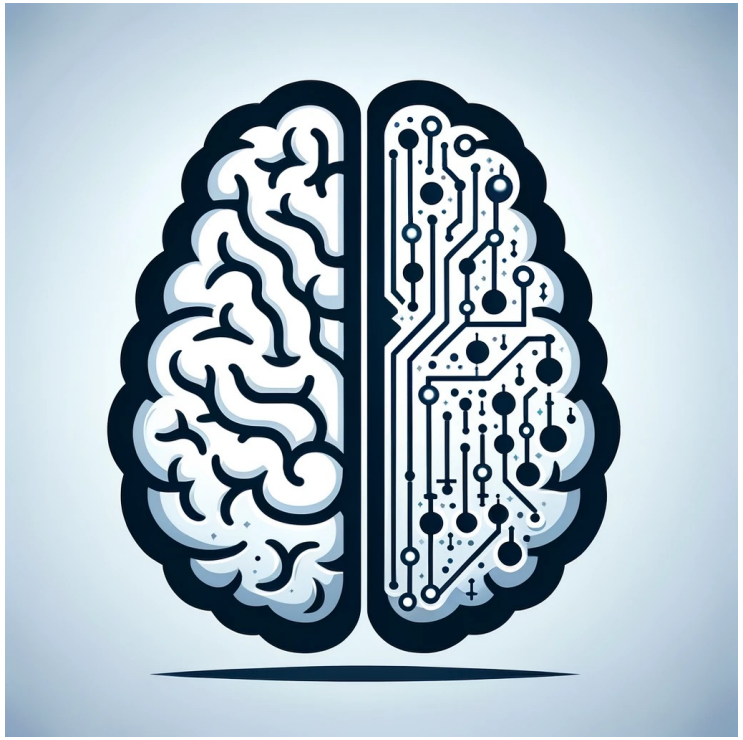
EN 601.473/601.673: Cognitive Artificial Intelligence (CogAI)



Lecture 14: Neural amortized inference

Tianmin Shu

Extending the Bayesian toolbox

- Neural networks
- Inference as optimization
- Neural amortized inference

Amortized inference

Gregory (1970)

Amortized inference

Gregory (1970)

Amortized inference

- Goal: $p(h|x)$
- Acquire / train a parameterized function $q(h|x;\phi)$ based on past experiences (i.e., training data)
- Training objective: KL-divergence between q and p, $\mathrm{D}_{KL}(q||p)$

$$\mathrm{D}_{KL}(q||p;\phi) = \sum_{h} q(h|x;\phi)\log(\frac{q(h|x;\phi)}{p(h|x)})$$

Connection to variational inference

- Goal: $p(h|x)$
- Approximate it with a family of distributions over the latent variables $q(h;\phi)$ with its own variational parameters ϕ
- Fit the parameters to train $q(h;\phi)$ as a proxy for the posterior

$$\log p(x) = \log \int_h p(x,h)dh$$
$$= \log \int_h p(x,h)\frac{q(h)}{q(h)}dh$$
$$= \log \int_h q(h)\frac{p(x,h)}{q(h)}dh$$
$$= \log\left(E_{h\sim q}\left[\frac{p(x,h)}{q(h)}\right]\right)$$
$$\geq E_{h\sim q}\left[\log\frac{p(x,h)}{q(h)}\right]$$

ELBO: evidence lower bound

$$E_{h\sim q}\left[\log\frac{p(x,h)}{q(h)}\right]$$

Maximize ELBO → maximize the lower bound of the marginalized probability p(x)

How is this related to KL-divergence?

$$f(\mathrm{E}[X]) \geq \mathrm{E}[f(X)]$$

Jensen's inequality

Connection to variational inference

- Goal: $p(h|x)$
- Approximate it with a family of distributions over the latent variables $q(h;\phi)$ with its own variational parameters ϕ
- Fit the parameters to train $q(h;\phi)$ as a proxy for the posterior

$$\mathrm{D}_{KL}(q(h)||p(h|x)) = \sum_h q(h)\log\left(\frac{q(h)}{p(h|x)}\right)$$

$$= E_{h\sim q}[\log(q(h))] - E_{h\sim q}[\log p(h|x)]$$

$$= E_{h\sim q}[\log(q(h))] - E_{h\sim q}[\log p(x,h)] + E_{h\sim q}[\log p(x)]$$

$$= -E_{h\sim q}\left[\log\left(\frac{p(x,h)}{q(h)}\right)\right] + \log p(x)$$

ELBO: evidence lower bound

$$E_{h\sim q}\left[\log\frac{p(x,h)}{q(h)}\right]$$

Minimize KL-divergence → maximize ELBO

Difference: p(x), which is what ELBO bounds

Amortized inference

- Goal: $p(h|x)$
- Acquire / train a parameterized function $q(h|x;\phi)$ based on past experiences (i.e., training data)
- Training objective: KL-divergence between q and p, $\mathrm{D}_{KL}(q||p)$

$$\mathrm{D}_{KL}(q||p;\phi) = \sum_{h} q(h|x;\phi)\log(\frac{q(h|x;\phi)}{p(h|x)})$$

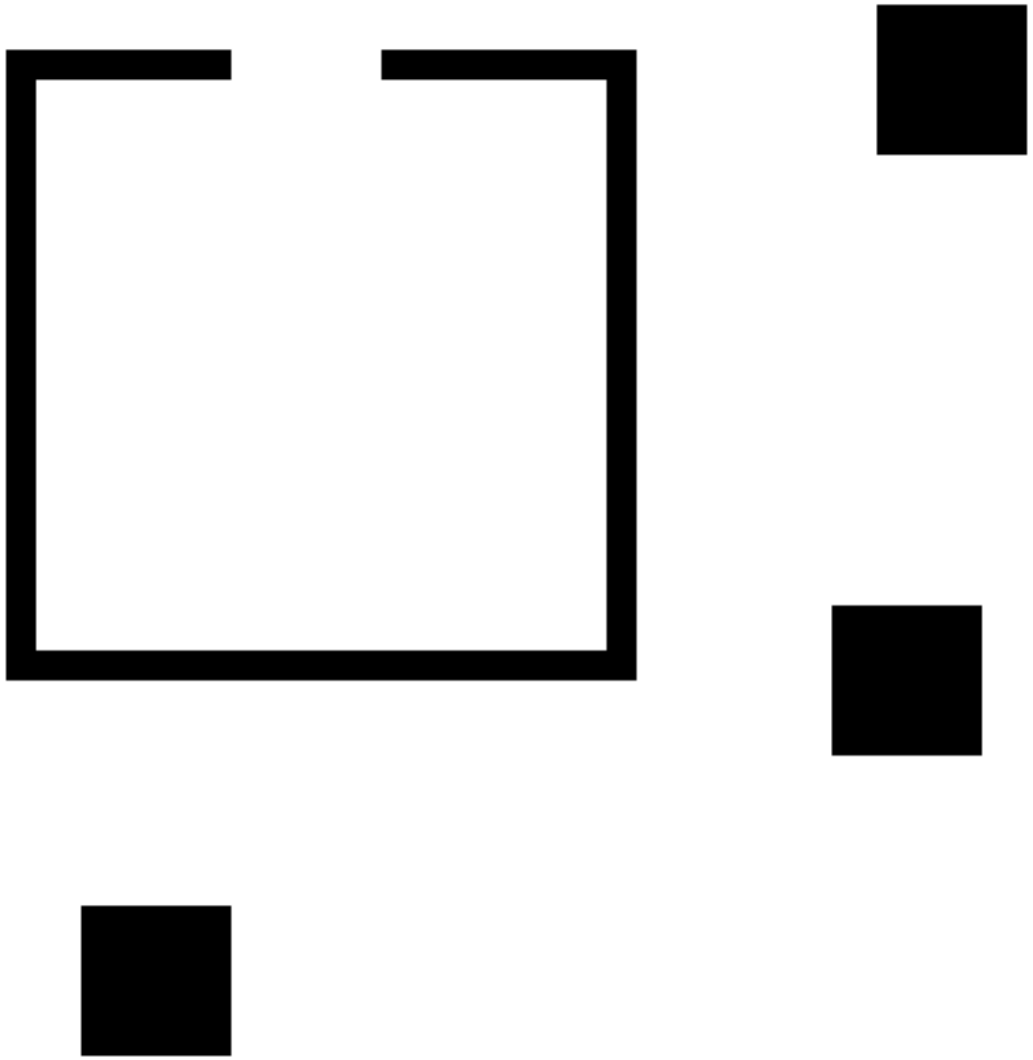
Neural amortized inference

- $q(h|x;\phi)$ as a neural network
- Train $q(h|x;\phi)$ to predict true posterior $\mathrm{p}(h|x)$ using labeled datasets $\{(x_i, p(h_i|x))\}$ by minimizing the KL-divergence

$$\phi = \mathrm{argmin}_{\phi}\, \mathrm{D}_{KL}(q||p;\phi)$$

- A simpler version: Train $q(h|x;\phi)$ to predict latent variables h based on inputs x using labeled datasets $\{(x_i, h_i)\}$
- Use $q(h|x;\phi)$ as a data-driven proposal (importance sampling, MCMC, SMC)

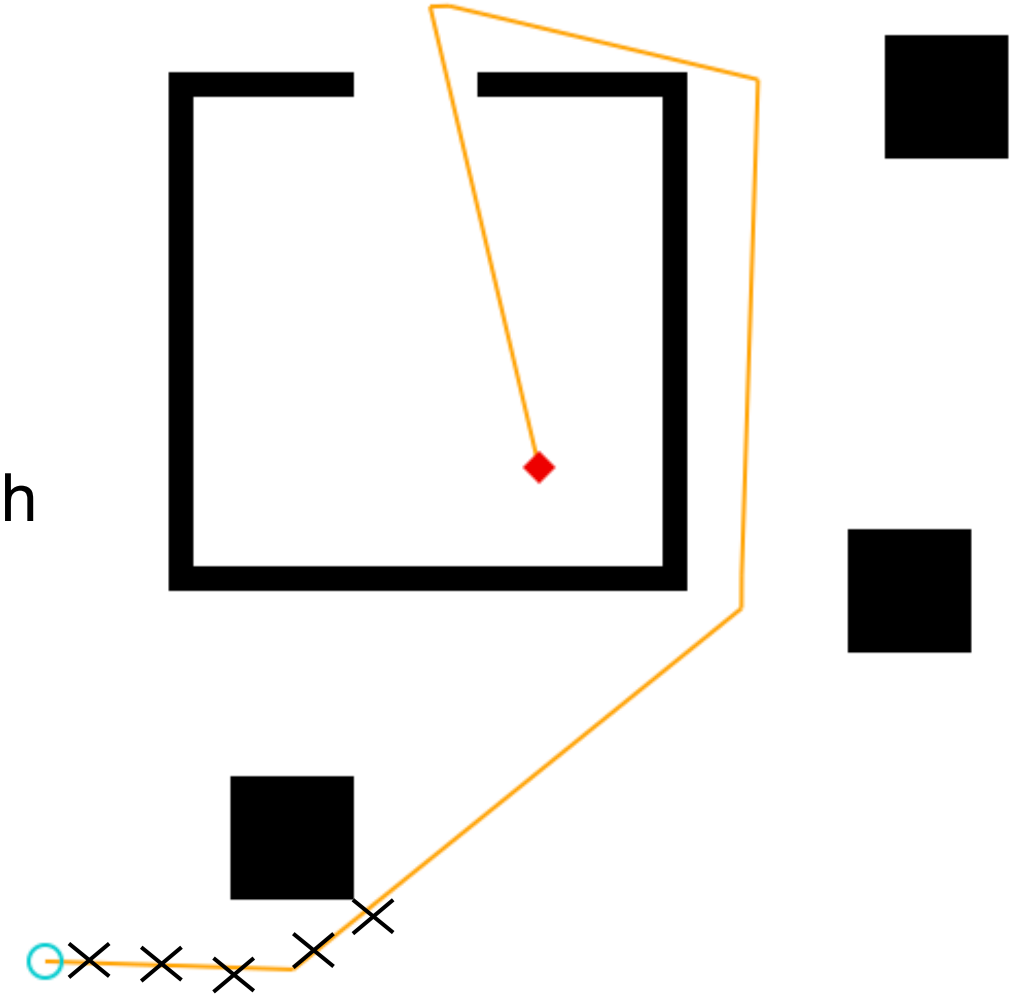
Example in Gen: goal inference



Example in Gen: goal inference

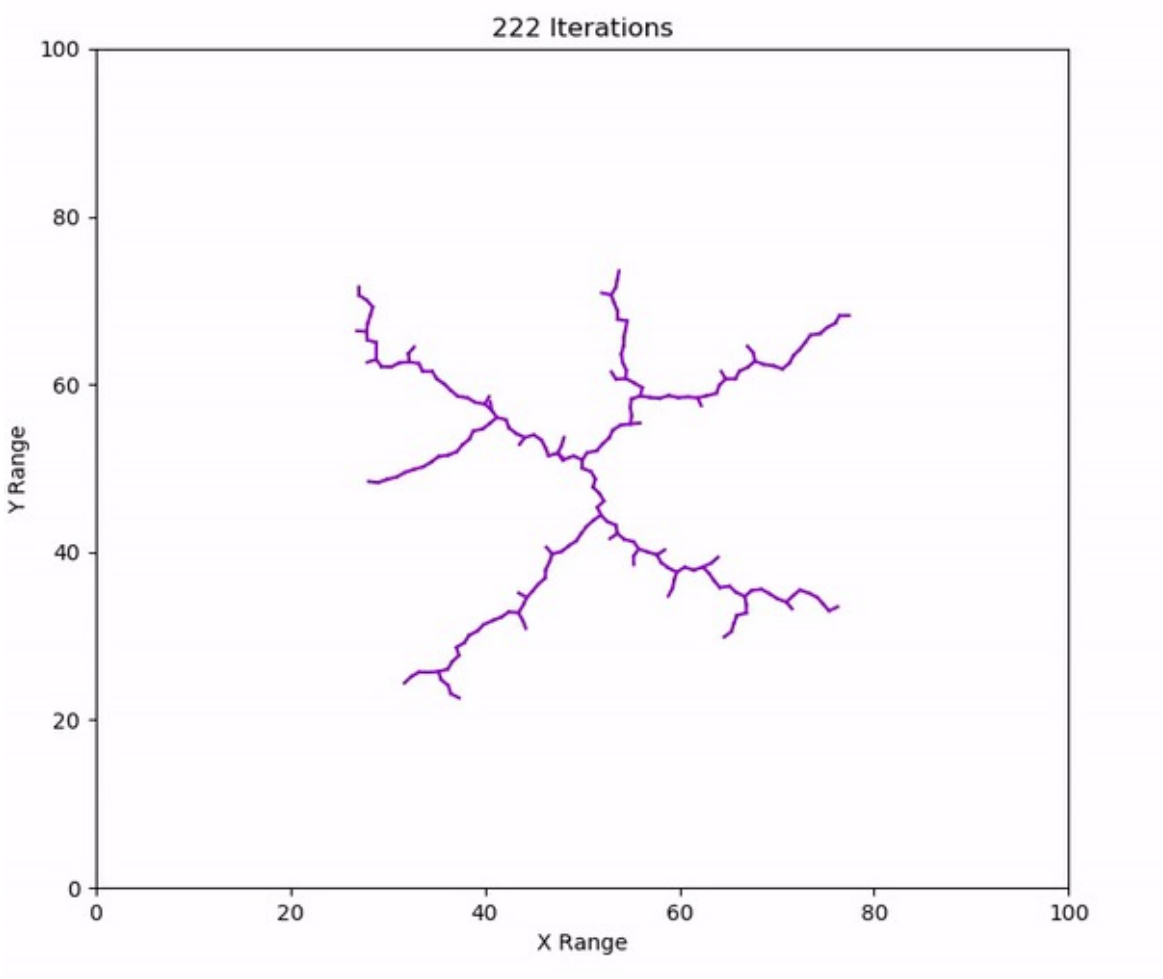
Generative agent model

Given the initial and goal positions,
1) Plan a path to reach the goal
2) Sample velocity
3) Sample observed trajectory alone the path



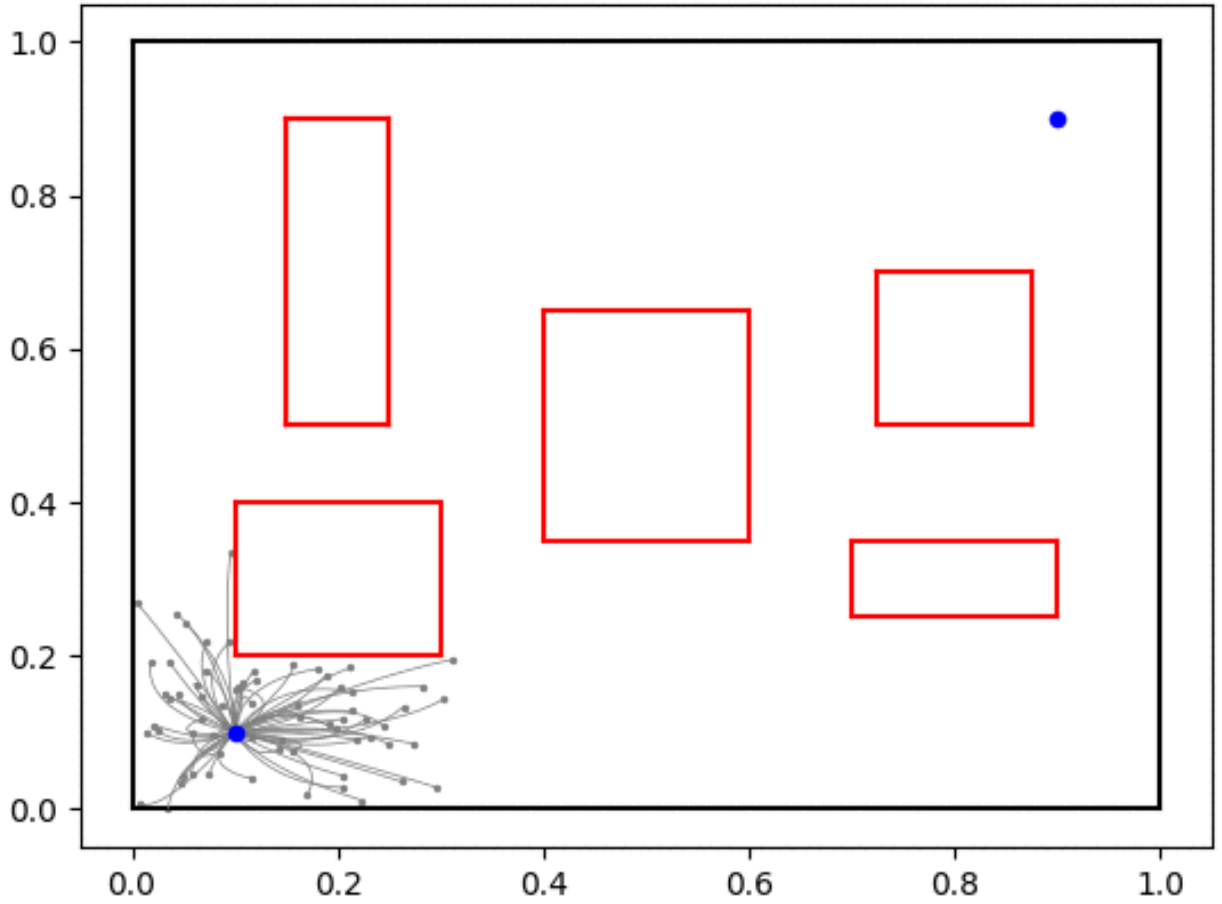
Rapidly exploring random tree (RRT)

- Starting from initial location, iteratively sample possible paths on a tree



Rapidly exploring random tree (RRT)

- Until the path reaches the target location => return the shortest path

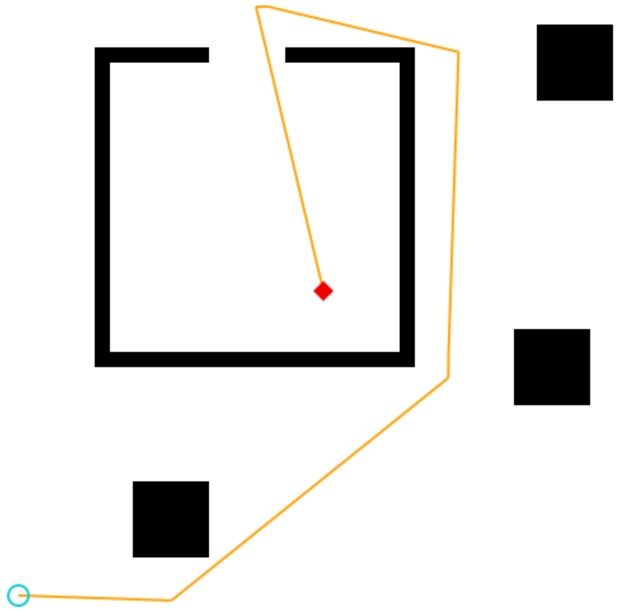


Rapidly exploring random tree (RRT)

Path: locations of way points

```
start = Point(0.1, 0.1)
dest = Point(0.5, 0.5)
planner_params = PlannerParams(rrt_iters=600, rrt_dt=3.0,
                               refine_iters=3500, refine_std=1.)
example_path = plan_path(start, dest, scene, planner_params)
```

```
Path(Point[Point(0.1, 0.1), Point(0.3006965602111723, 0.09344727126751506), Point(0.6639161596469678, 0.3859075491423317), Point(0.6640677335417938, 0.4132015144820388), Point(0.6773131251186395, 0.81394227709134), Point(0.42683497691435557, 0.8736413366304225), Point(0.4117137517524211, 0.8731459056920481), Point(0.5, 0.5)])
```



Generate a walk path for a certain length

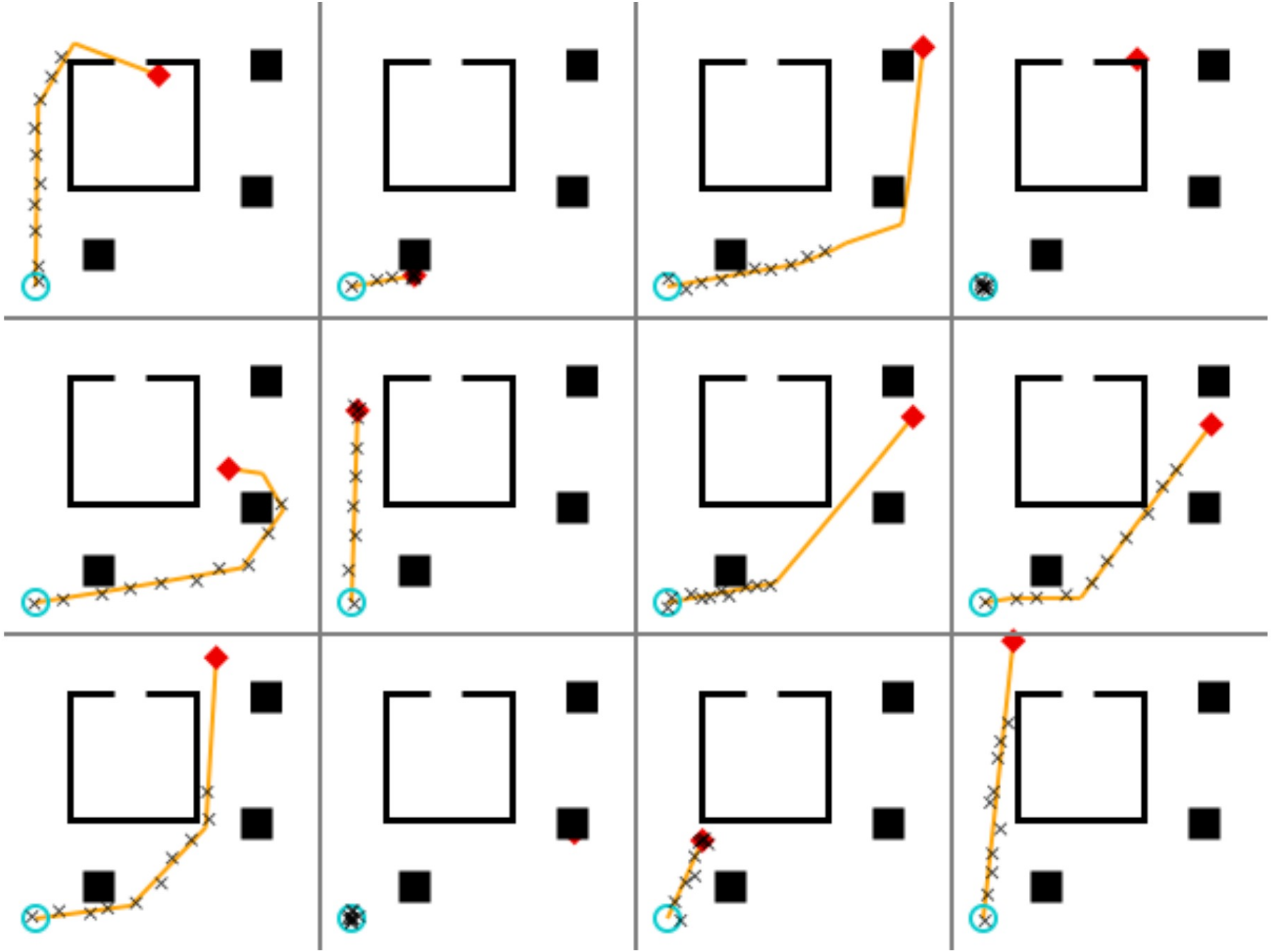
Walk path: location at each time step when moving alone the path at a constant speed
Speed: velocity
dt: time between two time steps
num_ticks: number of time steps

```
speed = 1.
dt = 0.1
num_ticks = 10;
example_locations = walk_path(example_path, speed, dt, num_ticks)
println(example_locations)
```

```
Point[Point(0.1, 0.1), Point(0.19994674167904422, 0.0967367458354572), Point(0.29989348335808846, 0.093473491670914
4), Point(0.37796006809990357, 0.15565896640131716), Point(0.45584942066446366, 0.21837458452849218), Point(0.53373
87732290239, 0.2810902026556672), Point(0.6116281257935839, 0.3438058207828422), Point(0.6642518809927379, 0.418772
9156366043), Point(0.6675553039918318, 0.5187183377253075), Point(0.6708587269909259, 0.6186637598140108)]
```

Generative agent model in Gen

```
@gen function agent_model(
        scene::Scene, dt::Float64, num_ticks::Int,
        planner_params::PlannerParams)

    # sample the start point of the agent from the prior
    start_x ~ uniform(0, 1)
    start_y ~ uniform(0, 1)
    start = Point(start_x, start_y)

    # sample the destination point of the agent from the prior
    dest_x ~ uniform(0, 1)
    dest_y ~ uniform(0, 1)
    dest = Point(dest_x, dest_y)

    # plan a path that avoids obstacles in the scene
    maybe_path = plan_path(start, dest, scene, planner_params)
    planning_failed = maybe_path === nothing

    # sample the speed from the prior
    speed ~ uniform(0.3, 1)

    if planning_failed
        # path planning failed; assume agent stays at start location indefinitely
        locations = fill(start, num_ticks)
    else
        # path planning succeeded; move along the path at constant speed
        locations = walk_path(maybe_path, speed, dt, num_ticks)
    end

    # generate noisy measurements of the agent's location at each time point
    noise = 0.01
    for (i, point) in enumerate(locations)
        x = {:meas => (i, :x)} ~ normal(point.x, noise)
        y = {:meas => (i, :y)} ~ normal(point.y, noise)
    end

    return (planning_failed, maybe_path)
end;
```

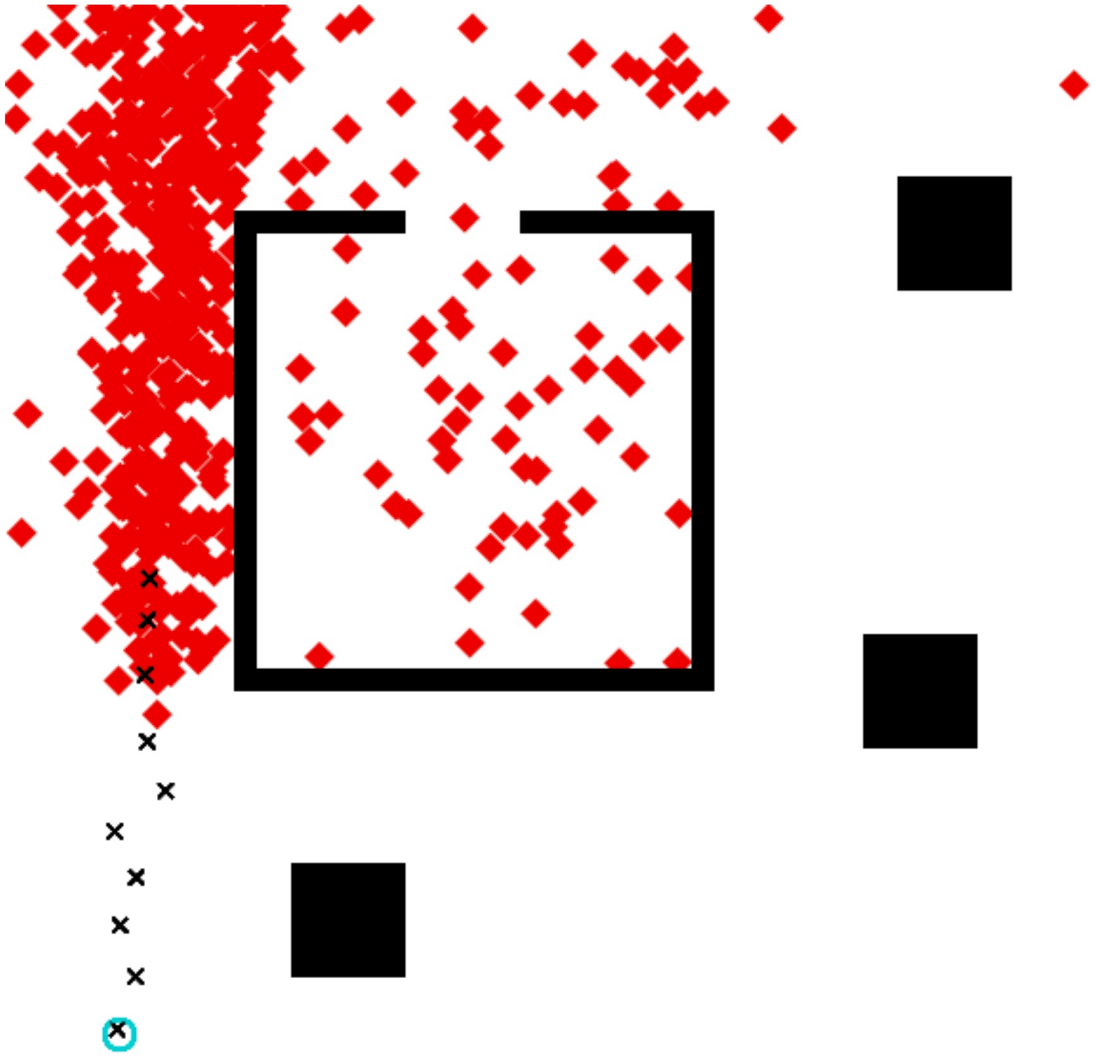


Inference using importance sampling

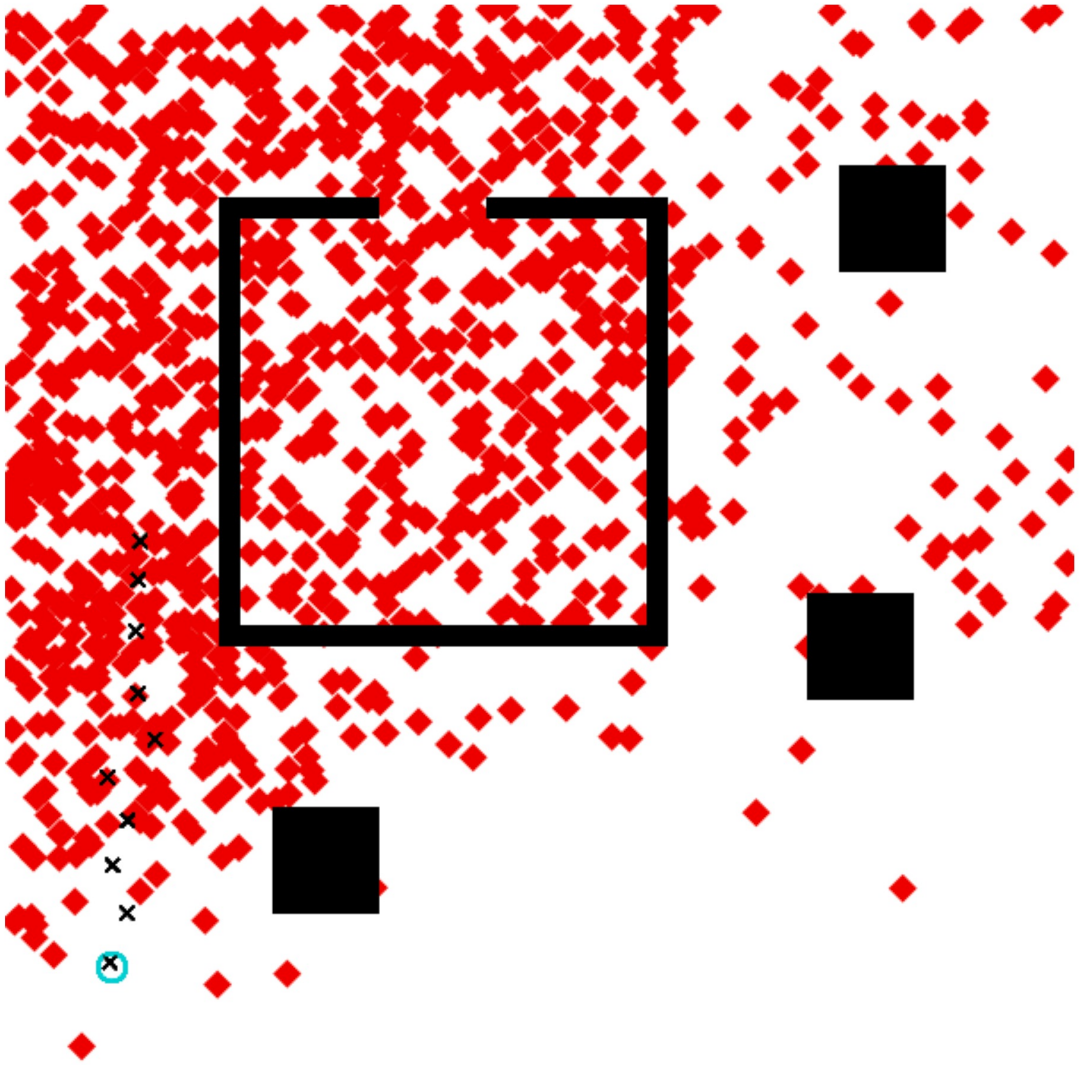
```
function do_inference(
        scene::Scene, dt::Float64, num_ticks::Int,
        planner_params::PlannerParams,
        start::Point, measurements::Vector{Point}, amount_of_computation::Int)

    # Constrain the observed measurements.
    observations = Gen.choicemap()
    observations[:start_x] = start.x
    observations[:start_y] = start.y
    for (i, m) in enumerate(measurements)
        observations[:meas => (i, :x)] = m.x
        observations[:meas => (i, :y)] = m.y
    end

     (trace, _) = Gen.importance_resampling(agent_model, (scene, dt, num_ticks, planner_params),
                    observations, amount_of_computation)

    return trace
end;
```

Results

(~ 100 sec)
The amount of compute: 100



(~ 5 sec)
The amount of compute: 5



How to use less compute to produce good inference?

- q(goal | observed trajectory)
  - q(goal_x | start_x, start_y, end_x, end_y)
  - q(goal_y | start_x, start_y, end_x, end_y, goal_x)

- Train both proposals (two NNs) using data sampled from the generative agent model

Importance sampling with customized proposal

```
function do_inference_data_driven(
        dest_proposal::GenerativeFunction,
        scene::Scene, dt::Float64,
        num_ticks::Int, planner_params::PlannerParams,
        start::Point, measurements::Vector{Point},
        amount_of_computation::Int)

    observations = Gen.choicemap((:start_x, start.x), (:start_y, start.y))
    for (i, m) in enumerate(measurements)
        observations[:meas => (i, :x)] = m.x
        observations[:meas => (i, :y)] = m.y
    end

    (trace, _) = Gen.importance_resampling(agent_model, (scene, dt, num_ticks, planner_params),
                    observations, dest_proposal, (measurements, scene), amount_of_computation)

    return trace
end;
```

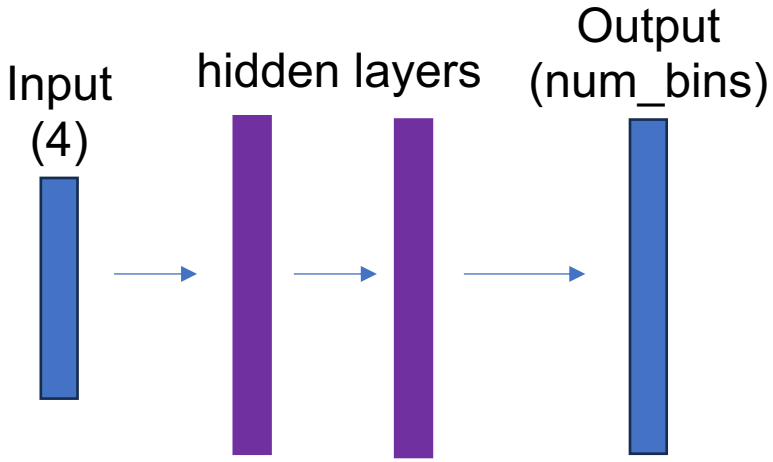
Neural network implementations in Gen

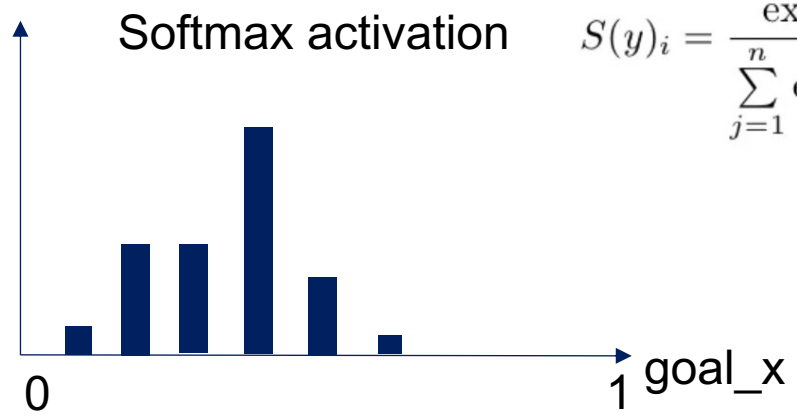
- Two options:
- Native Gen implementation
- PyTorch (GenPyTorch integrates PyTorch with Gen)

Native Gen neural network implementations

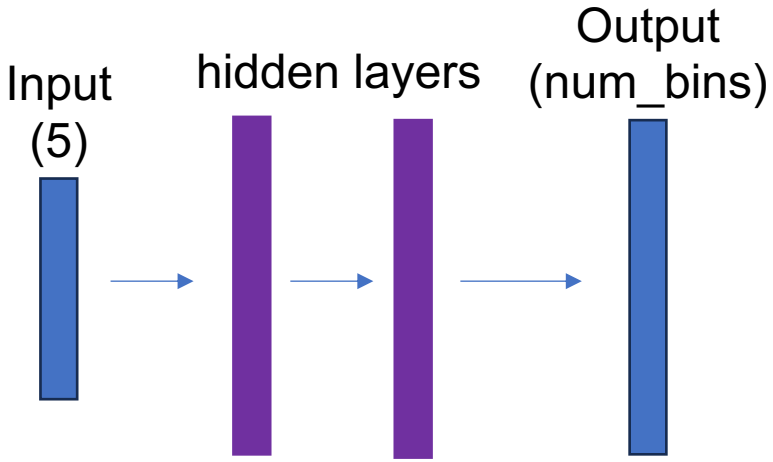
- q(goal_x | start_x, start_y, end_x, end_y)

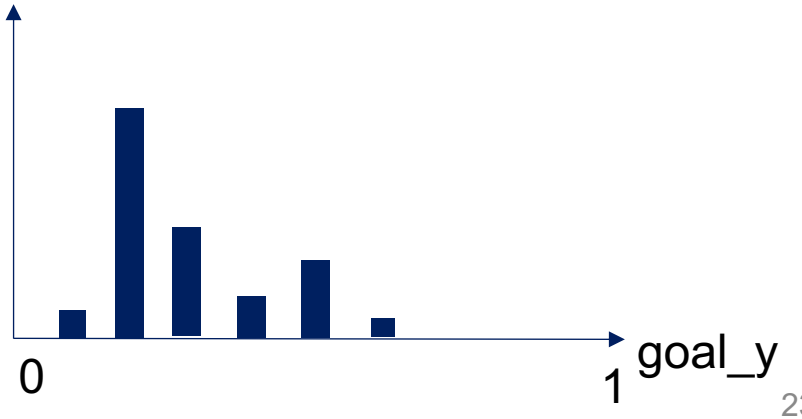
See jupyter notebook



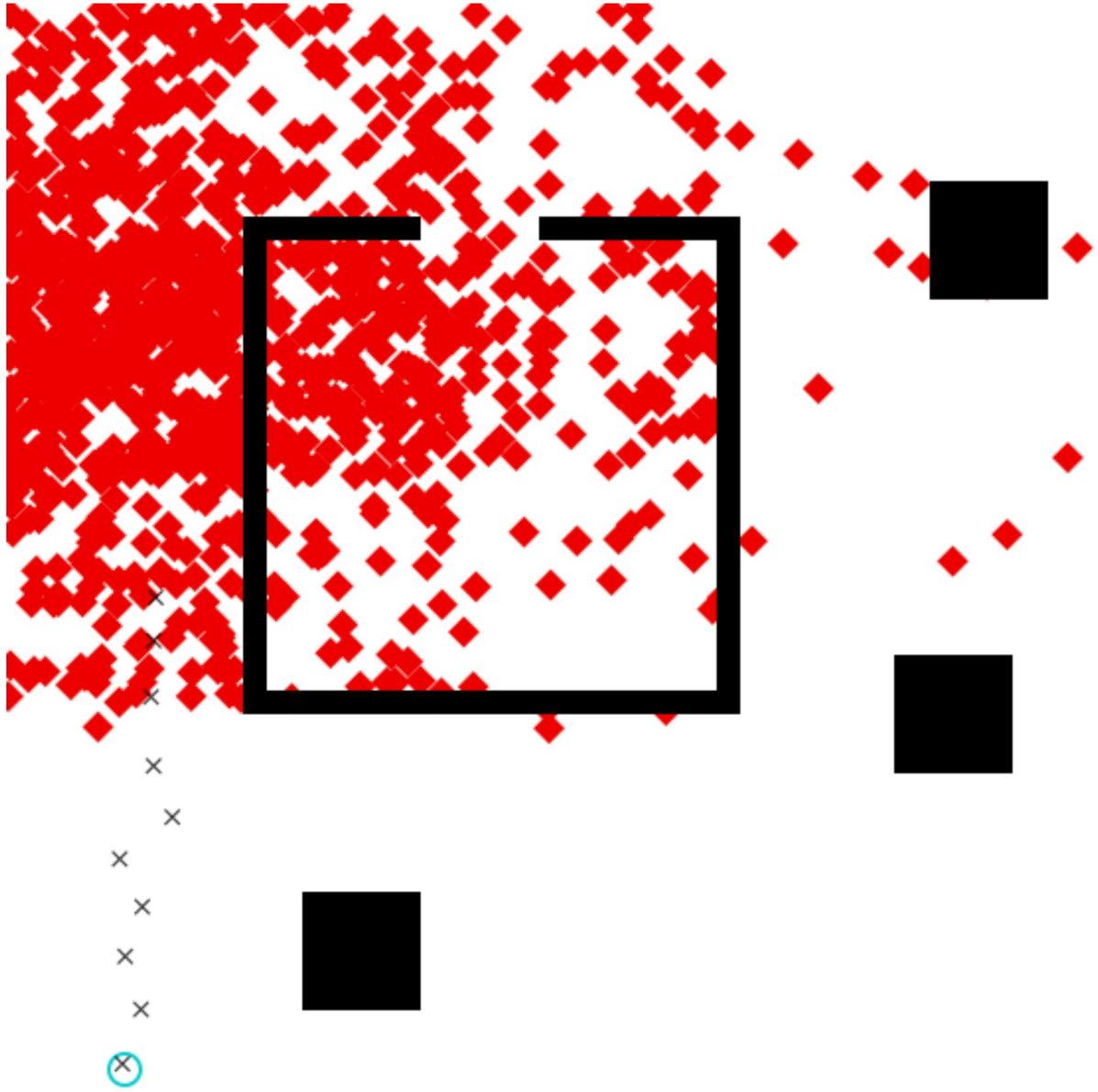


- q(goal_y | start_x, start_y, end_x, end_y, goal_x)





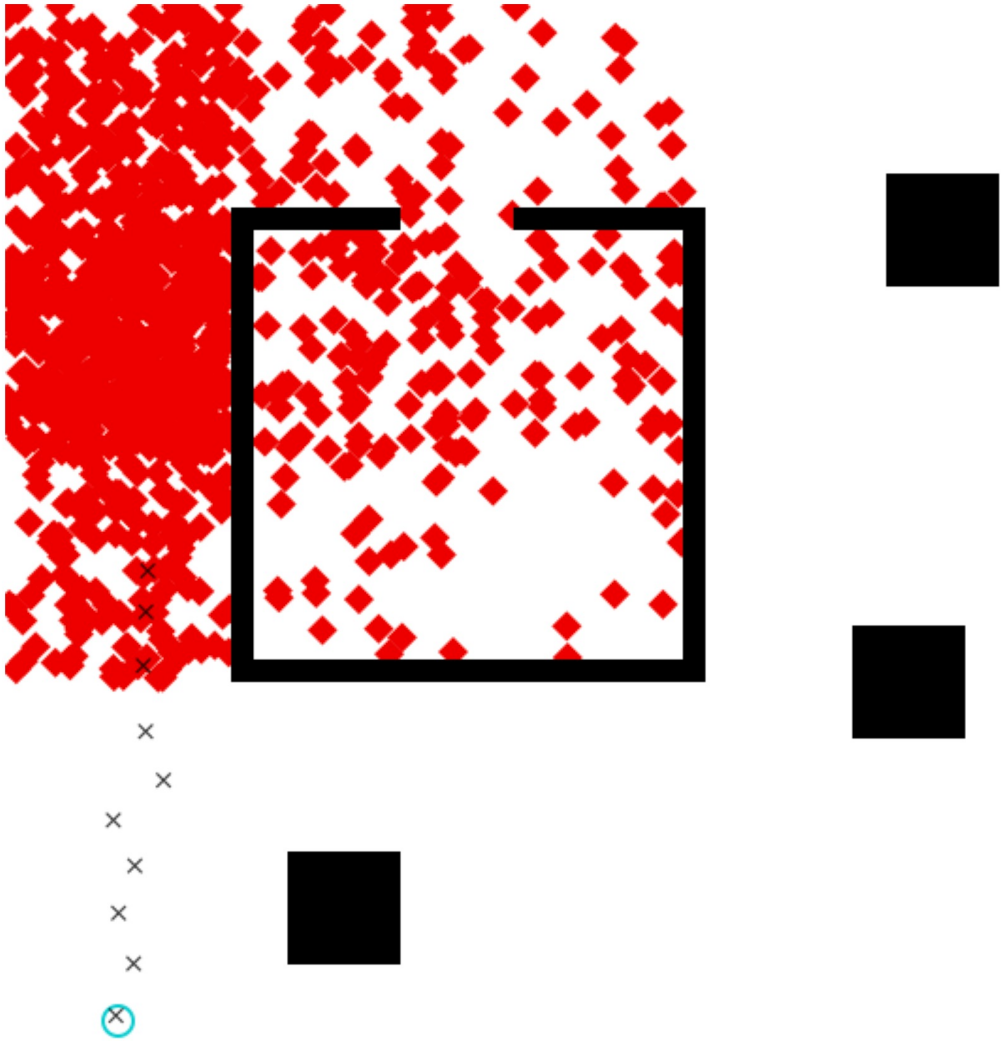
Sampled NN proposals



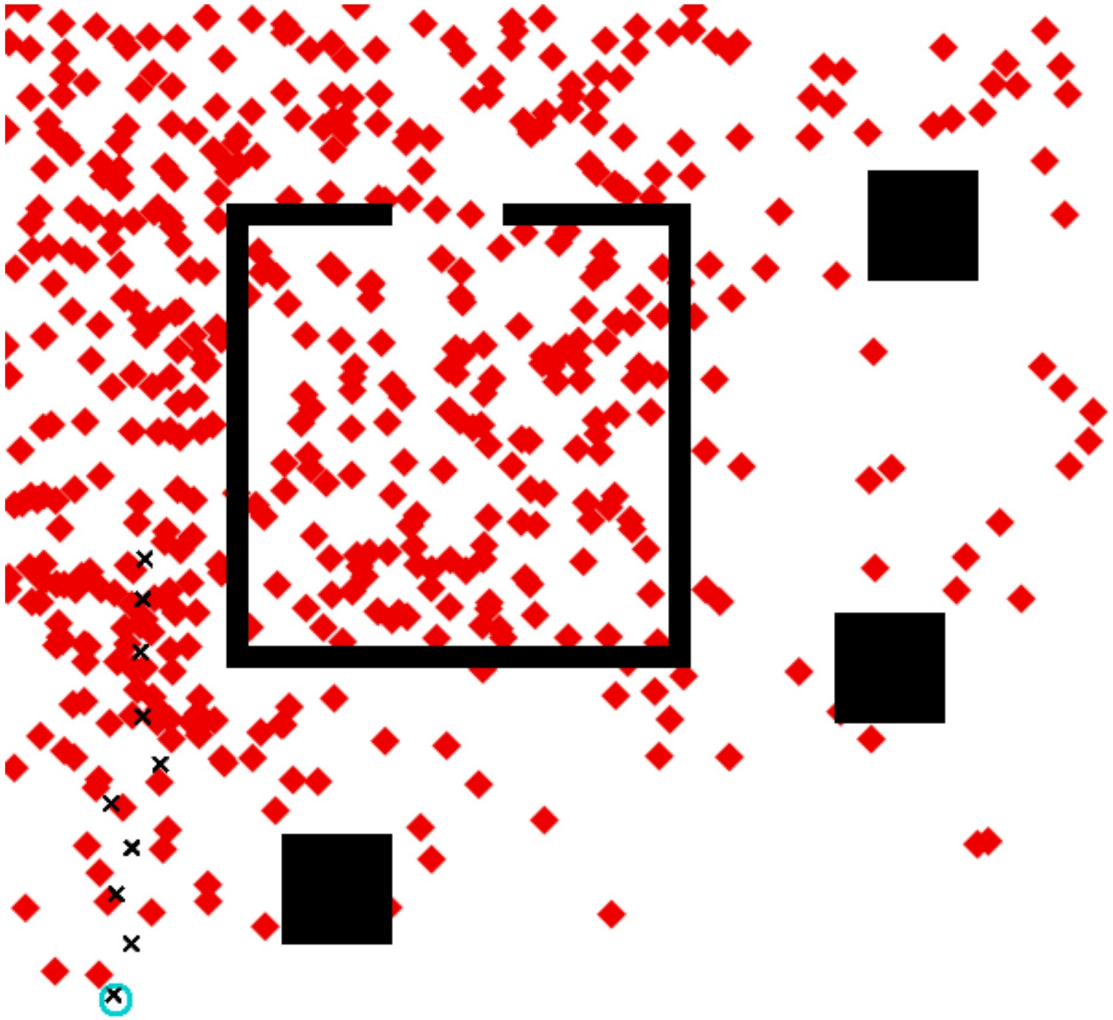
Comparison

The amount of compute: 5

Neural amortized inference



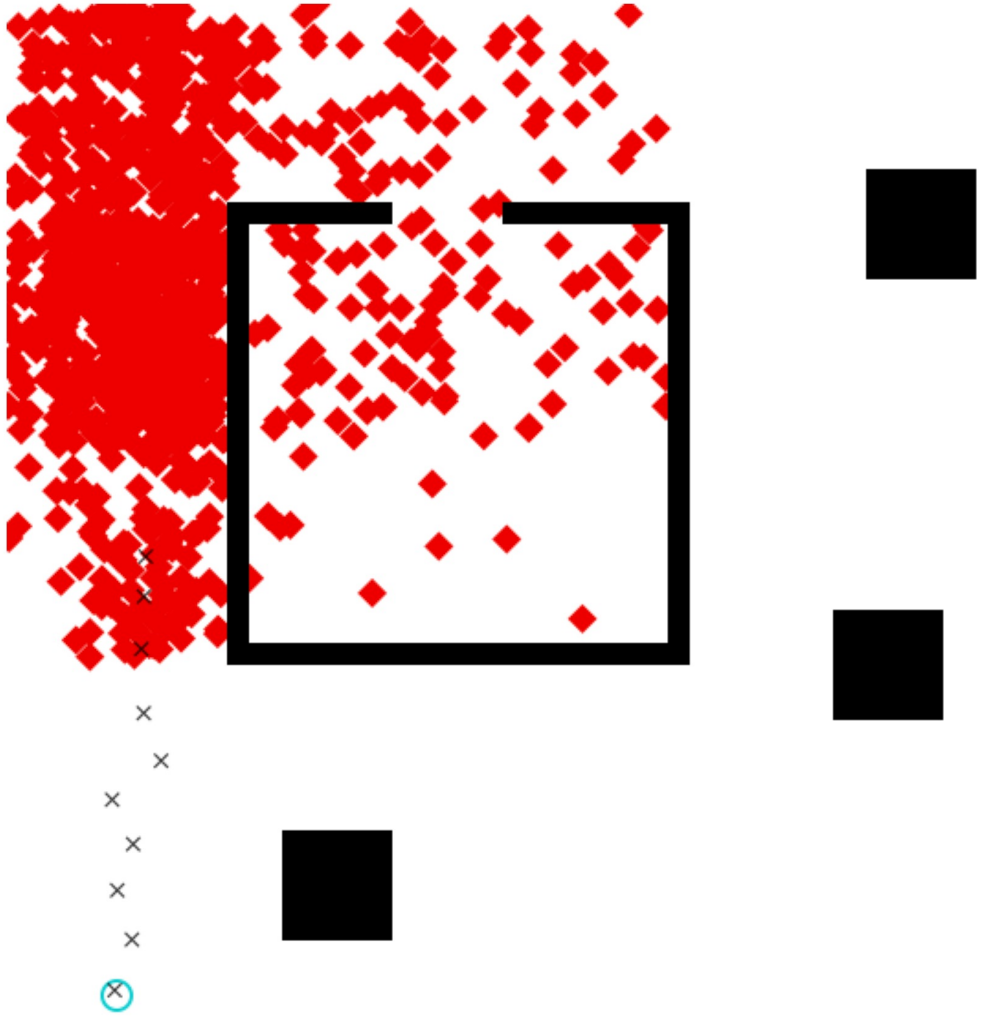
Built-in importance sampling



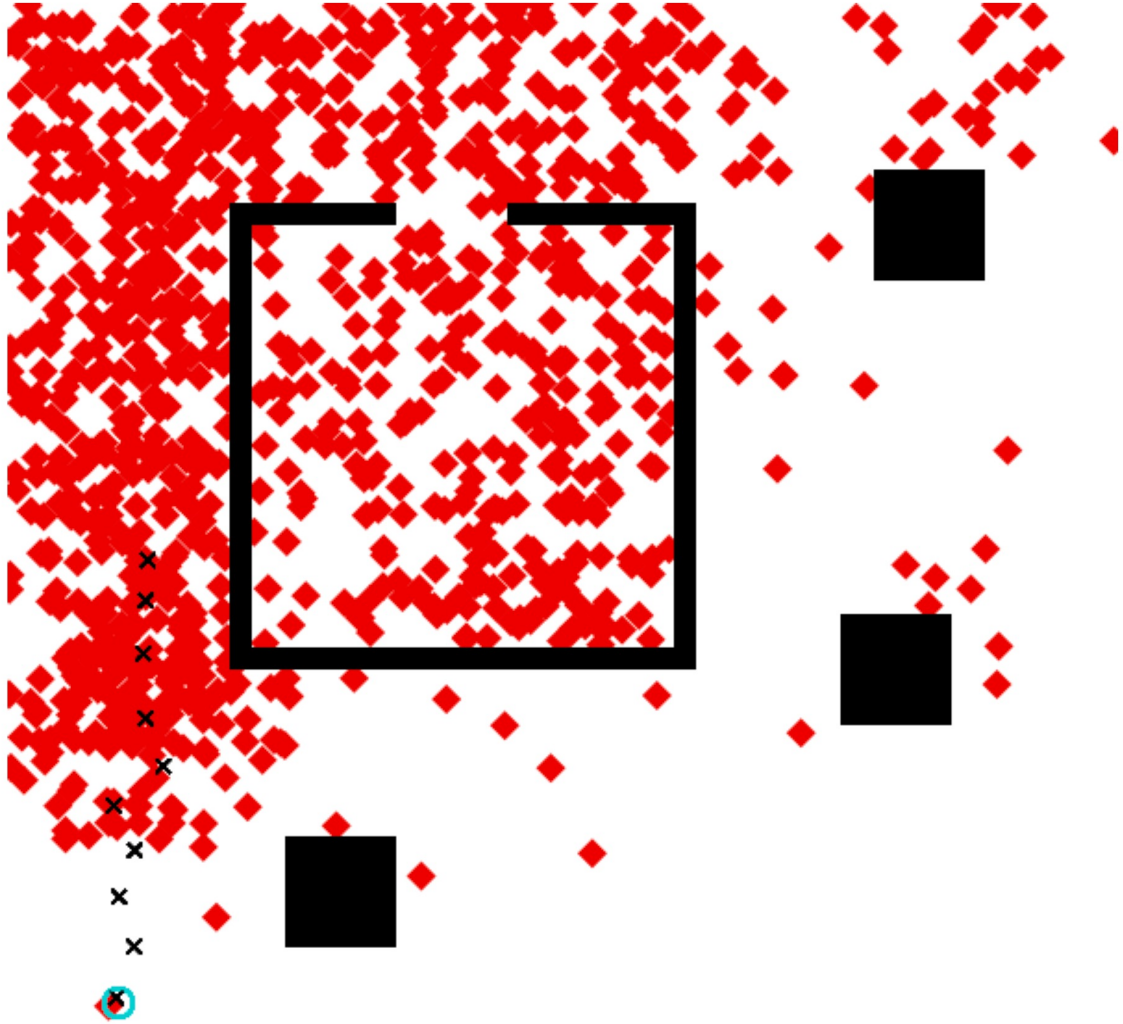
Comparison

The amount of compute: 10

Neural amortized inference



Built-in importance sampling



PyTorch integration

- PyCall: call python functions, import python packages
- GenPyTorch: write Gen functions using NNs implemented in PyTorch

See jupyter notebook