
Beta(2+1, 8+1), mean:0.250000, std:0.120096
```

Particle filter (CPS)

BYO-PPL

Particle filter

basic.ml

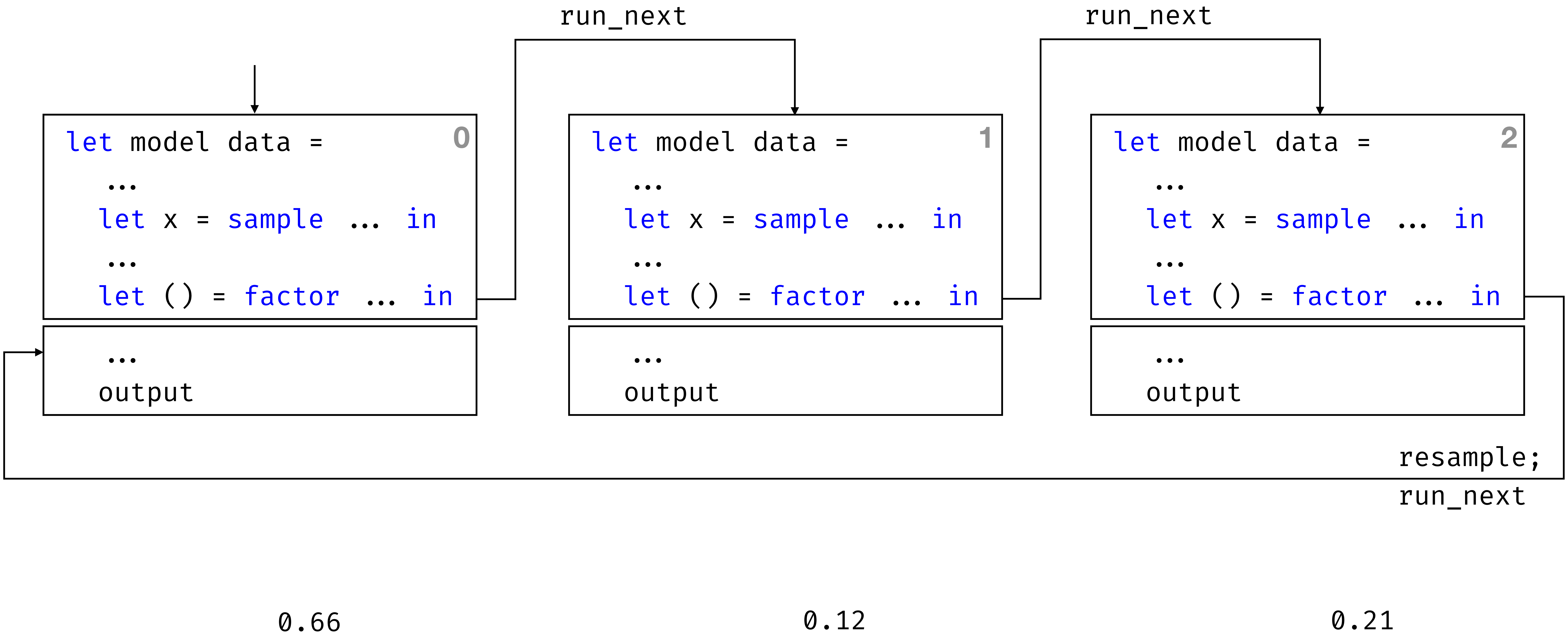
```
module Particle_filter = struct
  include Importance_sampling

  let resample particles = assert false
  let factor s k prob = assert false
end
```

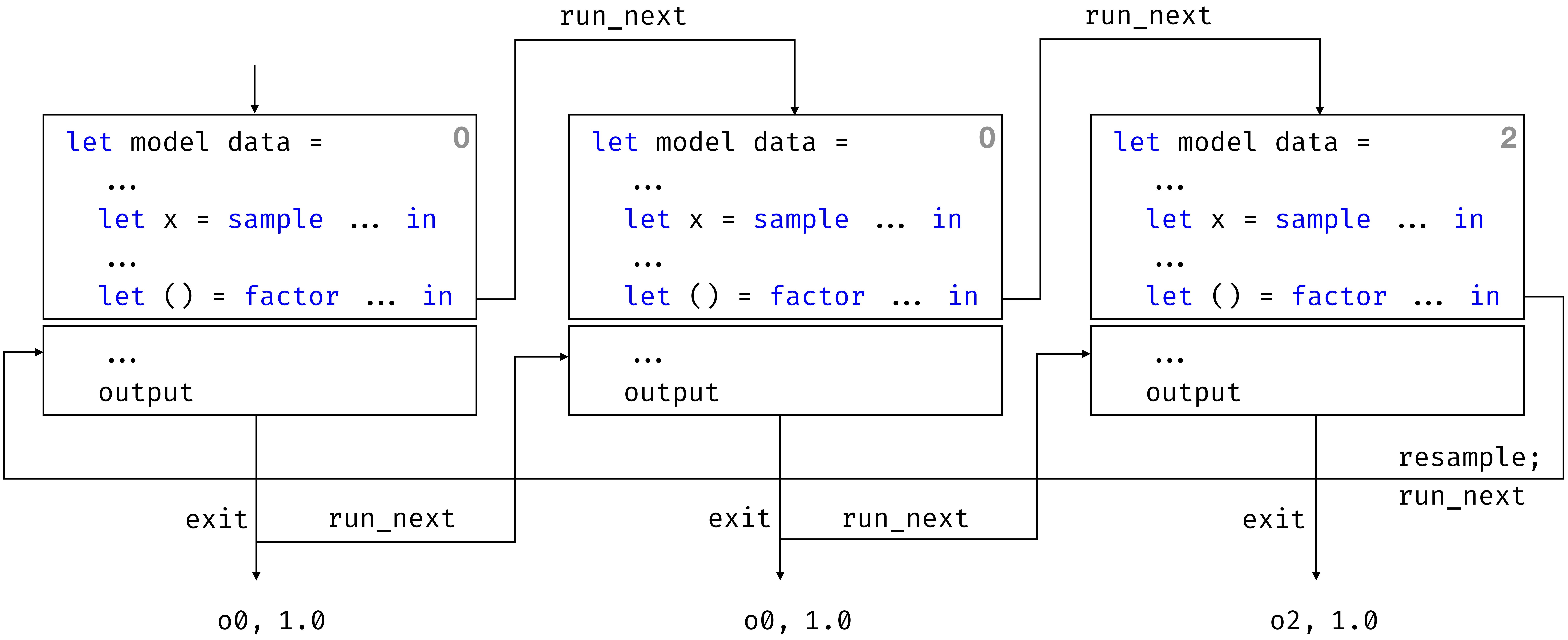
Inference algorithm : importance sampling, but...

- Add a resampling step at each `factor`
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution

Particle filter



Particle filter



Particle filter


```
module Particle_filter = struct
  include Importance_sampling

  let resample particles =
    let logits = Array.map (fun x → x.score) particles in
    let values = Array.map (fun x → { x with score = 0. }) particles in
    let dist = Distribution.support ~values ~logits in
    Array.iteri (fun i _ → particles.(i) ← Distribution.draw dist) particles

  let factor s k prob =
    let particle = prob.particles.(prob.id) in
    prob.particles.(prob.id) ← { particle with k = k (); score = s +. particle.score };
    prob
end
```

Particle filter

infer.ml

```
module Particle_filter = struct
  include Importance_sampling
  ...

  let infer ?(n = 1000) model data =
    let particles = Array.make n { value = None; score = 0.; k = (model data) exit } in
    while Array.for_all (fun p → Option.is_none p.value) particles do
      Array.iteri (fun i particle → ignore (particle.k { id = i; particles })) particles;
      resample particles
    done;
    let values = Array.map (fun p → Option.get p.value) particles in
    let logits = Array.map (fun p → p.score) particles in
    Distribution.support ~values ~logits
end
```

HMM: Hidden Markov Model

hmm.ml

```
open Infer.Particle_filter

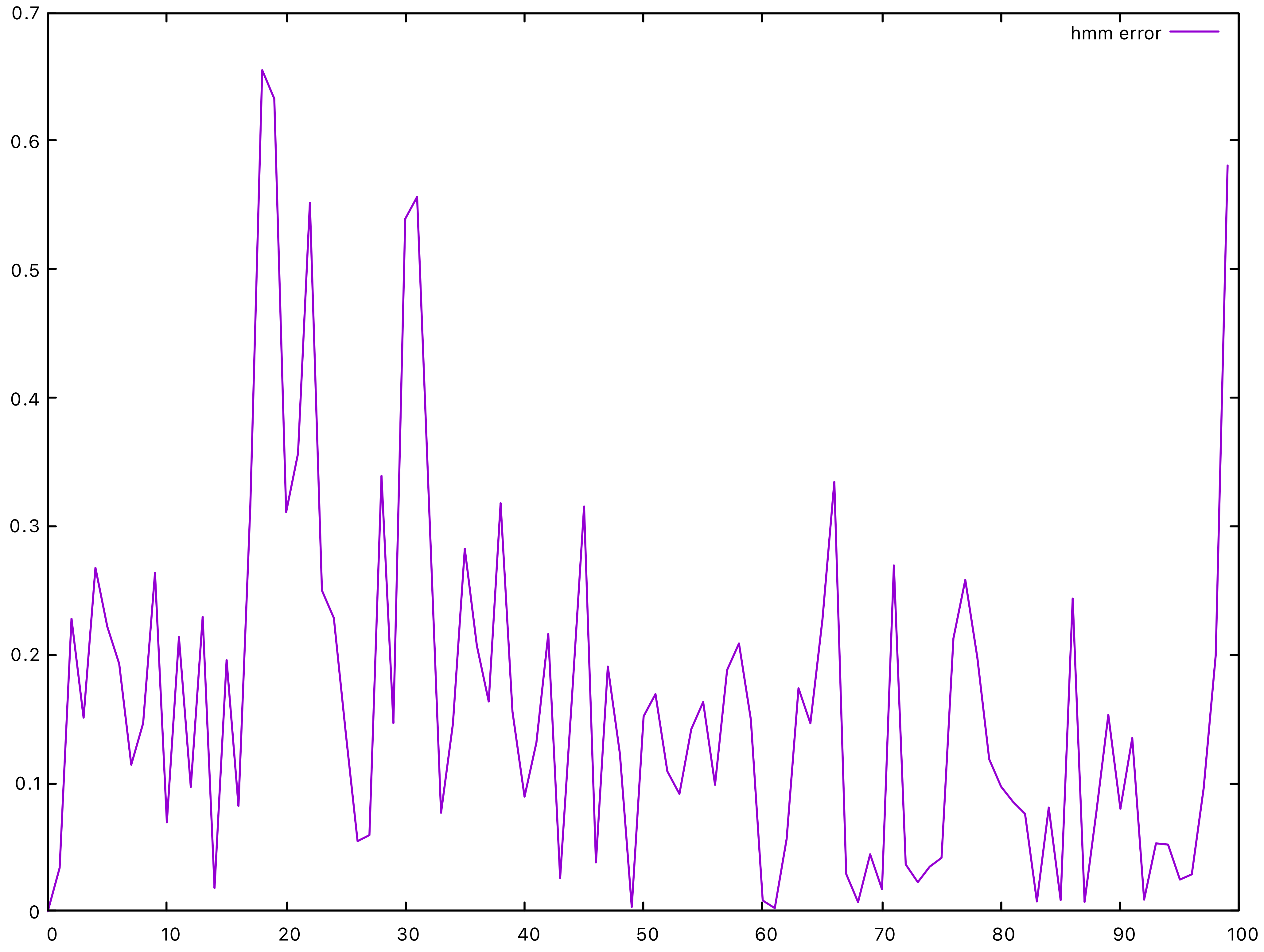
let hmm prob data =
  let rec gen states data =
    match (states, data) with
    | [], y :: data → gen [ y ] data
    | states, [] → return states
    | pre_x :: _, y :: data →
        let* x = sample prob (gaussian ~mu:pre_x ~sigma:1.0) in
        let* () = observe prob (gaussian ~mu:x ~sigma:1.0) y in
        gen (x :: states) data
  in
  gen [] data

let _ =
  let data = Owl.Arr.linspace 0. 20. 20 ▷ Owl.Arr.to_array ▷ Array.to_list in
  let dist = Distribution.split_list (infer ~n:100 hmm data) in
  let m_x = List.map Distribution.mean dist in
  List.iter2 (Format.printf "%f >> %f") data m_x
```

HMM: Hidden Markov Model

```
❯ dune exec ./hmm.exe

0.000000 >> 0.000000
1.052632 >> 0.997546
2.105263 >> 2.300316
3.157895 >> 3.289649
4.210526 >> 4.857555
5.263158 >> 4.907179
6.315789 >> 6.254198
7.368421 >> 7.208341
8.421053 >> 8.432642
9.473684 >> 8.938143
10.526316 >> 9.555007
11.578947 >> 11.098199
12.631579 >> 12.823460
13.684211 >> 13.701444
14.736842 >> 14.934314
15.789474 >> 16.115058
...
```



Exercises

foo.ml

What if the particles do not terminate at the same time?

```
let foo () =
  let* c = sample (bernoulli ~p:0.5) in
  if c = 1 then
    let* () = factor 1. in return c
  else
    return c

let _ =
  let dist = infer foo () in
  let { values; probs; _ } = get_support ~shrink:true dist in
  Array.iteri (fun i x → Format.printf "%d %f@." x probs.(i)) values
```

Exercises

foo.ml

What if the particles do not terminate at the same time?

```
let foo () =
  let* c = sample (bernoulli ~p:0.5) in
  let* () = if c = 1 then factor 10. else skip in
  return c

let _ =
  let dist = infer foo () in
  let { values; probs; _ } = get_support ~shrink:true dist in
  Array.iteri (fun i x -> Format.printf "%d %f@." x probs.(i)) values
```

```
dune exec ./examples/foo.exe
Fatal error: exception Invalid_argument("option is None")
```

Exercise: Enumeration (CPS)

BYO-PPL

Enumeration

infer.ml

```
module : sig
  type 'a prob
  and 'a next = 'a prob → 'a prob
  and ('a, 'b) model = 'a → ('b → 'b next) → 'b next

  val sample : 'a Distribution.t → ('a → 'b next) → 'b next
  val factor : float → (unit → 'b next) → 'b next
  val infer : ('a, 'b) model → 'a → 'b Distribution.t
end = struct ... end
```

Inference algorithm: depth first exploration

- Maintain a continuation queue
- `sample`: pick one value (and score), push the other in the queue
- `factor`: update the score
- At the end of the model, store the pair (value, score), restart from the top of the queue

Inference comparison

BYO-PPL

Coin

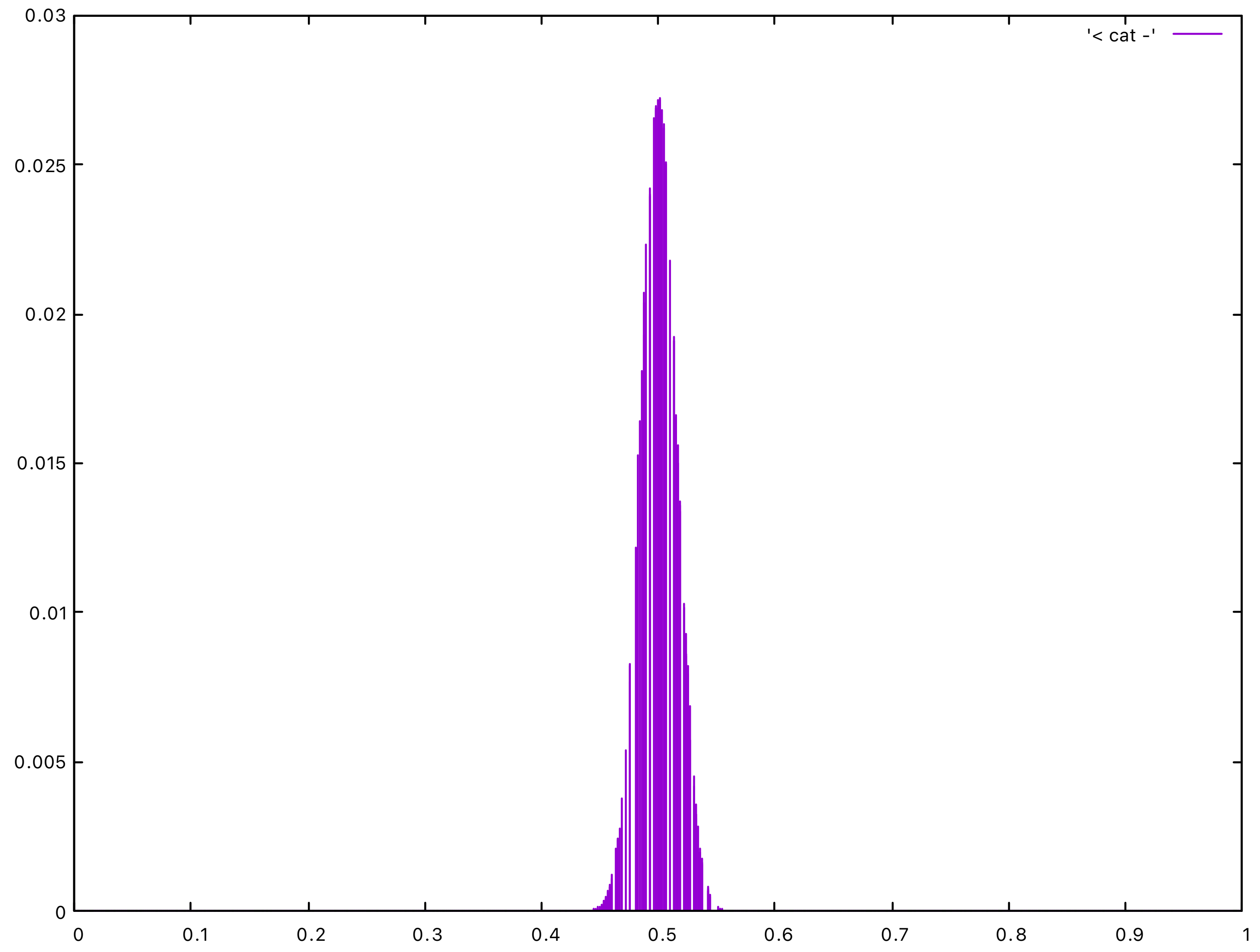
coin.ml

```
let _ =
  let data = List.init 1000 (fun i → i mod 2) in
  let dist = infer coin data in
  plot dist
```

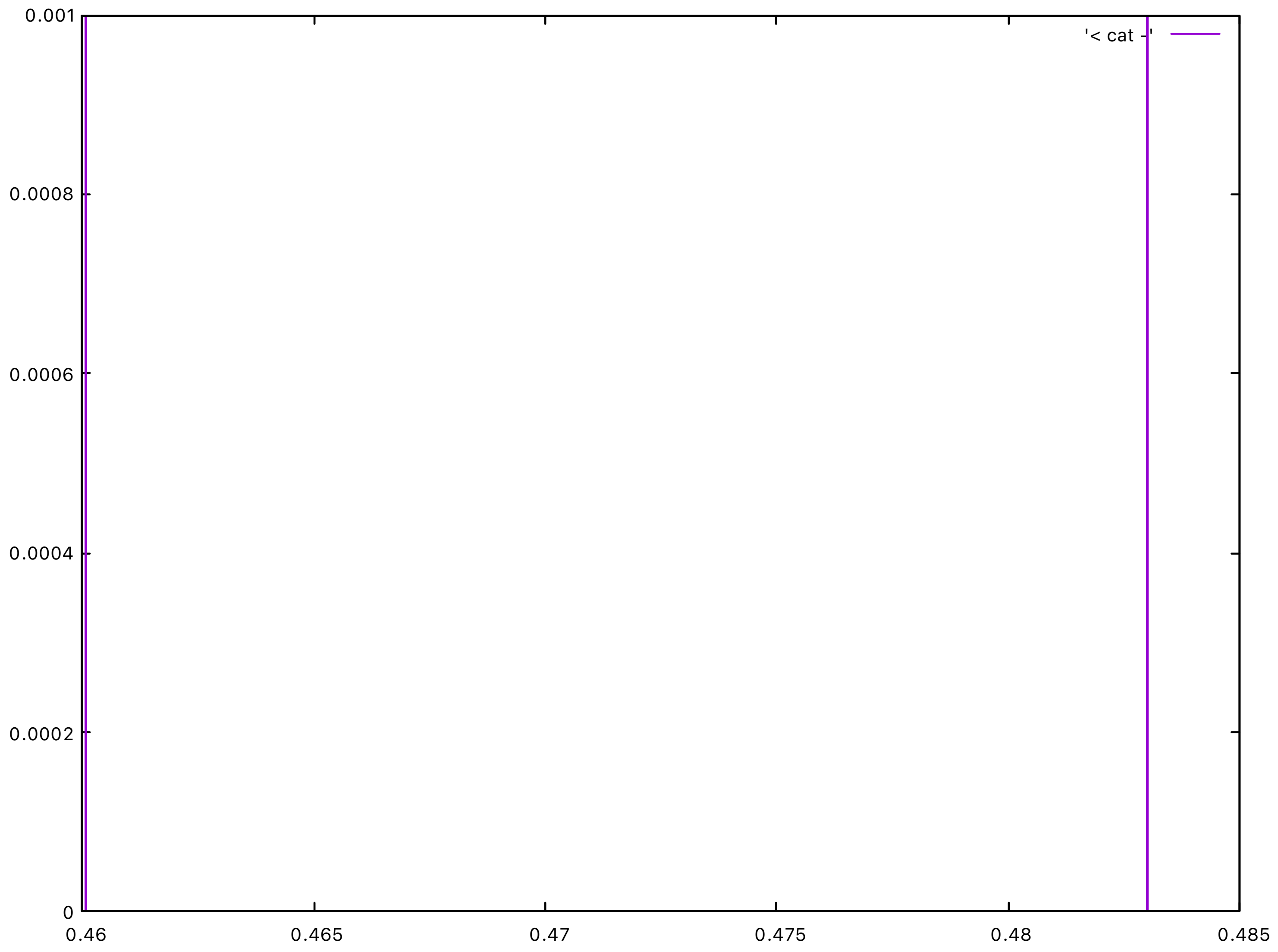
Rejection Sampling

Very (very) slow!

Importance Sampling



Particle Filter



Particle impoverishment

HMM

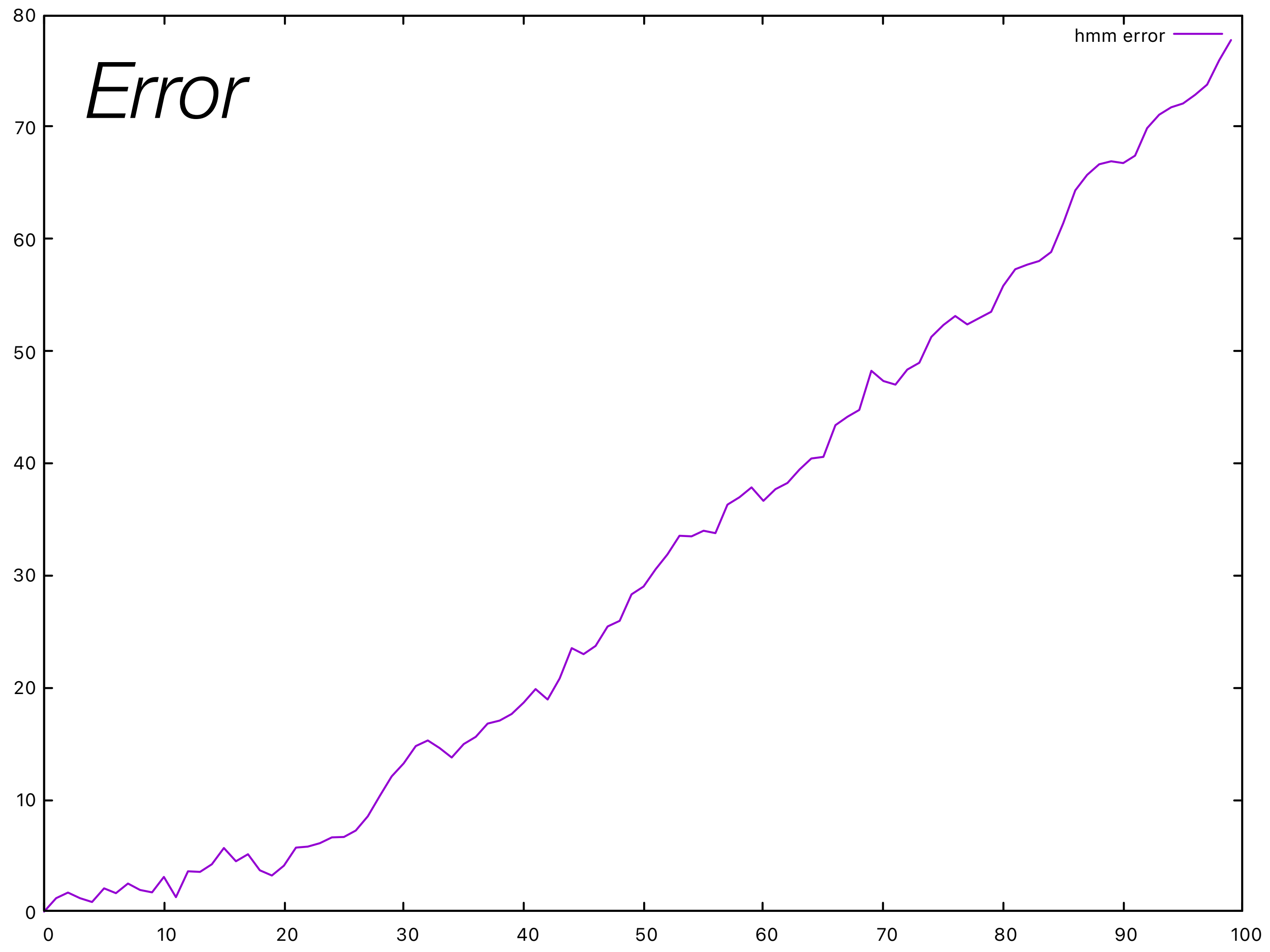
hmm.ml

```
let _ =
  let data = Owl.Arr.linspace 0. 100. 100 ▷ Owl.Arr.to_array ▷ Array.to_list in
  let dist = Distribution.split_list (infer hmm data) in
  plot (error (List.rev dist) data)
```

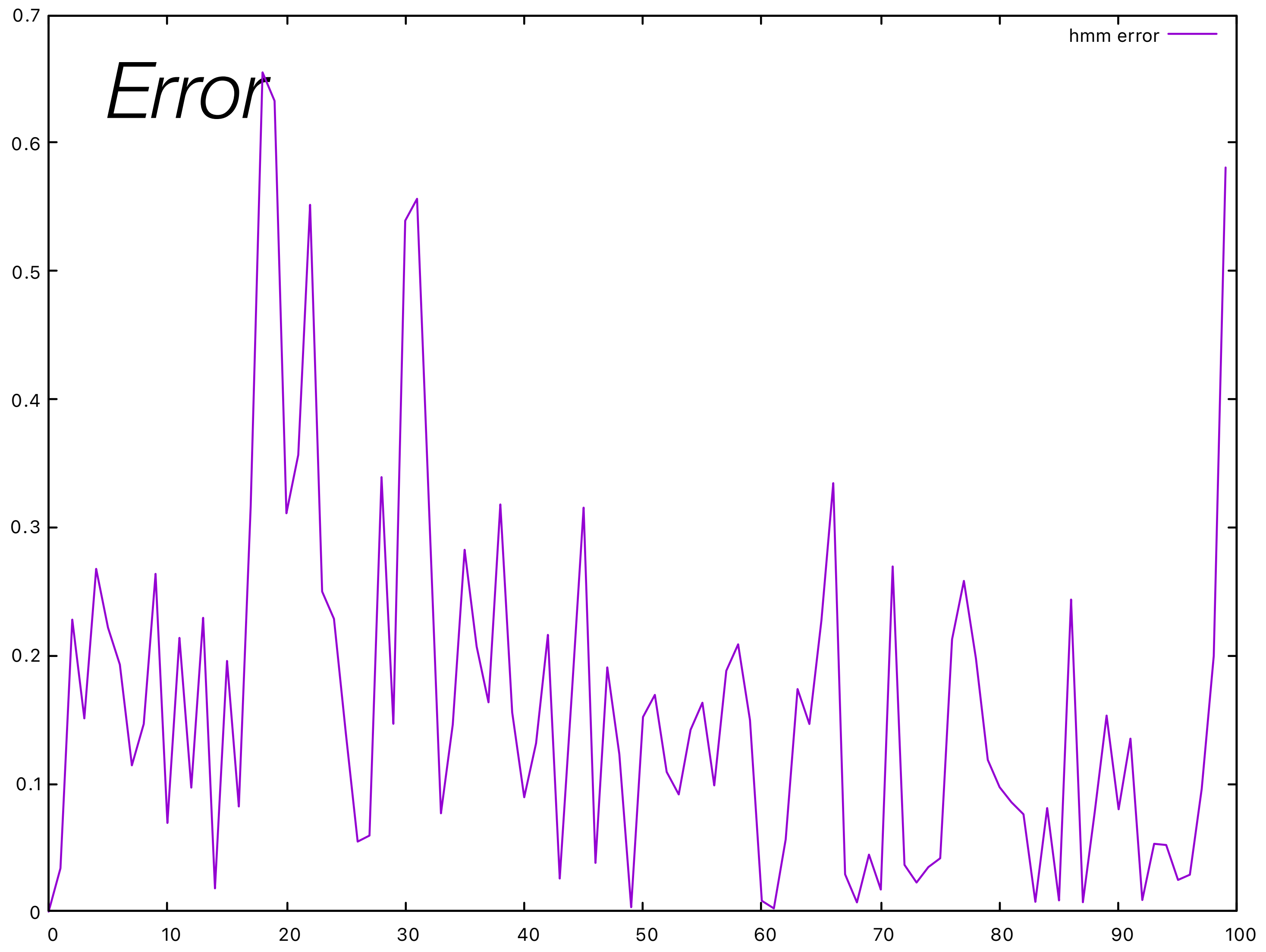
Rejection Sampling

Never terminate!

Importance Sampling



Particle Filter



Inference limitations

Rejection Sampling : Termination

- Only works if valid sample can be generated from the priors
- In practice: simple models with few discrete observations
- E.g., simple discrete models.

Importance Sampling : Weight collapse

- Score strictly decrease at each observation: eventually collapse
- In practice: continuous model to estimate fixed parameters from observations
- E.g., coin, linear regression

Particle Filter: Particle impoverishment

- Duplicate particle without resampling fixed parameter
- In practice: high-dimensional continuous models without fixed parameters
- E.g., hmm, tracker

Inference formalization

BYO-PPL

Sampler

Probabilistic semantics $G \vdash^P e : t$

- Expressions are interpreted as weighted samplers
- Draw random samples and track the execution weight
- Given an environment γ, $\{\!\![e]\!\!\}_\gamma = v, w$
- $\{\!\![e]\!\!\} : \Gamma \to V \times [0, \infty)$

$$
\begin{aligned}
\{\!\![c]\!\!\}_\gamma &= c, 1 \\
\{\!\![x]\!\!\}_\gamma &= \gamma(x), 1 \\
\{\!\![\texttt{sample}(e)]\!\!\}_\gamma &= \text{draw}([\![e]\!]_\gamma), 1 \\
\{\!\![\texttt{factor}(e)]\!\!\}_\gamma &= (\,), [\![e]\!]_\gamma \\
\{\!\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!\!\} &= \textit{let}\ v_1, w_1 = \{\!\![e_1]\!\!\}_\gamma\ \textit{in} \\
&\quad\ \textit{let}\ v_2, w_2 = \{\!\![e_2]\!\!\}_{\gamma + [x \leftarrow v_1]}\ \textit{in} \\
&\quad\ v_2, w_1 \times w_2
\end{aligned}
$$

Combine weights

Importance sampling

Inference algorithm

- Run a set of N independent executions
- `sample`: draw a sample from a distribution
- `factor`: associate a score to the current execution
- Gather output values and score to approximate the posterior distribution

$$
\begin{aligned}
[\![\texttt{infer}(e)]\!]_\gamma = \ & \textit{let } \left[v_i, w_i = \{\![e]\!\}_\gamma\right]_{1 \le i \le N} \textit{ in} \\
& \textit{let } W = \sum_{i=1}^{N} w_i \textit{ in} \\
& \lambda U.\ \sum_{i=1}^{N} \frac{w_i}{W} \times \delta_{v_i}(U)
\end{aligned}
$$

Particle filter

Inference algorithm : importance sampling, but...

- Add a resampling step at each `factor`
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution

Resampling

$$
\begin{aligned}
\{\!\!\{e\}\!\!\} &\propto \left\{\!\!\left\{\texttt{let}\ d = \texttt{infer}(e)\ \texttt{in}\ \texttt{sample}(d)\right\}\!\!\right\} \\
\left\{\!\!\left\{\texttt{let}\ x = e_1\ \texttt{in}\ e_2\right\}\!\!\right\} &\propto \left\{\!\!\left\{\texttt{let}\ d = \texttt{infer}(e_1)\ \texttt{in}\ \texttt{let}\ x = \texttt{sample}(d)\ \texttt{in}\ e_2\right\}\!\!\right\} \\
[\![\texttt{infer}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2)]\!] &= [\![\texttt{infer}(\texttt{let}\ d = \texttt{infer}(e_1)\ \texttt{in}\ \texttt{let}\ x = \texttt{sample}(d)\ \texttt{in}\ e_2)]\!]
\end{aligned}
$$

Checkpoints

BYO-PPL

Big-step semantics with checkpoints

resample flag

expression

$$e \to^{r} e', w$$

score

expression

Reduction rules: stop evaluation when r is true

Big-step semantics with checkpoints

$$\frac{e \to^{\textit{false}} \textit{true}, 1 \qquad e_1 \to^{r} e_1', w_1}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \to^{r} e_1', w_1}$$

$$\frac{e \to^{\textit{false}} \textit{false}, 1 \qquad e_2 \to^{r} e_2', w_2}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \to^{r} e_2', w_2}$$

$$\frac{e_1 \to^{\textit{true}} e_1', w_1}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \to^{\textit{true}} \texttt{let}\ x\ \texttt{=}\ e_1'\ \texttt{in}\ e_2, w_1}$$

$$\frac{e_1 \to^{\textit{false}} v_1, w_1 \qquad e_2 \to^{r} e_2', w_2}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \to^{r} e_2', w_1 \times w_2}$$

Combine weights

$$\frac{e \to^{\textit{false}} \mu, 1}{\texttt{sample}(e) \to^{\textit{true}} \mathsf{draw}(\mu), 1}$$

$$\frac{e \to^{\textit{false}} s, 1}{\texttt{factor}(e) \to^{\textit{true}} (), s}$$

Stop!

Particle filter

$$\frac{[e_i \to^{r_i} e'_i, w_i]_{1\leq i\leq N} \qquad \bigvee_{1\leq i\leq N} r_i \qquad \rho = \mathsf{Cat}\left(\{e'_i, w_i\}_{1\leq i\leq N}\right) \qquad [\mathsf{draw}(\rho)]_{1\leq i\leq N} \Rightarrow \mu}{[e_i]_{1\leq i\leq N} \Rightarrow \mu}$$

Resampling

$$\frac{\left[e_i \to^{\mathit{false}} v_i, w_i\right]_{1\leq i\leq N} \qquad W = \sum_{i=1}^{N} w_i}{[e_i]_{1\leq i\leq N} \Rightarrow \lambda U.\ \sum_{i=1}^{N} \frac{w_i}{W}\ \delta_{v_i}}$$

Gather the results

$$\frac{[e]_{1\leq i\leq N} \Rightarrow \mu}{e \Rightarrow^N \mu}$$

Launch N particles

References

WebPPL
Noah Goodman and Andreas Stuhlmüller
http://webppl.org/

The Design and Implementation of Probabilistic Programming Languages
Noah Goodman and Andreas Stuhlmüller
http://dippl.org/

An Introduction to Probabilistic Programming
Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, Frank Wood
https://arxiv.org/abs/1809.10756

Embedded probabilistic domain-specific language HANSEI
Oleg Kiselyov, Chung-chieh Shan
https://okmij.org/ftp/kakuritu/Hansei.html

BYO-PPL

Build Your Own Probabilistic Language

- Clone the repo: `git clone https://github.com/mpri-probprog/byo-ppl-22-23.git`
- Install the dependencies: `opam install . --deps-only`
- Build the project: `dune build`
- Test an example: `dune exec ./examples/funny_bernoulli.exe`

Implemented as an OCaml embedded domain specific language (eDSL)

- `Distribution`: small library of probability distributions and basic statistical functions.
- `Basic`: basic inference algorithms (rejection sampling inference sampling)
- `Infer`: inference algorithms for models written in Continuation Passing Style (CPS).
- `Cps_operators`: syntactic sugar to write CPS style probabilistic models.
- `Utils`: missing utilities functions used in other modules.