Gen: a probabilistic programming platform for probabilistic AI

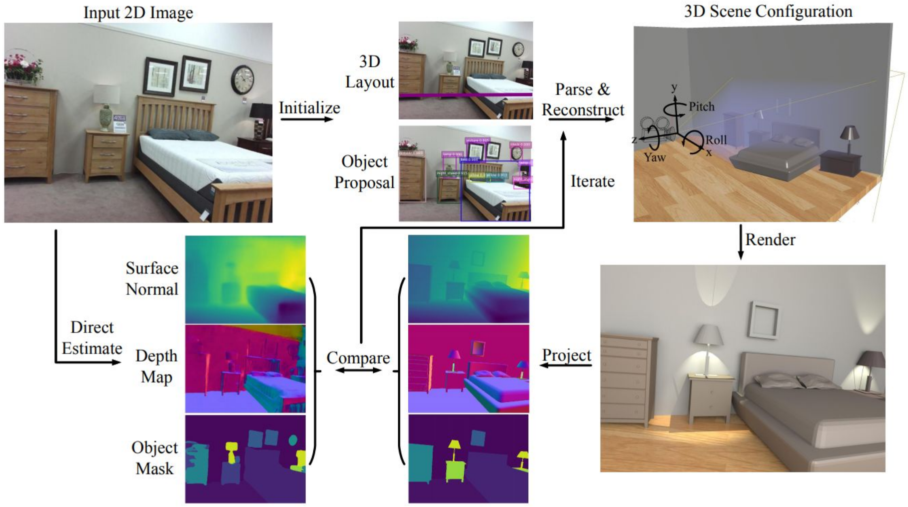
Marco Cusumano-Towner, Vikash Mansinghka

MIT Probabilistic Computing Project

Outline

- Motivation: visual scene understanding
- Example 1: Robust regression via optimization and MCMC
- Key technical idea: Gen Modules
- Example 2: 3D body pose inference using graphics engines, deep learning and Monte Carlo

Example domain: scene understanding



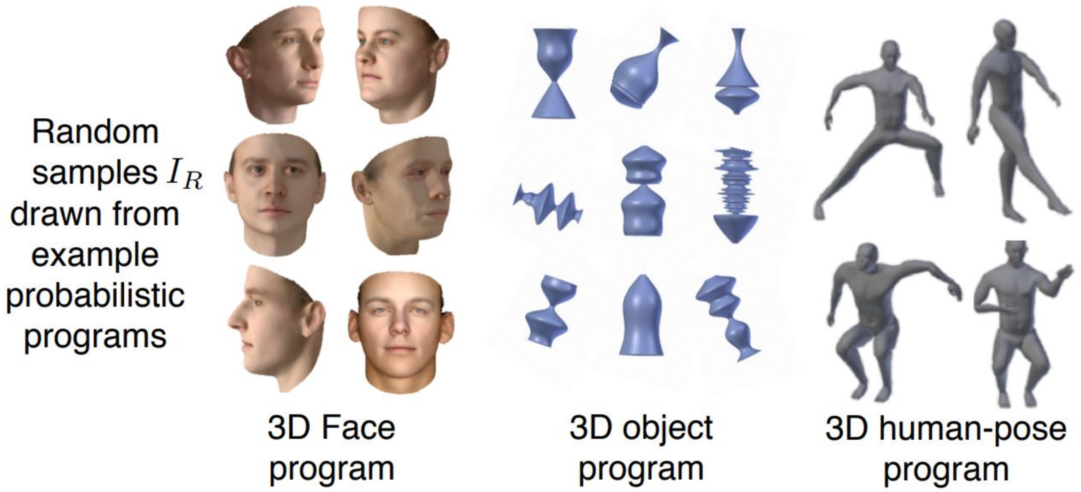
4 Inference

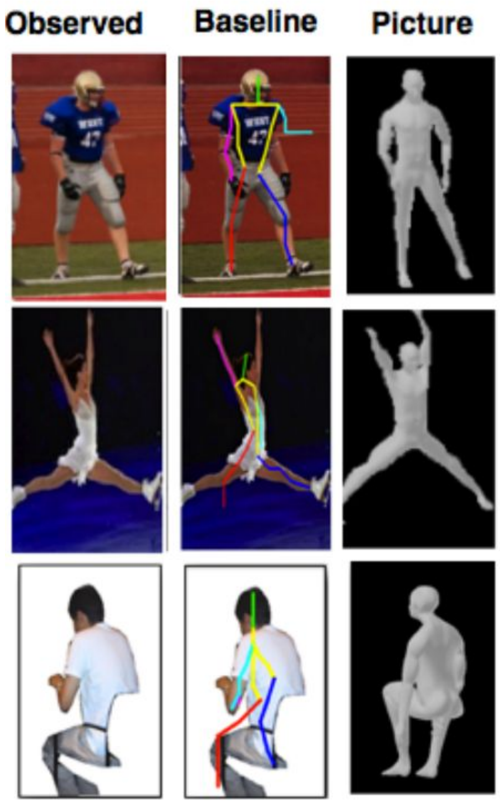
Given a single RGB image as the input, the goal of inference is to find the optimal **pg** that best explains the hidden factors that generate the observed image while recovering the 3D scene structure. The inference includes three major steps:

- *Room geometry estimation:* estimate the room geometry by predicting the 2D room layout and the camera parameter, and by projecting the estimated 2D layout to 3D. Details are provided in subsection 4.1.
- *Objects initialization:* detect objects and retrieve CAD models correspondingly with the most similar appearance, then roughly estimate their 3D poses, positions, sizes, and initialize the support relations. See subsection 4.2.
- *Joint inference:* optimize the objects, layout and hidden human context in the 3D scene in an analysis-by-synthesis fashion by maximizing the posterior probability of the **pg**. Details are provided in subsection 4.3.

Huang et al. "Holistic 3D Scene Parsing and Reconstruction from a Single RGB Image." arXiv (2018).

Picture





Kulkarni, et al. "Picture: A probabilistic programming language for scene perception." CVPR. 2015.

Design goals

Modeling and inference from multiple paradigms

Bayesian networks, Markov random fields, graphics/physics engines, deep neural network models

Monte Carlo inference, deep inference networks, numerical optimization

Programmable inference, not black-box

"Use Gibbs sampling to update X|Y, then optimize Y|X"

Advanced techniques, e.g. reversible jump and particle MCMC

Custom MCMC/SMC proposals, without requiring users to derive proposal densities and Jacobians

Easy to combine built-in algorithms with arbitrary user-specified inference code

Fast enough for real-time applications

Out-of-the-box performance competitive with handwritten samplers

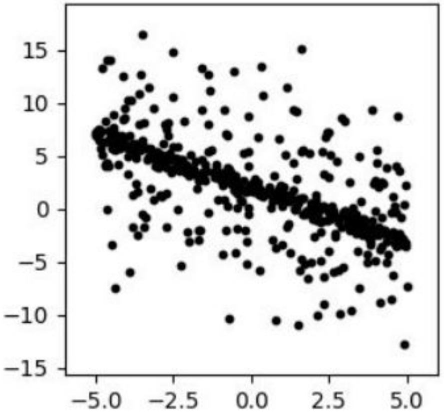
Users can optimize performance for slow components

Existing platform that meets requirements:

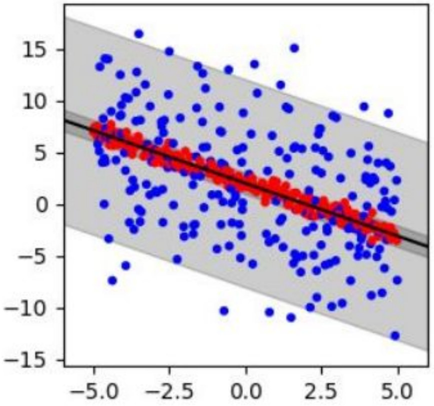
C? Julia?

More abstractions, please.

Example of programmable inference in Gen



Inference



Example of programmable inference in Gen

```
@gen function datum(x, @ad(slope)), @ad(intercept),
                       @ad(inlier_noise)), @ad(outlier_noise))
    is_outlier = @addr(bernoulli(0.5), :z)
    noise = is_outlier ? outlier_noise : inlier_noise
    y = @addr(normal(x * slope + intercept, noise), :y)
    return y
end

@static @gen function model(xs::Vector{Float64})
    n::Int = length(xs)
    slope = @addr(normal(0, 2), :slope)
    intercept = @addr(normal(0, 2), :intercept)
    inlier_noise = exp(@addr(normal(0, 2), :log_inlier_noise)
    outlier_noise = exp(@addr(normal(0, 2), :log_outlier_noise))
    ys = @addr(replicate(datum, xs, fill(slope, n), fill(intercept, n),
                         fill(inlier_noise, n), fill(outlier_noise, n)),
               :data)
    return ys
end
```

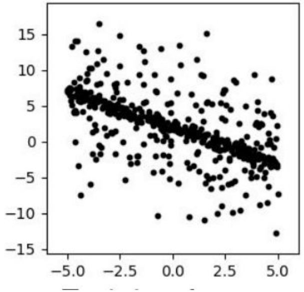
Generative model

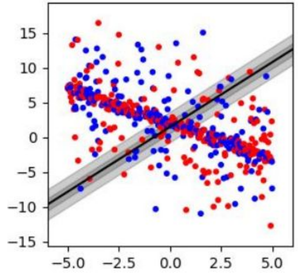
```
├─ :slope = 0.123
├─ :intercept = -0.2
├─ :log_inlier_noise = 0.4
├─ :log_outlier_noise = 2.3
└─ :data
   ├─ 1
   │  ├─ :z = true
   │  └─ :y = 1.22
   ├─ 2
   │  ├─ :z = false
   │  └─ :y = 1.23
   ├─ 3
   │  ├─ :z = true
   │  └─ :y = 1.32
   ..
   └─ 100
      ├─ :z = true
      └─ :y = 4.32
```

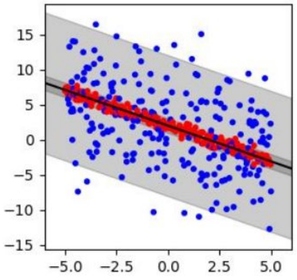
Hierarchical assignment

```
function my_inference_algorithm(xs::Vector{Float64}, ys::Vector{Float64})
  observations = Assignment()
  for i=1:length(xs)
     observations[:data => i => :y] = ys[i]
  end

  (trace, _) = generate(model, (xs,), observations)

  for iter=1:100

    # Gradient ascent moves on parameters
    for j=1:5
      trace = map_optimize(model, select(:slope, :intercept), trace)
      trace = map_optimize(model, select(:log_inlier_noise, :log_outlier_noise), trace)
    end

    # Metropolis-Hastings move on the outlier indicator variables
    for j=1:length(xs)
      trace = metropolis_hastings(model, custom_proposal, (j,), trace)
    end
  end
  return trace
end
```

Inference Program

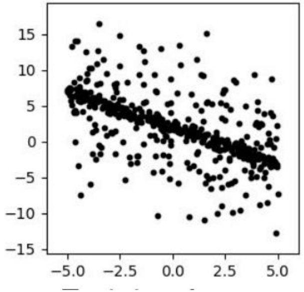


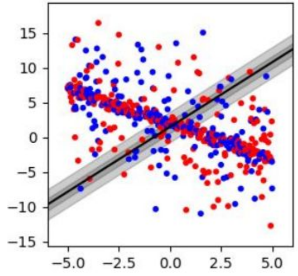


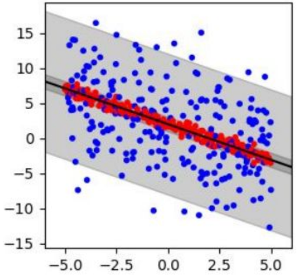


```
@gen function custom_proposal(previous_trace, i::Int)
    prev_is_outlier = get_assignment(previous_trace)[:data => i => :z]
    @addr(bernoulli(prev_is_outlier ? 0.0 : 1.0), :data => i => :z)
end

function my_inference_algorithm(xs::Vector{Float64}, ys::Vector{Float64})
  observations = Assignment()
  for i=1:length(xs)
     observations[:data => i => :y] = ys[i]
  end

  (trace, _) = generate(model, (xs,), observations)

  for iter=1:100

    # Gradient ascent moves on parameters
    for j=1:5
      trace = map_optimize(model, select(:slope, :intercept), trace)
      trace = map_optimize(model, select(:log_inlier_noise, :log_outlier_noise), trace)
    end

    # Metropolis-Hastings move on the outlier indicator variables
    for j=1:length(xs)
      trace = metropolis_hastings(model, custom_proposal, (j,), trace)
    end
  end
  return trace
end
```

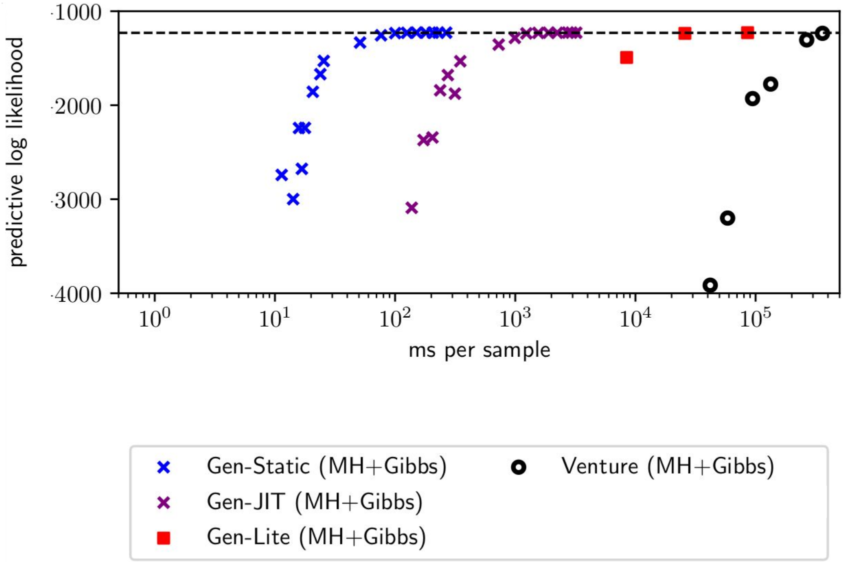
Inference Program



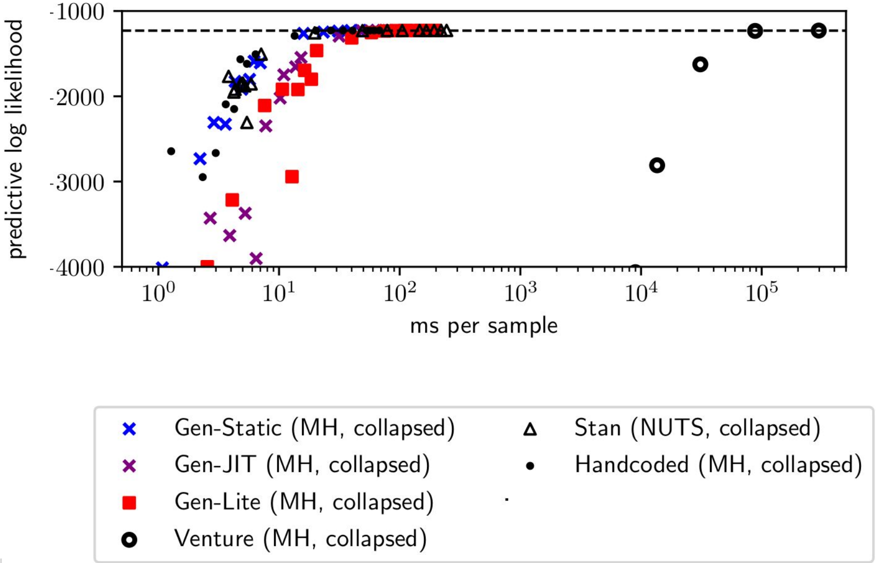




Performance of Gen's JIT compiler



Uncollapsed model



Manually collapsed model

Challenge: integrating multiple modeling & inference paradigms

Monte Carlo

- Models defined by arbitrary generative code in Julia
- Fast editing of execution traces during MCMC inference, via incremental computation
- Fast resampling of execution traces for SMC inference, via persistent data structures

Deep learning

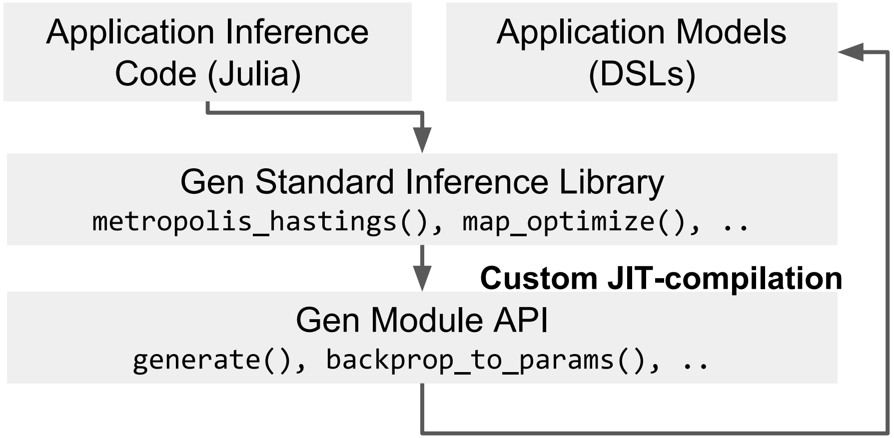
- Models defined by differentiable TensorFlow computations mixed with Julia code
- Batched gradients with respect to large parameter arrays located on GPU

Gradient-based inference

- Gradients with respect to ~10s of random variables (non-contiguous in memory)
- MAP, HMC, MALA, etc.

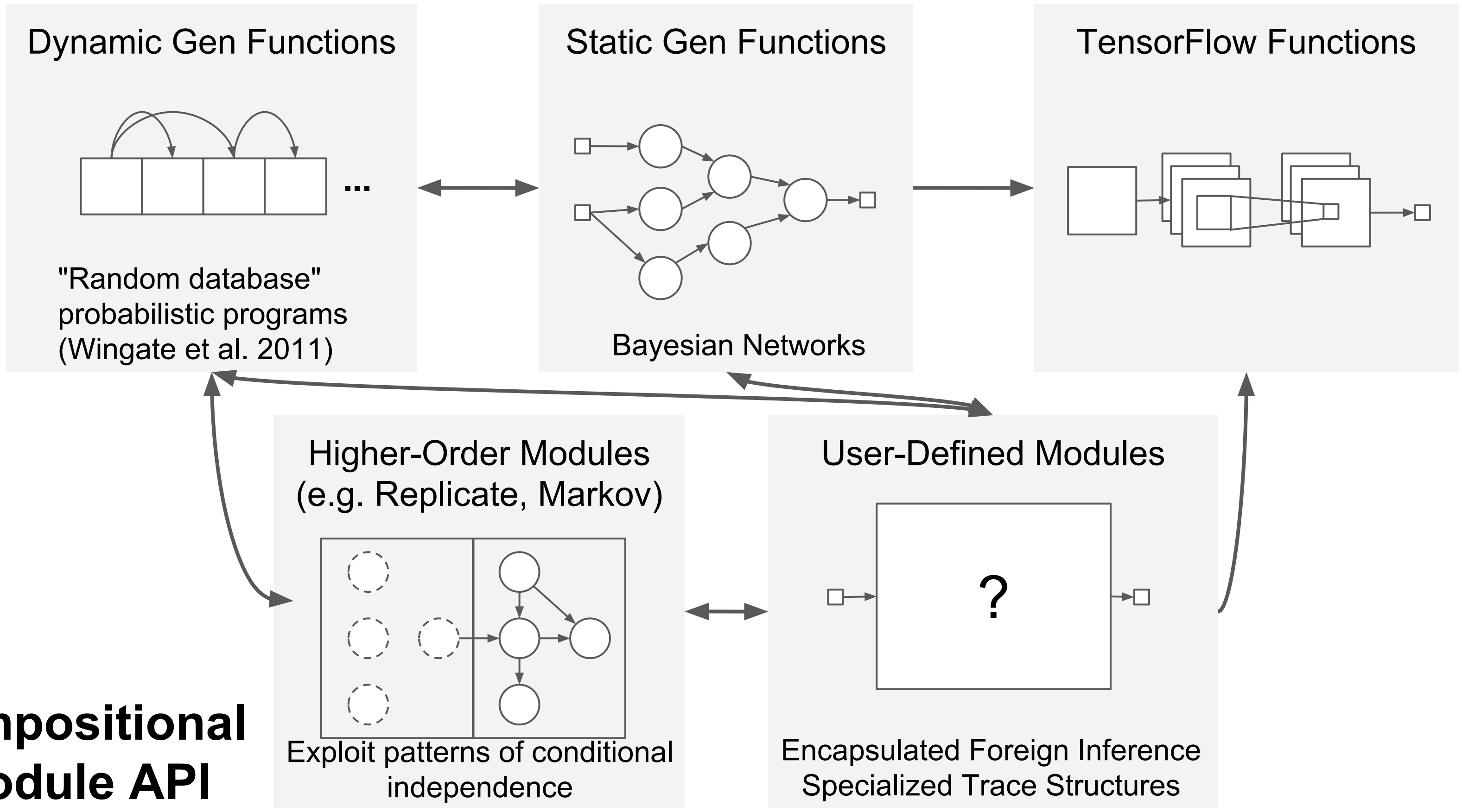
Key technical ideas in Gen

- Extensible set of domain-specific *Gen modules* that encapsulate modeling and inference
- Low-level *Gen module API* for creating, updating, and transforming execution traces
- JIT compilation, including static inference of execution trace types [1]
- *Standard inference library* for Monte Carlo, numerical optimization, and deep learning
- New mechanism for encapsulating auxiliary variables [2]



[1] Cusumano-Towner and Mansinghka, MAPL 2018
[2] Cusumano-Towner and Mansinghka, PPS 2017

Example types of Gen modules

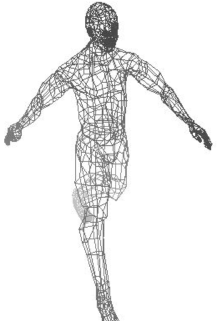


Compositional module API

Example: body pose inference as inverse graphics

```
struct BodyPose
    body_rotation::Point3D
    elbow_right_loc::Point3D
    elbow_left_loc::Point3D
    ...
end
```

3D model



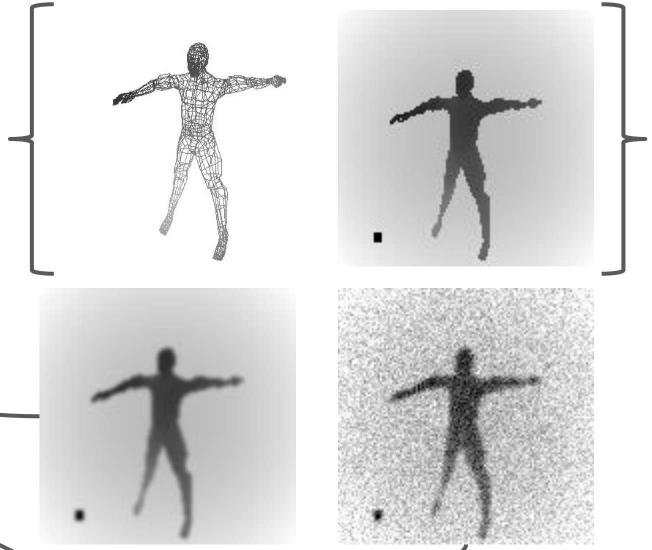
Renderer

Inference

Generative model based on a graphics engine

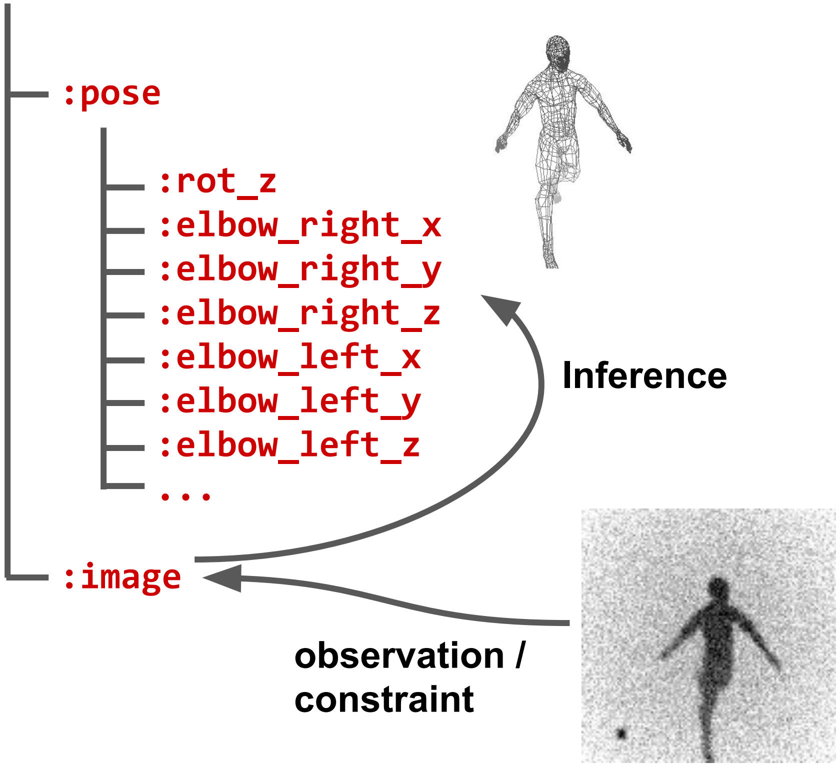
```
@gen function body_pose_prior()
    ...
end

@gen function generative_model()

    # sample pose from prior
    pose = @addr(body_pose_prior(), :pose)

    # render depth image and add blur
    image = render_depth_image(pose)
    blurred = gaussian_blur(image, 1)

    # pixel-wise likelihood model
    @addr(pixel_noise(blurred, 0.1), :image)
end
```

```
struct BodyPose
    rotation::Point3
    elbow_r_loc::Point3
    elbow_l_loc::Point3
    ...
end
```

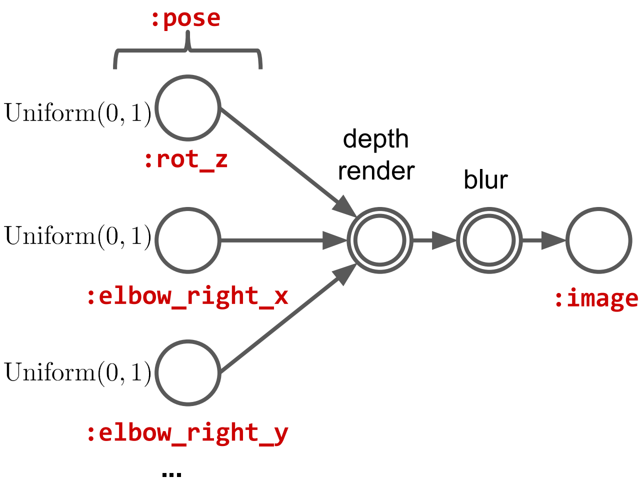


Generative model based on a graphics engine

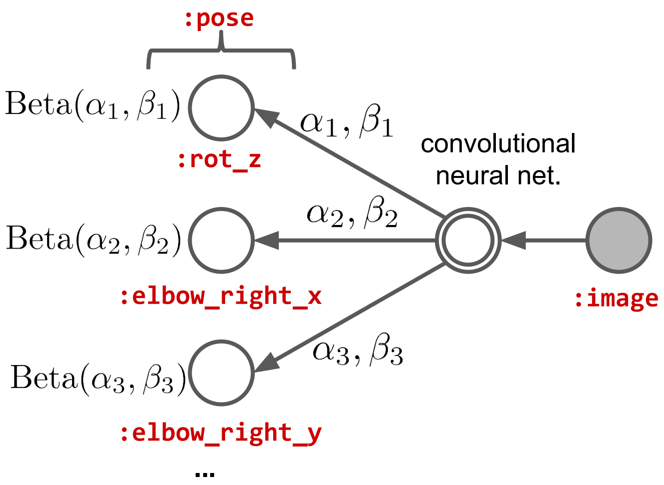
```
@gen function body_pose_prior()
    ...
end

@gen function generative_model()

    # sample pose from prior
    pose = @addr(body_pose_prior(), :pose)

    # render depth image and add blur
    image = render_depth_image(pose)
    blurred = gaussian_blur(image, 1)

    # pixel-wise likelihood model
    @addr(pixel_noise(blurred, 0.1), :image)
end
```



Inference using deep learning and Monte Carlo



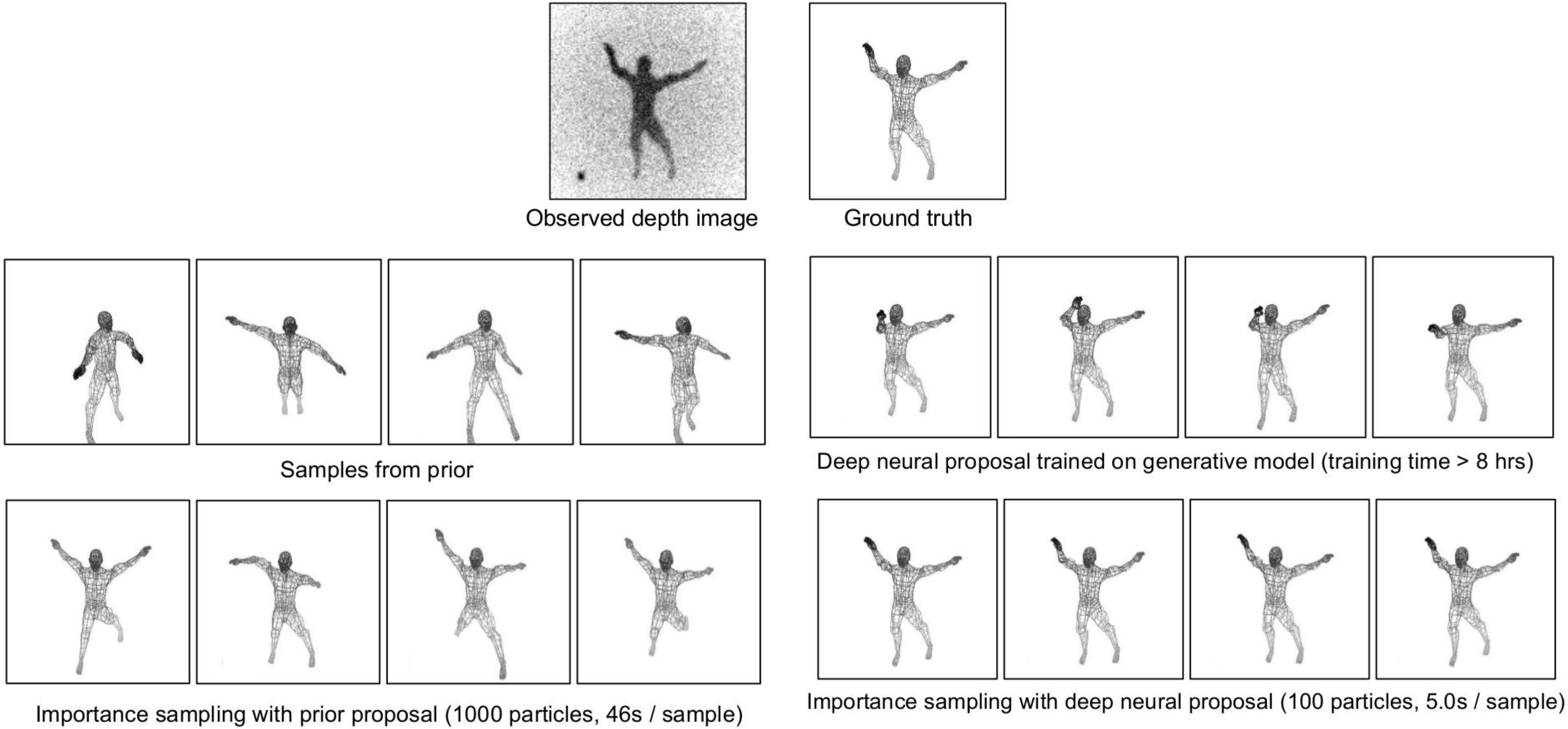
Generative model $p(\text{pose}, \text{image})$



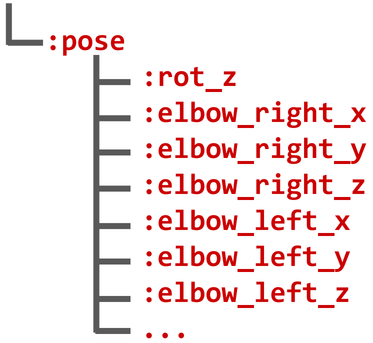
Importance distribution $q(\text{pose}; \text{image}, \phi)$

$$\max_{\phi} \mathbb{E}_{\text{pose},\text{image} \sim p(\cdot)} \left[\log q(\text{pose}; \text{image}, \phi) \right]$$

Optimize using sleep phase of wake-sleep algorithm (Hinton, 1995); inference compilation (Le et al., 2016); neural nested inference (Cusumano-Towner et al., 2017)


Observed depth image
Ground truth
Samples from prior
Deep neural proposal trained on generative model (training time > 8 hrs)
Importance sampling with prior proposal (1000 particles, 46s / sample)
Importance sampling with deep neural proposal (100 particles, 5.0s / sample)

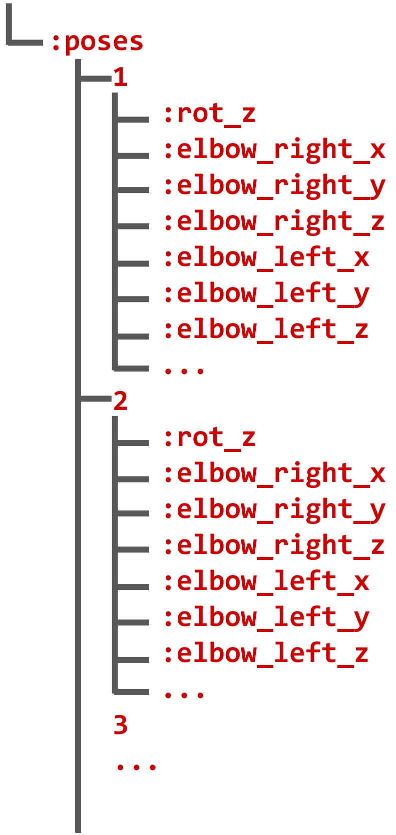
```
@gen function neural_proposal(image::Matrix{Float64})
    image_flat = reshape(image, 1, 128 * 128)
    output_layer = @addr(neural_network(image_flat), :network)
    @addr(predict_body_pose(output_layer[1,:]), :pose)
end

neural_network = @tensorflow_module begin

  @input image_flat Float32 [-1, 128 * 128]
  image = tf.reshape(image_flat, [-1, 128, 128, 1])

  @param W_conv1 initial_weight([5, 5, 1, 32])
  @param b_conv1 initial_bias([32])
  h_conv1 = tf.nn.relu(conv2d(image, W_conv1) + b_conv1)
  h_pool1 = max_pool_2x2(h_conv1)
  ...

  @param W_fc1 initial_weight([16 * 16 * 64, 1024])
  @param b_fc1 initial_bias([1024])
  h_fc1 = tf.nn.relu(h_pool3_flat * W_fc1 + b_fc1)

  @param W_fc2 initial_weight([1024, 32])
  @param b_fc2 initial_bias([32])

  @output Float32 (tf.matmul(h_fc1, W_fc2) + b_fc2)
end
```



```julia
@gen function neural_proposal(image::Matrix{Float64})
    image_flat = reshape(image, 1, 128 * 128)
    output_layer = @addr(neural_network(image_flat), :network)
    @addr(predict_body_pose(output_layer[1,:]), :pose)
end

@gen function predict_body_pose(@ad(output_layer::Vector{Float64}))

    # global rotation
    @addr(beta(exp(output_layer[1]), exp(output_layer[2])), :rot_z)

    # right elbow location
    @addr(beta(exp(output_layer[3]), exp(output_layer[4])), :elbow_right_x)
    @addr(beta(exp(output_layer[5]), exp(output_layer[6])), :elbow_right_y)
    @addr(beta(exp(output_layer[7]), exp(output_layer[8])), :elbow_right_z)

    # left elbow location
    @addr(beta(exp(output_layer[11]), exp(output_layer[12])), :elbow_left_x)
    @addr(beta(exp(output_layer[13]), exp(output_layer[14])), :elbow_left_y)
    @addr(beta(exp(output_layer[15]), exp(output_layer[16])), :elbow_left_z)
  ..
end
```

```
└:pose
    ├ :rot_z
    ├ :elbow_right_x
    ├ :elbow_right_y
    ├ :elbow_right_z
    ├ :elbow_left_x
    ├ :elbow_left_y
    ├ :elbow_left_z
    └ ...
```

```
@gen function neural_proposal_batched(images::Vector{Matrix{Float64}})

    images_flat = vectorize_images(images)

    # run inference network in batch
    output_layer = @addr(neural_network(images_flat), :network)

    # make prediction for each image given inference network outputs
    batch_size = length(images)
    for i=1:batch_size
        @addr(predict_body_pose(outputs[i,:]), :poses => i)
    end
end
```



Training

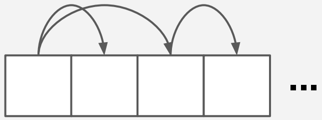
```
input_constructor = (training_assignments::Vector) -> ([assignment[:image] for assignment in training_assignments],)

function constraint_constructor(training_assignments::Vector)
    poses = vectorize_assignments([get_internal_node(a, :pose) for a in training_assignments])
    constraints = Assignment()
    set_internal_node!(constraints, :poses, poses)
    return constraint
end

minibatch_callback = (batch::Int, minibatch::Int, avg_score::Float64) -> tf.run(session, network_update)
batch_callback = (batch::Int) -> nothing

conf = TrainBatchedConf(num_batch, batch_size, num_minibatch, minibatch_size,
                             input_constructor, constraint_constructor, minibatch_callback, batch_callback)

train_batched(generative_model, (), neural_proposal_batched, conf)
```

Sampling importance resampling

```
observations = Assignment()
observations[:image] = image


(trace, _) = importance_resampling(generative_model, (), observations, neural_proposal, (image,), num_particles=100)


assignment = get_assignment(trace)
println(assignment[:elbow_r_loc_x])
```

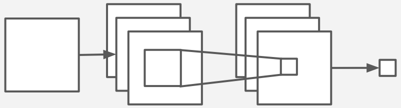
- Neural network encapsulated in a *TensorFlow Module*
- Compositional automatic differentiation using Gen module API
- Trainable parameters managed by individual modules

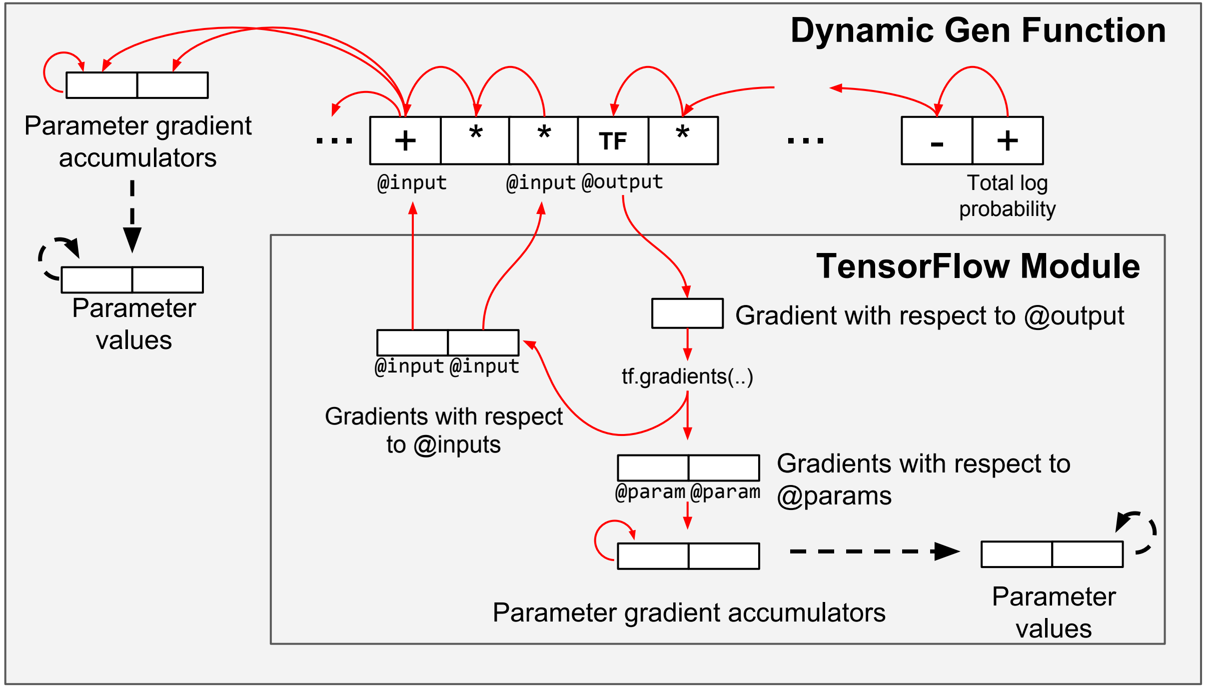
Dynamic Gen Functions



"Random database" probabilistic programs (Wingate et al. 2011)

TensorFlow Modules





Summary

- Prototype probabilistic programming platform for probabilistic AI
- Multi-paradigm, programmable inference, good performance
- Key idea: Gen Modules
- Embedded in Julia with TensorFlow integration

Thanks!