Testing Inference Programs for Generative Scene Graphs

by

Austin J. Garrett

Department of Electrical Engineering and Computer Science

Proposal for Thesis Research in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 2019

Author .....
Department of Electrical Engineering and Computer Science
December 11, 2019

Certified by .....
Vikash Mansinghka
Research Scientist
Thesis Supervisor

Accepted by .....
Placeholder
Chairman, Department Committee on Graduate Theses

Testing Inference Programs for Generative Scene Graphs

by

Austin J. Garrett

Submitted to the Department of Electrical Engineering and Computer Science
on December 11, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed and implemented a compiler which performs optimizations that reduce the number of low-level floating point operations necessary for a specific task; this involves the optimization of chains of floating point operations as well as the implementation of a "fixed" point data type that allows some floating point operations to simulated with integer arithmetic. The source language of the compiler is a subset of C, and the destination language is assembly language for a micro-floating point CPU. An instruction-level simulator of the CPU was written to allow testing of the code. A series of test pieces of codes was compiled, both with and without optimization, to determine how effective these optimizations were.

Thesis Supervisor: Vikash Mansinghka
Title: Research Scientist

Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Testing scene graph models

1.1.1 History of Inverse Graphics

1.1.2 Scene Graphs as Compact Scene Representations

1.2 Summary of Thesis

TODO: replace me

Chapter 2

Scene Graphs

2.1 Mathematical Description

We model the geometric state of a scene at a single time point as a *scene graph*, which is a tuple (G, Θ, Z), where $G = (V, E)$ is a directed tree that encodes the scene graph *structure* and Θ encodes the scene graph *continuous parameters* and Z encodes the scene graph *discrete parameters*. Although the framework we present can represent articulated objects, in this paper we only consider scenes involving a set of rigid objects O. The scene graph structure includes a vertex $v_o \in V$ for each object $o \in O$ that represents the 6DoF pose of object o, as well as a vertex $r \in V$ that represents the the coordinate frame of the observer. A directed edge $e = (v_i, v_j) \in E$ between objects $i, j \in O$ encodes that the pose of object j is parametrized *relative to* the pose of object i, by parameters θ_e and z_e. The pose of object j relative to object i is denoted $\Delta x(z_e, \theta_e) \in SE(3)$. A directed edge $e = (r, v_j) \in E$ from the root to an object $j \in O$ encodes that the pose of object j is not parametrized relative to that of any object, but is instead a full 6DoF pose $\theta_e \in SE(3)$ where $SE(3)$ is the special Euclidean group consisting of all 6DoF poses. The continuous and discrete parameters of the scene graph consist of the parameters for each edge in the structure ($\Theta := \{\theta_e\}_{e \in E}$ and $Z := \{z_e\}_{e \in E}$). The set of edges E must *span* the set of vertices $V := \{r\} \cup \{v_o\}_{o \in O}$; that is the scene graph structure G is a *directed spanning tree*.

2.1.1 Computing the 6DoF poses of all objects in the scene

Given a scene graph (G, Θ, Z) the pose $x_i \in SE(3)$ of an object $i \in O$ relative to the observer coordinate frame can be computed by walking path in the tree G from the root vertex $r \in V$ to the vertex $v_i \in V$, and successively computing the pose of each object along the path from the pose of its parent (the set of all such poses is $X := \{x_i\}_{i \in O}$. That is, given pose $x_u \in SE(3)$ and edge $(u, v) \in E$, the pose $x_v \in SE(3)$ is computed as $x_v := x_u \cdot \Delta x(z_e, \theta_e)$ where $\Delta x(z_e, \theta_e) \in SE(3)$ is the relative pose between vertex u and vertex v. For an edge from the root vertex to an object vertex ($e = (r, v_i)$), $x_u := \mathbf{1}$ (the identity element of $SE(3)$) and $\Delta x(z_e, \theta_e) = \theta_e \in SE(3)$ (note that there are no discrete parameters in this case, so $z_e := ()$). The functional form of $\Delta x(z_e, \theta_e)$ for an edge e between two objects is discussed below. One requirement is that $\Delta x(z_e, \theta_e)$ is a differentiable function of θ_e; this enables inference algorithm that exploit gradient information.

2.1.2 Modeling face-to-face contact between two objects

An edge $(v_i, v_j) \in E$ from object i to object j indicates that the pose of object j is represented relative to the pose of object i. Various types of relative pose parametrizations for two objects are possible; for simplicity we model objects as polyhedra, and only model face-to-face contact between objects. That is, an edge $e = (v_i, v_j)$ indicates that a face of object i is in flush contact with a face of object j. Since each object has multiple faces, the choice of which pair of faces is in contact is encoded in the discrete parameters z_e for the edge. Concretely, let F_i and F_j denote the faces of object i and object j respectively Then, $z_e \in F_i \times F_j$. The continuous parameters θ_e for an edge e between two objects is an element of $\mathbb{R}^2 \times [0, 2\pi)$ that contains two translational degrees of freedom ($s, t \in \mathbb{R}$, for the relative offset of the two faces) and one rotational degree of freedom ($\phi \in [0, 2\pi)$). For example, a cuboid object $i \in O$, the set of faces is $F_i = \{\text{Top}, \text{Bottom}, \text{Left}, \text{Right}, \text{Front}, \text{Back}\}$. For an edge $e = (i, j)$ where objects $i, j \in O$ where both objects i and j are cuboids, there are 36 possible values for z_e.

Slack variables for face-to-face contact We extend the parametrization of face-to-face contact between objects with three additional degrees of freedom of *slack variables*: (i) one degree of freedom that encodes the perpendicular distance ($d \in [0, \infty)$) between the two contact faces, and (ii) two degrees of freedom for relative orientation of the two faces, encoded the surface normal unit vector $\mathbf{n}$ of the child object's face, relative to the parent object's face, which takes values on the sphere S^2. Therefore, $\theta_e \in \mathbb{R}^2 \times [0, 2\pi) \times [0, \infty) \times S^2$ for an edge $e = (v_i, v_j)$ between two objects i and j. Note that although this edge parametrization uses six degrees of freedom for object-to-object edges like the edge parametrization for edges from the root ($e = (r, v_j)$), the prior distribution on these parameters (described below) encourages object j to be *almost* in face-to-face contact with object i; whereas the prior distribution on the pose of object j for an edge of the form (r, v_j) is very different, and is typically a uniform distribution over positions within the scene bounding volume, and a uniform distribution on orientations.

2.1.3 A Prior Distribution on Scene Graphs

Various prior distributions on scene graphs are possible. In our experiments, we use a generic prior distribution on scene graphs over a collection of objects O that factors into two components: (i) a prior distribution on scene graph structures G, denoted $p(G)$, and (ii) a prior distribution on scene graph parameters (Z, Θ) given structure, denoted $p(Z, \Theta|G)$. While uncertainty about the number of objects is possible to represent in our framework and implementation, for simplicity assume that the set of objects is known a-priori, and that there is one vertex for each object, and one vertex representing the observer coordinate frame, so V is fixed a-priori to $\{r\} \cup \{v_o\}_{o \in O}$. Therefore, $p(G)$ reduces to a prior distribution on edges in the scene graph. For the prior distribution on structure, we use a uniform distribution on directed trees that are rooted at vertex r and span all $|O| + 1$ vertices in the graph. This set of directed trees is isomorphic to the set of undirected spanning trees over $|O| + 1$ vertices. Therefore, the prior probability of a graph $G = (V, E)$ is obtained by using Cayley's formula to count the number of undirected spanning trees on $|O| + 1$

vertices:

$$p(G) := p_{\text{unif}(|O|)}(G) := \begin{cases} (|O|+1)^{1-|O|} & \text{if } G \text{ is a directed spanning tree over vertices } V \text{ rooted at } r \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

The prior distribution on scene graph parameters factors over the edges in the scene graph:

$$p(Z, \Theta|G) := \prod_{e \in E} p_e(z_e, \theta_e) \tag{2.2}$$

where $p_e(z_e, \theta_e)$ is a probability distribution on z_e and θ_e that depends on the two vertices in the edge $e = (u, v)$. For edges $e = (r, v_j)$ where j is an object, $z_e = ()$ and $p_e(z_e, \theta_e)$ is the uniform distribution on elements of $B \times SO(3)$ where B is the bounding volume of the scene and $SO(3)$ is 3D rotation group (the uniform distribution on $SO(3)$ is given by the Haar measure). For edges $e = (v_i, v_j)$ where i and j are objects, recall that $z_e \in F_i \times F_j$ and (using the edge parametrization including slack variables) $\theta_e = (s, t, \phi, d, \alpha_1, \alpha_2) \in \mathbb{R}^2 \times [0, 2\pi) \times [0, \infty) \times [0, 2\pi)^2$. The prior distribution on edge parameters is:

$$p_e(z_e, \theta_e) := \frac{1}{|F_i||F_j|} \cdot p_{\text{norm}(0,\sigma)}(s) \cdot p_{\text{norm}(0,\sigma)}(t) \cdot \frac{1}{2\pi} \cdot p_{\text{exp}(\beta)}(d) \cdot \frac{1}{2\pi} \cdot \frac{1}{2\pi} \tag{2.3}$$

(where p_{norm} and p_{exp} are the normal and exponential distribution density functions, respectively).

2.1.4 Probabilistic Dynamics on Scene Graphs

We build temporal models of scene geometry based on Markov chains of scene graphs $\{(G_t, \Theta_t, Z_t)\}_{t=1}^T$, where t indexes time, with transitions $p(G_t, \Theta_t, Z_t|G_{t-1}, \Theta_{t-1}, Z_{t-1})$. We factor the dynamics model on scene graphs decomposes into a dynamics model on scene graph structure, and a dynamics on parameters:

$$p(G_t, \Theta_t, Z_t|G_{t-1}, \Theta_{t-1}, Z_{t-1}) := p(G_t|G_{t-1}) \cdot p(Z_t, \Theta_t, |Z_{t-1}, \Theta_{t-1}, G_{t-1}, G_t) \tag{2.4}$$

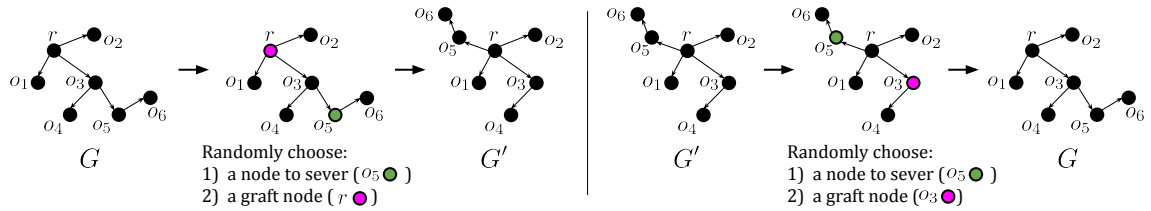


Figure 2-1: Our random walk transition model on the structure of scene graphs modifies the spanning tree that defines the scene graph structure. Left: A transition from structure G' to G that makes o_5 a child of the root r instead of object o_3. In G, parametrization of the pose of o_5 was previously parametrized relative to the pose of object o_3; in G' it is parametrized independently of the pose of the other objects. Right: The transition on structures is reversible, and the reverse transition is unique. This allows us to use the transition model on structure as the basis of reversible jump MCMC moves on structure, as well within the dynamics model on structure.

The dynamics on graph structure is based on a mixture of a random walk on structures (to capture incremental changes to the structure of a scene that often occur) and a uniform distribution on structures (to model sudden unexpected changes in structure):

$$p(G_t|G_{t-1}) := 0.9 \cdot p_{\text{walk}}(G_t|G_{t-1}) + 0.1 \cdot p_{\text{unif}(|O|)}(G_t) \tag{2.5}$$

where $p_{\text{walk}}(G'|G)$ is the distribution on graph structures induced by a sampling process in which one vertex in the undirected spanning tree is selected at random, severed from the tree, and grafted back onto the graph at a uniformly chosen vertex, subject to the condition that the resulting graph is a spanning tree; Figure **??** shows an example of this process. Recall that since the tree contains the root node r, this process can change the edge type (u, v_j) for an object i from (r, v_j) (representing independent 6DoF pose) to (v_i, v_j) (representing flush contact pose with another object v_i. The dynamics on parameters factor according to edges in the graph:

$$p(Z_t, \Theta_t|Z_{t-1}, \Theta_{t-1}, G_{t-1}, G_t) := \prod_{e \in E} p_e(z_e, \theta_e|G_t, G_{t-1}, z_{t-1_e}, \theta_{t-1_e}) \tag{2.6}$$

If an edge e is present in both G_t and G_{t-1}, then this distribution is a mixture of a random walk and a uniform distribution. If an edge e in G_t was not present in G_{t-1}, then this distribution is a uniform distribution.

2.1.5 Robust Pose Likelihood for Modeling Noisy Object Detections

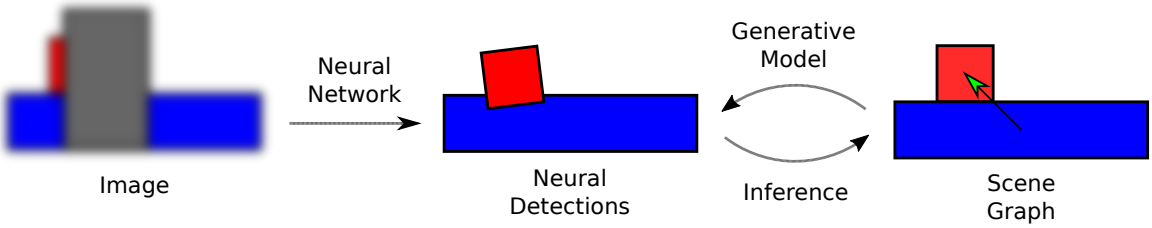


Figure 2-2: Abstract representation of our scene graph model; image data is parsed into noisy unstructured neural detections, which are modeled as observations of an underlying structured scene graph.

Our model differs from previous inverse graphics work in an important way; while most inverse graphics pipelines create a full generative model all the way to pixel-level data, we instead model the output of a bottom-up feature detector that estimates the poses of objects from visual data. This was specifically motivated by the desire to leverage the power of existing deep learning approaches to object detection, and integrate them with a Bayesian approach. Figure 2-2 shows the abstract pipeline for our model. Our generative model is thus grounded in attempting to explain the output of a noisy detector, which requires us to accurately predict the presence and distribution of "outlier" detections. In practice, the failure modes of deep neural networks are complex and varied, and can depend on a huge range of factors, including occlusion, lighting condition, the presence of novel objects, etc. We provide a simple first-pass likelihood model for representing noisy bottom-up detectors, and note that developing more accurate likelihoods for either noisy bottom-up detectors, or directly rendered visual data, is an important next step.

We introduce a likelihood over observed object pose detections $Y := \{y_o\}_{o \in O} = \{(y_o^{\text{pos}}, y_o^{\text{rot}})\}_{o \in O}$. The likelihood consists of two components in a mixture model: (i) a uniform distribution over a finite cuboid region in 3D space, and (ii) a product of a diagonal multivariate Gaussian over object positions, and a von Mises–Fisher distribution over rotations, represented as unit vectors in S^3 (quaternions). The first component represents the possibility of outlier neural "flicker", where objects

may be properly recognized as being in the scene, but are very poorly localized. The second component represents where the inlier cases where the neural detector roughly captures the correct location of the object in the scene, with small local inaccuracies.

$$p(Y|X,\sigma,\kappa) = \prod_{o \in O} \left(p_{\text{outlier}} \cdot \frac{1}{LWH} + (1 - p_{\text{outlier}}) \cdot \mathcal{N}\left(y_o^{\text{pos}}; x_o^{\text{pos}}, \sigma\right) \cdot \text{VMF}(y_o^{\text{rot}}; x_o^{\text{rot}}, \kappa) \right) \tag{2.7}$$

2.1.6 Reversible Jump MCMC for Structure Inference

When considering how to infer the posterior distribution over scene graph structure G, we might take inspiration from our graph random walk distribution $p_{\text{walk}}(G_t|G_{t_1})$ to build an incremental graph modification kernel that proposes small changes to the structure. However, when changing an edge type from $e \to e'$, we also induce a new set of parameters $\theta_{e'}$ that must be simultaneously proposed or resimulated. Ultimately these two parameterizations are just different ways to express identical absolute poses $x_j = x'_j$, as either relative to another object, or to the observer coordinate frame.

Reversible jump MCMC (RJMCMC) is a generalization of the MH algorithm that allows moves between models with different state spaces, while reusing information in the different model latents to accelerate inference. We provide the formulation of RJMCMC here without proof of asymptotic correctness; for a full derivation see Green et. al [**?**]. We define moves over a projection of the scene graph model to the static case; the dynamic case can easily be extended with an additional parameter for time.

Define a set of models $k \in \mathcal{K}$, each with latent variables $\theta_k \in \mathbb{R}^{N_k}$ and joint density $p_k(k, \theta_k, \mathcal{D})$, as well as a set of reversible jump moves $m \in \mathcal{M}$, each of which switches between two models $k, k' \in \mathcal{K}$. For each model, we have a distribution over possible moves $q_k(m)$ with support over a subset of $\mathcal{M}$. Each reversible jump move from $k \to k'$ proposes a value for $\theta_{k'}$ given θ_k, using a differentiable bijection f. However, in general $N_k \neq N_{k'}$, which prevents us from making a bijection between these two spaces. To ameliorate this, we pad the latent spaces of both models with additional auxiliary randomness. In the forward direction ($k \to k'$), we sample $u \sim q(\cdot)$ where $u \in \mathbb{R}^{D_k}$,

and in the backward direction ($k' \to k$), we sample $u \sim q'(\cdot)$ where $u \in \mathbb{R}^{D_{k'}}$; we additionally require $N_k + D_k = N_{k'} + D_{k'}$ such that we can define a bijection between our two extended spaces. We then let the bijection be between these two extended state spaces; the forward move then proposes new values $(\theta_{k'}, u') = f(\theta_k, u)$, which we accept with probability $\min\{1, \alpha\}$, where

$$\alpha = \frac{p_{k'}(k', \theta_{k'}, \mathcal{D}) q(u) q_{k'}(m)}{p_k(k, \theta_k, \mathcal{D}) q'(u') q_k(m)} \cdot |\det J_f| \tag{2.8}$$

Because our different parameterizations ultimately represent the same absolute poses, RJMCMC offers a way to reuse our inferred continuous parameters θ_e to efficiently propose values for $\theta_{e'}$. Explicitly, we define a class of reversible jump moves parameterized by $j \in V$ that determines which node will have a new parent proposed. If j is floating in $G = (V, E)$, then the models reachable have structure $(V, E \setminus \{(r, j)\} \cup \{(i, j)\})$, and discrete parameter $z_{(i,j)} = (f_i, f_j)$. The corresponding moves are sampled from the distribution $m_{\text{sliding}} = (i, f_i, f_j) \sim q_{\text{floating}}(\cdot, \cdot, \cdot)$. Inversely, if j is sliding, then the only model reachable has structure $(V, E \setminus \{(i, j)\} \cup \{(r, j)\})$. We call the move corresponding to this m_{floating}, and $q_{\text{sliding}}(m_{\text{floating}}) = 1$. The slack terms mean both floating and sliding continuous parameters have 6DoF, thereby precluding the need for sampling auxiliary continuous random variables; this is the main motivation for explicitly modeling a slack term.

The final component are the bijections. All sliding to floating moves have the bijection $f_{m_{\text{floating}}}(G, \Theta, Z) = (G', \Theta', Z')$, where $E' = E \setminus \{(i, j)\} \cup \{(r, j)\})$, $z'_{(i,j)} = ()$, and $\theta_{(i,j)}$ is calculated using the method described in section 2.1.1. All floating to sliding moves inversely have the bijection $f_{(i,f_i,f_j)}(G', \Theta', Z') = (G, \Theta, Z)$, where $E = E' \setminus \{(r, j)\} \cup \{(i, j)\})$, $z'_{(i,j)} = (f_i, f_j)$, and $\theta_{(i,j)}$ is calculated as the sliding continuous parameters given contacting faces f_i, f_j and parent pose x_i, such that the absolute pose $x'_j = x_j$.

The position parameterizations in the floating and sliding cases are both $\mathbb{R}^3$, and thus the Jacobian correction is simply 1 for these subspaces. Orientations are slightly more complex; in the floating case, the orientation is represented as a unit vector in S^3

(which has a double-covering surjective homomorphism to the rotation group SO(3)). In the sliding case, the orientation is represented as the Hopf fibration, which has local product structure $S^2 \times S^1$. This local topology means our Jacobian correction for the transformation between these spaces is constant. TODO: Ask Marco how he got the Jacobian correction

$$\left|\det J_{f_{m_{\text{sliding}}}}\right| = \left|\det J_{f_{m_{\text{floating}}}}\right|^{-1} = 4 \tag{2.9}$$

Gen automatically calculates the acceptance ratio for RJMCMC and makes appropriate MH moves, given the involutions and associated Jacobian corrections.

2.2 Representation as a Generative Program

We implement our scene graph model in the Gen probabilistic proramming system[1] [?]. We leverage the dynamic DSL provided within Gen to implement the different modeling components as modular generative functions. Figure 2-3 shows example code of a probabilistic program that implements our scene graph model.

Gen provides an abstract interface to automatically run user-specified programmable inference on a generative program using its built-in inference library. Among the algorithms provided, Gen provides utilities for running RJMCMC, by specifying a generative proposal function to generate auxiliary randomness, and an associated involution function that provides the associated Jacobian correction. Figure 2-4 shows an user-space implementation of these algorithmic components.[2].

[1] `https://www.gen.dev`

[2] We note that at the time of writing this document, Gen has added a DSL for involutive MCMC, which can automate the calculation of Jacobian corrections using differentiable programming, and provides convenience macros to further simplify the development of custom RJMCMC algorithms. See documentation at `https://www.gen.dev/dev/ref/mcmc/#Involutive-MCMC-1`

```
@gen function model(T::Int)
    latent_gs = []
    for t = 1:T
        # structure and parameter dynamics
        if t == 1
            structure ~ UniformDirectedForest(N)
            params ~ params_init(structure)
        else
            (prev_structure, prev_params) = decompose(gs[t-1])
            structure ~ StructureTransition(prev_structure)
            params ~ params_dynamics(structure, prev_params)
        end

        # observation
        latent_g = SceneGraph(structure, params)
        obs ~ noise_model(latent_g)
        push!(latent_gs, latent_g)
    end
    return latent_gs
end
```

(a) Top-level scene graph model

```
@gen function init_params(structure::SimpleDiGraph)
    params = []
    for i in vertices(structure)
        if isFloating(structure, i)
            xs = {i} ~ init_floating_pose()
        else
            xs = {i} ~ init_sliding_pose(parent(structure, i))
        end
        push!(params, xs)
    end
    return params
end
```

(b) Parameter initialization

```
@gen function params_dynamics(structure, prev_params, hypers)
    new_params = []
    for i in vertices(structure)
        if isFloating(structure, i)
            new_xs = {i} ~ floating_pose_dynamics(prev_params[i])
        else
            new_xs = {i} ~ sliding_pose_dynamics(prev_params[i], parent(structure, i))
        end
        push!(new_params, new_xs)
    end
    return new_params
end
```

(c) Parameter dynamics

```
RobustNoisyPoseLikelihood = Mixture([GaussianVMF, UniformPose])

@gen function noise_model(g)
    # use Mixture to construct a mixture of uniform and gaussian vmf with prob_outlier of being uniform
    observed_poses = []
    for i in vertices(g)
        latent_pose = get_floating_pose(g, i)  # get 6DoF pose for object i
        observed_pose = {i} ~ RobustNoisyPoseLikelihood([1 - p_outlier, p_outlier],
                                                        [(latent_pose, inlier_pos_stdev, inlier_rot_conc),
                                                        (outlier_bounds,)])
        push!(observed_poses, observed_pose)
    end
    return observed_poses
end
```

(d) Noisy observational model

Figure 2-3: Probabilistic pseudocode for an example dynamic scene graph model

```
@gen function structure_move_randomness(prev_trace, i::Int)
    prev_structure = get_structure(prev_trace)
    floating_to_sliding_move = isFloating(prev_structure, i)

    if floating_to_sliding_move
        prev_parent_node = ROOT_NODE_ID

        # sample a new parent object from a categorical
        probs = ones(N)
        probs[i] = 0.0  # don't propose to make the object its own parent
        new_parent_node_object ~ categorical(1:N, probs ./ sum(probs))

        # sample new faces from a categorical
        box_faces = [:bottom, :top, :left, :right, :front, :back]
        (parent_face_probs, child_face_probs) =
            structure_move_face_distributions(prev_trace, i, parent_object)
        parent_face ~ categorical(box_faces, parent_face_probs)
        child_face ~ categorical(box_faces, child_face_probs)
    else
        prev_parent_node = parent(prev_structure, i)
        new_parent_node = ROOT_NODE_ID
    end
    new_structure = replaceEdge(prev_structure, i, prev_parent_node, new_parent_node)
    return (new_structure, floating_to_sliding_move)
end
```

(a) Generative function for sampling auxiliary randomness. In Gen, sampling the discrete move is combined with sampling continuous auxiliary randomness in a single function.

```
function structure_move_involution(
    prev_trace,              # original sampled scene graph model
    fwd_randomness_choices,  # choices (i, f_i, f_j) sampled from structure_move_randomness
    fwd_randomness_ret,      # values returned from structure_move_randomness
    fwd_randomness_args      # arguments passed to structure_move_randomness
)
    i, = fwd_randomness_args
    (new_structure, floating_to_sliding_move) = fwd_randomness_ret
    prev_scene_graph = get_latent_scene_graph(prev_trace)

    proposed_choices = choicemap()  # to set proposed reversible jump moves in the trace
    bwd_choices = choicemap()       # to tell Gen which involution we're using in the backward direction
    proposed_choices[:structure] = new_structure

    if floating_to_sliding_move
        parent_object = fwd_randomness_choices[:parent_object]
        parent_face = fwd_randomness_choices[:parent_face]
        child_face = fwd_randomness_choices[:child_face]
        sliding_choices =
            get_equivalent_sliding_param(prev_scene_graph, parent_object, i, parent_face, child_face)
        set_submap!(proposed_choices, :params => (:sliding, i), sliding_choices)
        log_jacobian_correction = log(4)
    else
        prev_parent_object = parent(get_structure(prev_scene_graph), i)
        (choices, prev_parent_face, prev_child_face) =
            get_equivalent_floating_param(prev_scene_graph, prev_parent_object, i)
        bwd_choices[:parent_object] = prev_parent_object
        bwd_choices[:parent_face] = prev_parent_face
        bwd_choices[:child_face] = prev_child_face
        set_submap!(proposed_choices, :params => (:floating, i), cm)
        log_jacobian_correction = -log(4)
    end

    # update trace with proposed values
    args = get_args(prev_trace)
    argdiffs = map((_) -> NoChange(), args)
    new_trace, weight, = update(prev_trace, args, argdiffs, proposed_choices)
    weight += log_jacobian_correction
    return (new_trace, bwd_choices, weight)
end
```

(b) Involution for reversible jump moves

Figure 2-4: Custom implementation of RJMCMC algorithm for structure inference

2.3 Example Application Domains of Scene Graphs

We present some example visual perception domains where scene graphs provide a useful abstraction for representing scene information. Our applications have thus far been applied with a specific orientation toward two main problems: inferring structured object relationships from unstructured poses of those objects, and robustly filtering object poses under failures in bottom-up visual perception. Our modeling choices were made with these goals in mind, so we focus on describing how these features are captured by our representation. In theory, we can extend these models to include even richer information about a scene, including dynamic physical properties of objects, or their functional (as opposed to geometric) relationships; we leave these and other applications for later work.

2.3.1 YCB Objects on a Synthetic Tabletop

Our scene graph abstraction is concretely implemented as a part of the GenSceneGraphs library. This library provides facilities for constructing and manipulating scene graphs composed of objects that can be rendered in a synthetic view. Figure **??** shows a variety of example scenes generated using this library; each visualized scene has an underlying scene graph representation that captures all relevant information. As such, these rendered scenes show an intuitive view of what our information our representation captures.

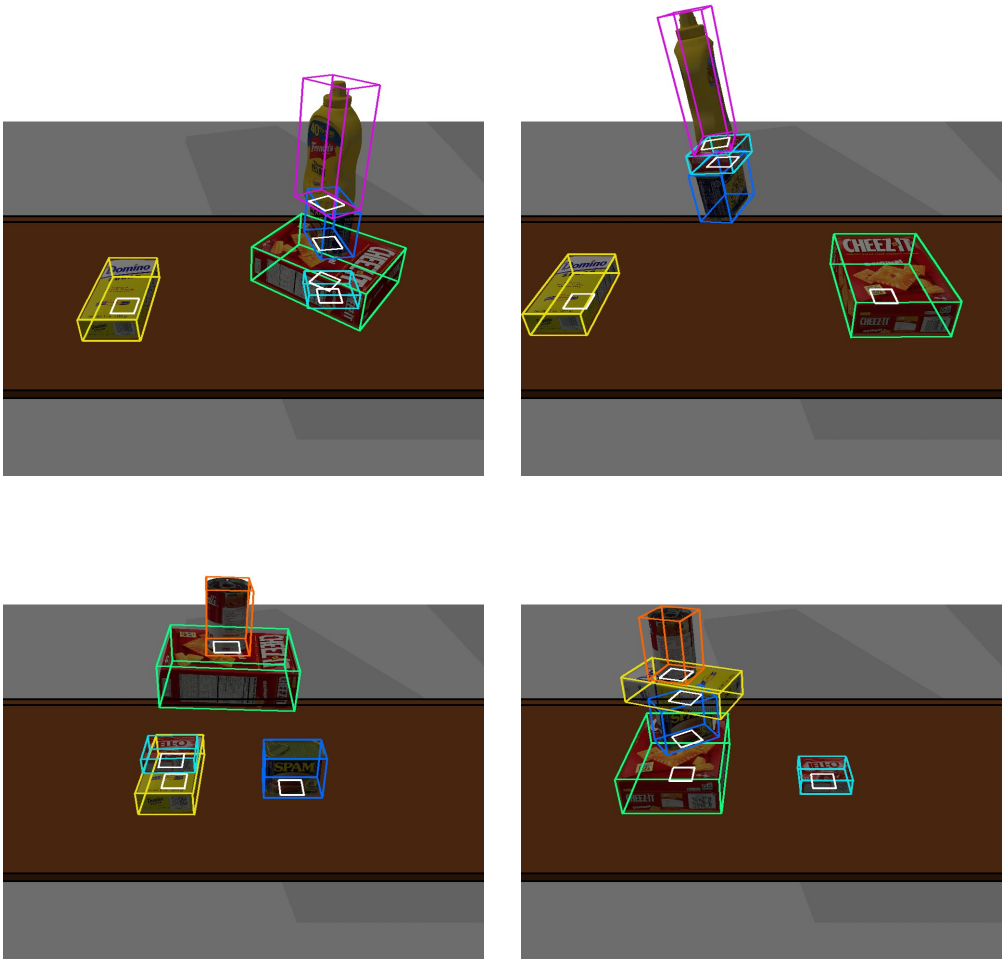
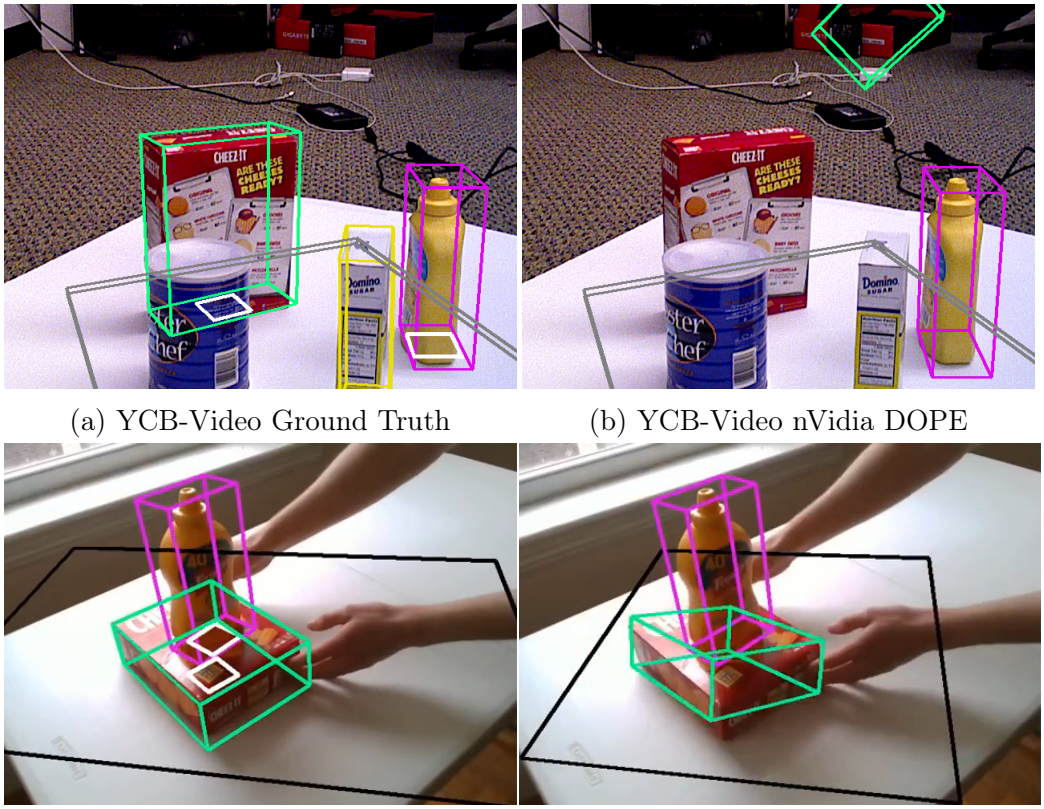


Figure 2-5: Synthetic scenes generated using the GenSceneGraphs library in simulated dim light. The colored wireframe bounding boxes represent objects in the scene graph, and the white boxes represent contact edges (see Chapter 3 for more detail on how this visualization was produced).

2.3.2 Real YCB Objects on a Physical Tabletop



(a) YCB-Video Ground Truth (b) YCB-Video nVidia DOPE

(c) In-House Dataset (d) In-House Dataset nVidia DOPE

Figure 2-6: Example data from real-world scenes containing objects from the YCB dataset. (a) and (c) show the underlying latent scene graph. (b) and (d) show observed inaccurate pose estimates generated from the nVidia DOPE detector. (b) in particular demonstrates the types of catastrophic detection and localization errors that deep neural networks can exhibit.

Our scene graph representation can be used to represent information about real-world scenes as well. Figure 2-6 shows how scene graphs can represent the underlying poses and geometric relationships between objects in the real world. In contrast to the synthetic scenes, the real world contains a much larger variety and range of extraneous variables, that inject noise into visual perception systems and make inference more difficult. Deep neural networks have shown incredibly promise in real-world object perception. However, neural networks struggle with robustness and generalization, and the presence of occluders, or even unfamiliar objects, can cause them to fail, as Figure 2-6 also shows. Our scene graph model was designed with the goal of filtering

noisy visual perception systems, and enhancing them with structured information, by inferring the underlying scene graph representation from a set of noisy bottom-up detections.

Chapter 3

Visualizing Scene Graphs

3.1 Desiderata

Before we're able to implement our possible scene graph models, Visualizing the richly structured geometric information contained in a generative scene graph model is vital for proper debugging and analysis of modeling and inference programs. The most basic requirement is visualizing the components of our scene graph representation (G, Θ, Z). It is important to be able to clearly distinguish each part of our representation; as we see later, incorrect behavior in any part of our model or inference can introduce huge variation in the posterior and inferred approximation. Observed neural detections can be visualized as a special-case scene graph with no edges. As such, this covers a point-estimate view of the majority of addresses in the scene graph models we explored.

On top of this, we'd also like to capture some information about distributions over scene graphs. Providing an clearly interpretable view of all possible distributions $p(G, \Theta, Z)$ is a huge task, so we restrict ourselves to sample-based approximations of unimodal distributions over continuous parameters, and relatively small numbers of structures. Within this class of scenes, common characterizing features are the mean and uncertainty, so our visualizations should provide a clear view of one or more of these features.

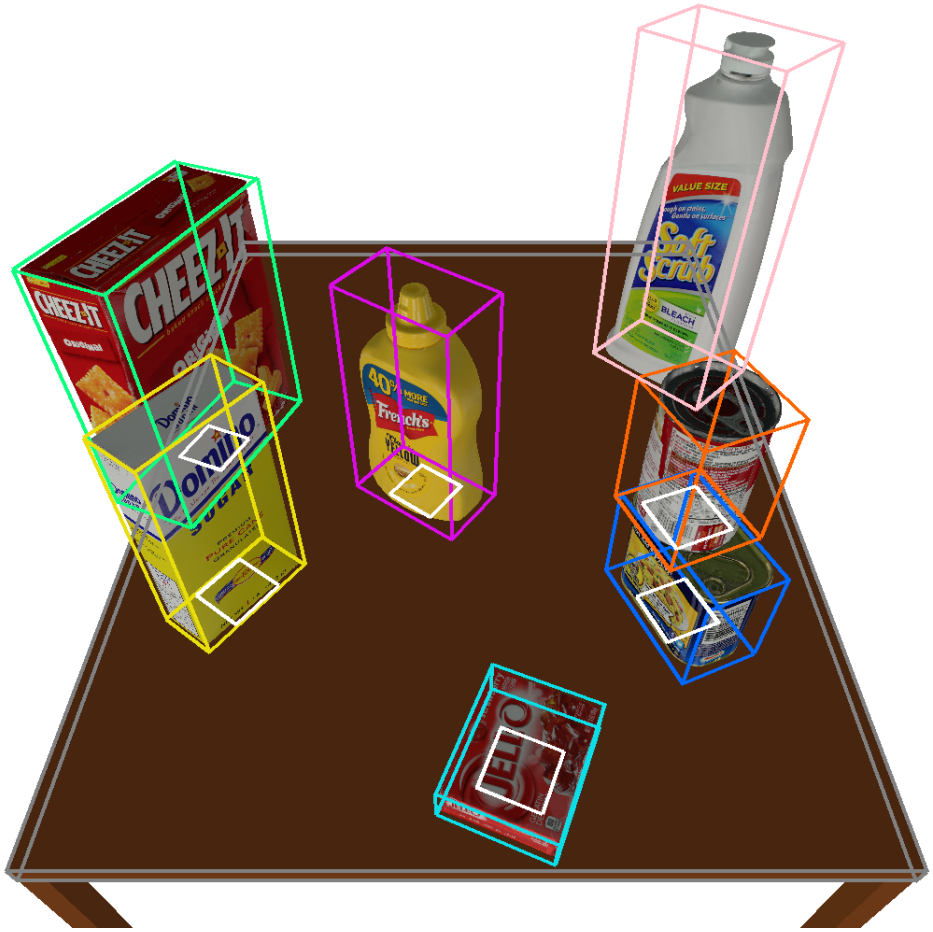


Figure 3-1: Visualization of a synthetic scene with its abstract scene graph overlaid.

3.2 Examples

3.2.1 Visualizing a Single Scene Graph

Our main utility is a function that accepts an image, a scene graph (G, Θ, Z) where $G = (V, E)$, and a camera configuration, and renders that scene graph as a wireframe representation overlaid on top of the image, as viewed from the provided camera specification. Figure **??** demonstrates using this function to see a synthetic rendered scene's underlying scene graph. For each object $o \in O$ in the scene graph, the function renders a colored wireframe bounding box with the dimensions of o, and 6DoF pose given by x_o. Importantly, this is the *absolute* pose of the object, including any slack in relative contacts. Thus, the continuous parameters $\Theta := \{\theta_e\}_{e \in E}$ are not explicitly used in rendering, but can be distinguished in contacting objects $i, j \in O$

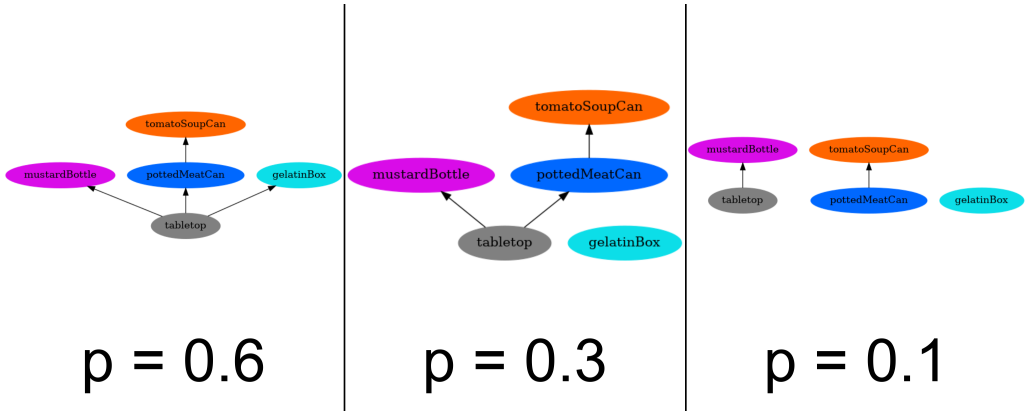


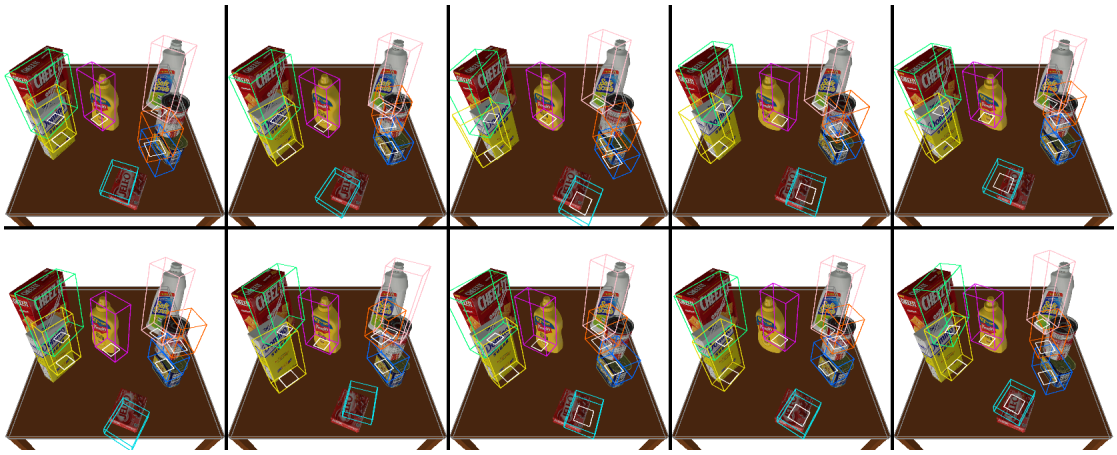
Figure 3-2: Visualization of a distribution over abstract scene graph structure using Graphviz.

by the distance in their corresponding wireframe renderings. Contact edges $e \in E$ are visualized as white squares, located at the center of face f_i. Face f_j is not directly visualized, but can often be inferred from which face is closest to f_i. This does not preclude potential ambiguity for certain pathological slack terms that rotate i close to a multiple of $\pi/2$. However, in practical modeling applications we find such occurrences to be rare, as sensible priors weight heavily against large slack terms.

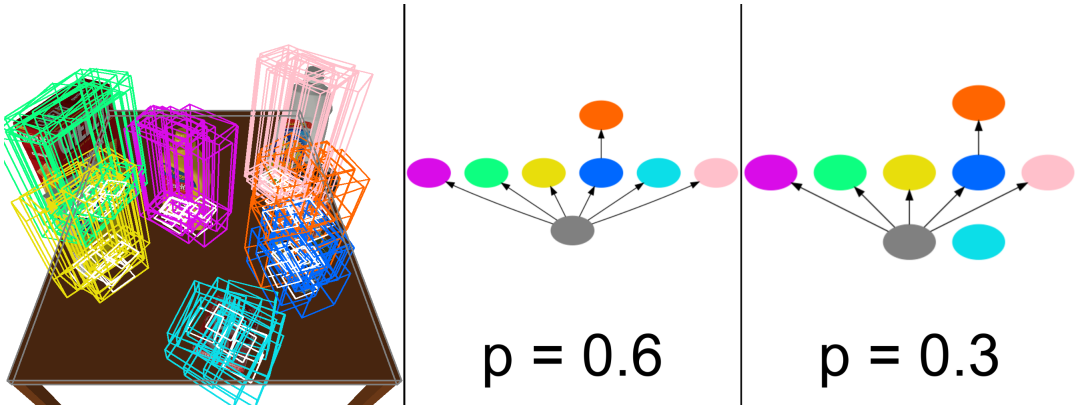
3.2.2 Distributions over Structure Beliefs

As we will quickly see, using the utility shown in Figure 3-1 quickly runs into issues displaying multiple structures over the same image. Since edges are rendered roughly in the same place, it can be hard to tell exactly how many samples have an edge between any two given objects. Even more, it's virtually impossible to tell which vertices and edges are part of the same scene graph. We develop a utility based off the Graphviz [**?**] graph visualization library to more clearly view different discrete structures. It accepts a distribution over scene graphs, and renders a specified number of the most probable present in a scene. In Figure 3-2, we can see how this clearly represents the top competing hypotheses for scene graph structure. This tool can be combined with the wireframe overlay to give rich visualizations of distributions over scene graphs.

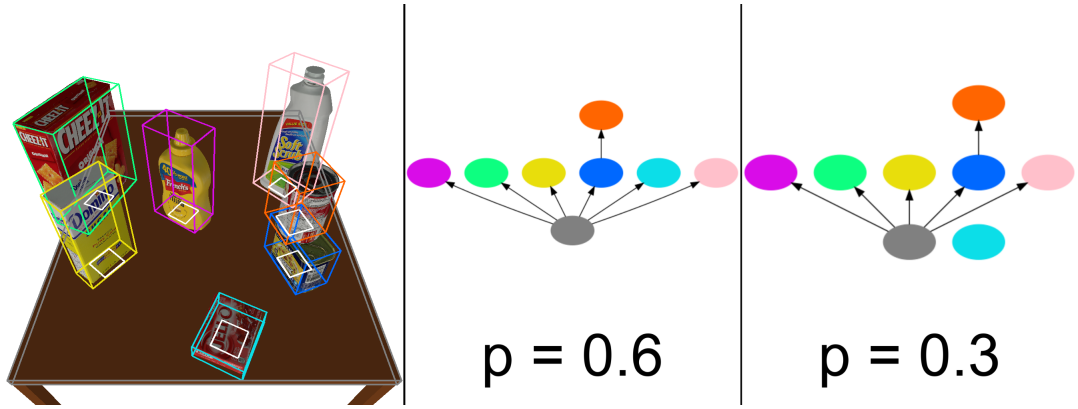
3.2.3 Distributions over Scene Beliefs



(a) Visualization of each sample's state using our scene graph visualization utility.



(b) All 10 samples, rendered over the same image. Only the two most frequent structures are shown.



(c) Aggregated visualization of all the samples. Only the two most frequent structures are shown.

Figure 3-3: Various methods for visualizing multiple beliefs. Node labels are omitted in the rendered structure distribution, and instead the wireframe rendering and GraphViz node for an object o share the same color.

We can now combine our utilities to render rich views of multiple samples. To generate our example visualization distribution, we generate 10 scene graphs $\{\mathcal{G}_i\}_{i\in 1:10}$, with object parameters sampled from a uniform distribution centered at the true scene graph as seen in Figure 3-1. Figure 3-3a shows the simplest form of visualizing the collection of samples, as a collage of separate renderings of each sample. This provides the most information, but at a glance it's difficult to get a sense of the mean and uncertainty of the distribution its representing.

We instead try aggregating the separate samples into a single representative "belief" $\mathcal{G}^* = \text{agg}(\{\mathcal{G}_i\}_{i\in 1:N})$ that is then overlaid on top of the rendered scene. We can then composite this with the abstract structure visualization to improve our ability to see uncertainty in scene structure.

One possible aggregation method is simply taking the disjoint union $\text{agg}(\{\mathcal{G}_i\}_{i\in 1:N}) = \bigsqcup_{i=1}^{N} \mathcal{G}_i$. As we see in Figure 3-3b, this has the effect of overlaying all of the beliefs on top of the same image. This can give a sense of the amount of uncertainty in object positions and rotations, but obscures the underlying object, making it difficult to determine the accuracy of the pose beliefs.

The second method we try $\text{agg}(\{\mathcal{G}_i\}_{i\in 1:N}) = (G^*, \Theta^*, Z^*)$ first takes the most probable structure $G^* = \text{argmax}_G\ \hat{p}(G, \Theta, Z)$ among our samples. The discrete parameters Z^* are then selected arbitrarily from one of the graphs that has this structure. We then let the continuous parameters Θ^* be determined by the average of that scene's objects' absolute poses $x_o^* = \text{avg}(\{x_{(i,o)}\}_{i\in 1:N})$ across all other samples (the average position uses a simple sum, while the average orientation is calculated using the method described by Markley et. al in [**?**]). Figure 3-3c shows an example of this method. In contrast to the Figure 3-3b, this method clearly shows the accuracy of pose beliefs, but loses information about uncertainty in the continuous parameters. These trade-offs mean choosing a visualization is somewhat contextually dependent on if the accuracy or uncertainty is more relevant.

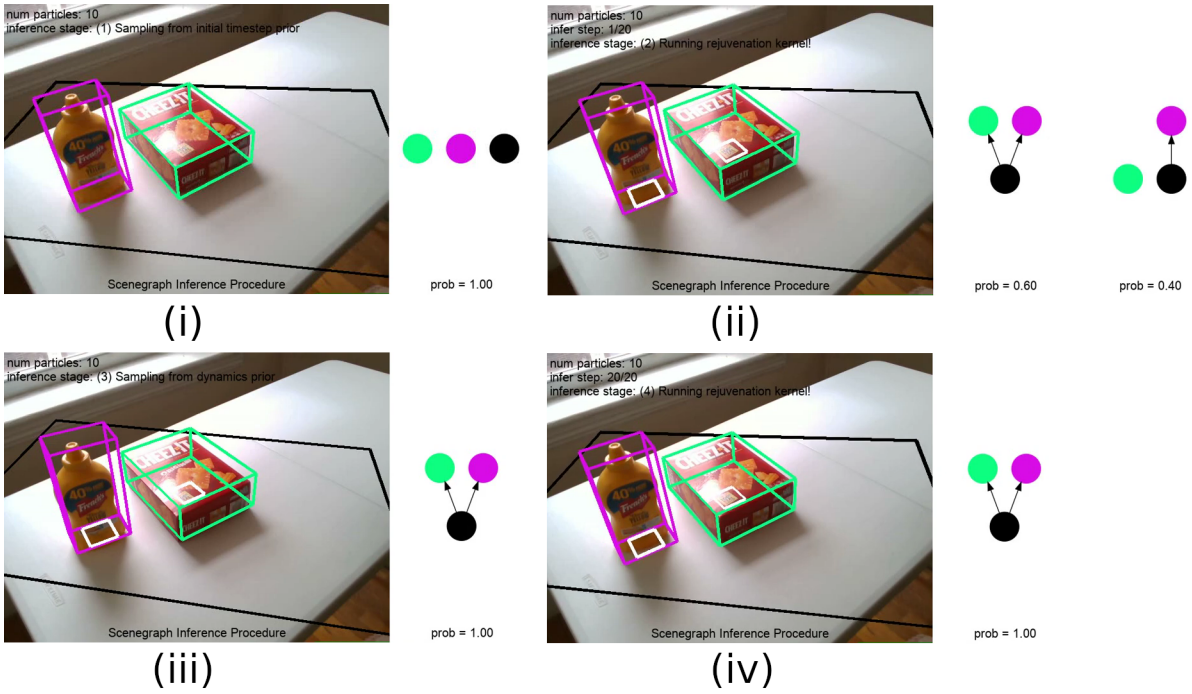


Figure 3-4: Visualization of sampling and rejuvenation moves in a particle filter. A complete video of this scene can be seen at: `https://www.youtube.com/watch?v=0_0TvrGC65Q`. (i) shows the first time step with the overlaid beliefs from the initial time step prior (initialized to the observed neural pose estimates), before structure inference. (ii) shows the beliefs over the first time step, after running 1 step of the rejuvenation kernel. (iii) shows the beliefs over the second time step, after sampling from the dynamics model. (iv) shows the beliefs over the second time step, after running 20 steps of the rejuvenation kernel.

3.2.4 Visualizing Inference in a Particle Filter

Finally, we demonstrate a real-world application by showing the actual usage of these visualization utilities in a particle filter with a corresponding complex inference program. The scene graph model we used leverages the prior from TODO: cite prior scene graph model section and dynamics from TODO: cite scene graph model dynamics. Object poses are initialized to their observed neural detections in the first time step. The inference program leverages a rejuvenation MCMC kernel after sampling from the prior for the first and second time step. This kernel is in turn composed of an interleaved RJMCMC kernel (see TODO: cite RJMCMC section) for discrete structure, and a drift kernel for continuous parameters. We run the

particle filter with 10 particles, and 20 iterations of the rejuvenation kernel. Figure 3-4 leverages the visualization method seen in Figure 3-3c to aggregate the particle filter state into a stable visual estimate of the mean object poses, and to view the distribution over inferred structure. This example shows how we can leverage the methods introduced in this chapter in practice for visualizing and debugging complex inference programs.

Chapter 4

Analysis of Posteriors and Inference by Enumeration

Part of the power of Gen's design is that it facilitates the easy implementation of very complex generative programs with great expressive capability. However, this also incurs an increased cost in development time and difficulty of analysis of the behavior of large models. Predicting the qualitative behavior of a posterior distribution can be difficult even in somewhat simple programs. In a model as complex as a generative scene graph program, it becomes essential to carefully check and analyze the behavior of the posterior to ensure it is a good representation of the data. We conduct a set of synthetic experiments that carefully inspect isolated aspects of our model to ensure their posteriors exhibit reasonable behavior that conforms to intuition.

4.1 Enumerative Inference

In conducting an analysis of a generative scene graph program, we would like to inspect the posterior distribution under different scene configurations. Calculating the full posterior is intractable (indeed, this is why we have to perform inference in the first place). However, by conditioning the model on specific settings for most of the latent values, we can reasonably enumerate over a small set of free variables, and plot the posterior distribution over a *projection* of our model to a lower dimensional

slice. We note that this approach is limited in its ability to examine interactions between variables, and in generality the posterior of the full model can exhibit very different behavior from isolated projections to lower dimensions. Nonetheless, we show that these projections can still be a rich source of information for understanding and improving scene graph priors.

4.2 Structure Posterior

The first analysis we conduct looks at the model's predictions for structure between objects, as a function of the observed vertical offset y between their closest faces. We consider the static model and dynamic model respectively, and look at the behavior of the structure posterior under multiple settings of their hyperparameters. We demonstrate the high variability of the qualitative behavior of the posterior under different settings for hyperparameters; this in turn shows the vital importance of proper hyperparameter tuning in ensuring the correctness of scene graph models.

4.2.1 Static Model

We set up an experimental scene with two objects, to visualize the enumerated structure posterior for G. given an observed vertical offset y between the object's closest faces. We restrict our latent parameters to be the same as the observed poses, and for the two observed poses to be the same, except for a one-dimensional vertical offset y between their closest faces. We set the structure prior to be uniform, so all structures have equal prior probability. Recall the slack offset model is a normal distribution with standard deviation σ_{slack}.

Figure **??** shows the posterior probability that G is a "sliding" configuration, given an offset y, for a few different settings of σ_{slack}. Predictably, the wider the distribution on slack, the larger a gap the model is willing to allow while still classifying the objects as "sliding". We also note the relatively sharp transition in the sliding probability from almost 0 to almost 1, at a discrete cutoff point; as we will examine later this could be a result of a poor model for our slack variable.

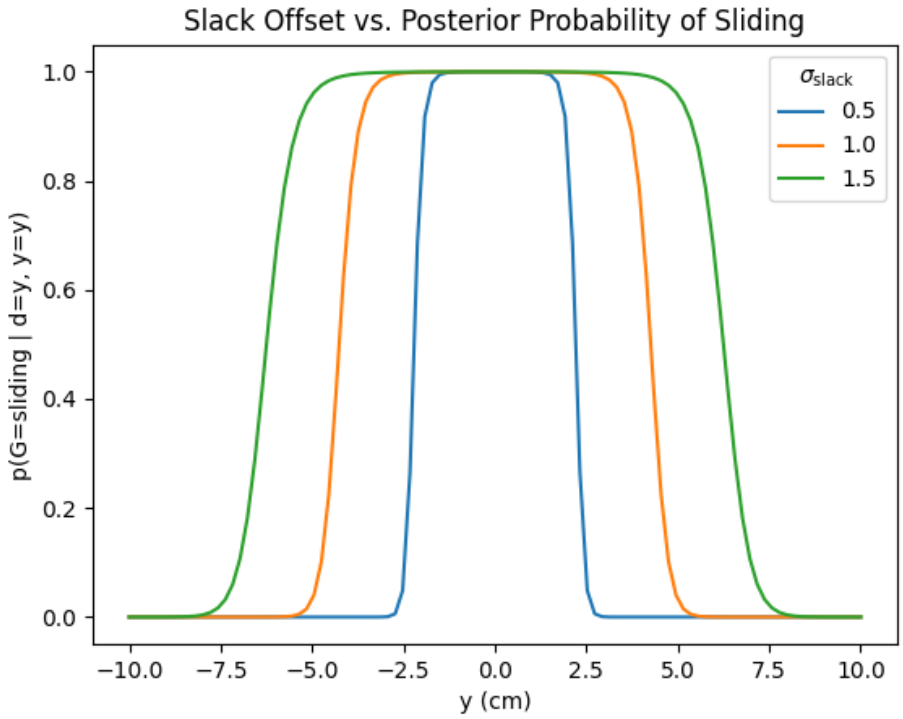


Figure 4-1: Probability that an object is sliding in a static scene with different settings of σ_{slack}, and observed/latent vertical offset y.

4.2.2 Dynamic Model

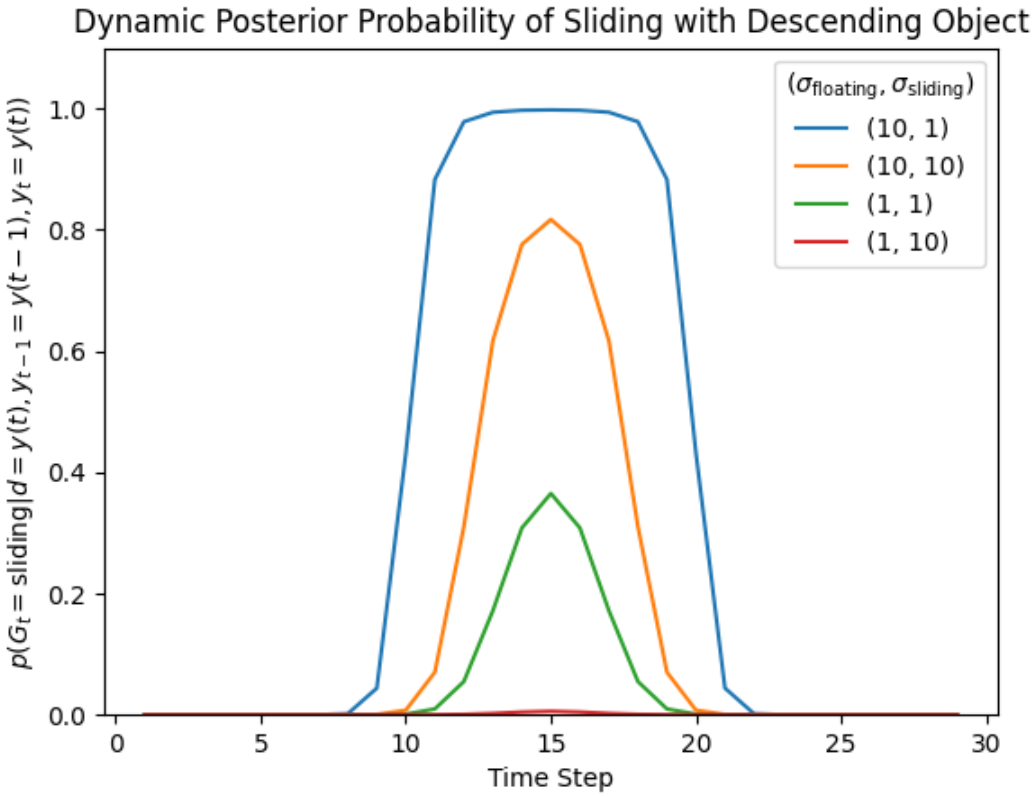


Figure 4-2: Probability that the scene structure G_t is in a "sliding" configuration at time t, in a dynamic scene where the object's observed position and slack offset is given by $y(t) = 10 - 0.67 \cdot t$ cm, with hyperparameters $(\sigma_{\text{floating}}, \sigma_{\text{sliding}})$.

The second experimental scene is very similar to the first, except we introduce a dynamic element to examine the effect of the dynamics parameters. Across 30 time steps, the top object (and its observation) is lowered along the vertical offset from 10 cm above the bottom object, to 10 cm interpenetrating with the bottom object. For simplicity, we again set the prior structure at each time step to be uniform and temporally independent. We set $\sigma_{\text{slack}} = 1cm$, and recall that the floating position dynamics are a 3D normal distribution with standard deviation σ_{floating}, while the sliding dynamics are a 2D normal distribution with standard deviation σ_{sliding}.

Figure 4-2 shows the posterior probability that G_t is a "sliding" configuration, given the observed/latent offset y, for each time step t of the dynamic scene. Adding

dynamics has added completely new behavior to the structure posterior, dependent on the additional hyperparameters. The pose displacement is always 2/3 cm every time step, meaning the most accurate dynamics model in this figure is the one with the tightest distribution. The change in posterior probability in sliding structure is a consequence of the Bayesian Occam's Razor; the posterior probability for sliding is greatest (blue curve) when the sliding dynamics model is much more confident (concentrated) than the floating dynamics model, and is least in the opposite case (red curve). This is our first encounter with the Bayesian Occam's Razor adding complications to the development of scene graph models, but as we shall see there are several cases where the concentration of our different models have a huge impact on the behavior of the posterior.

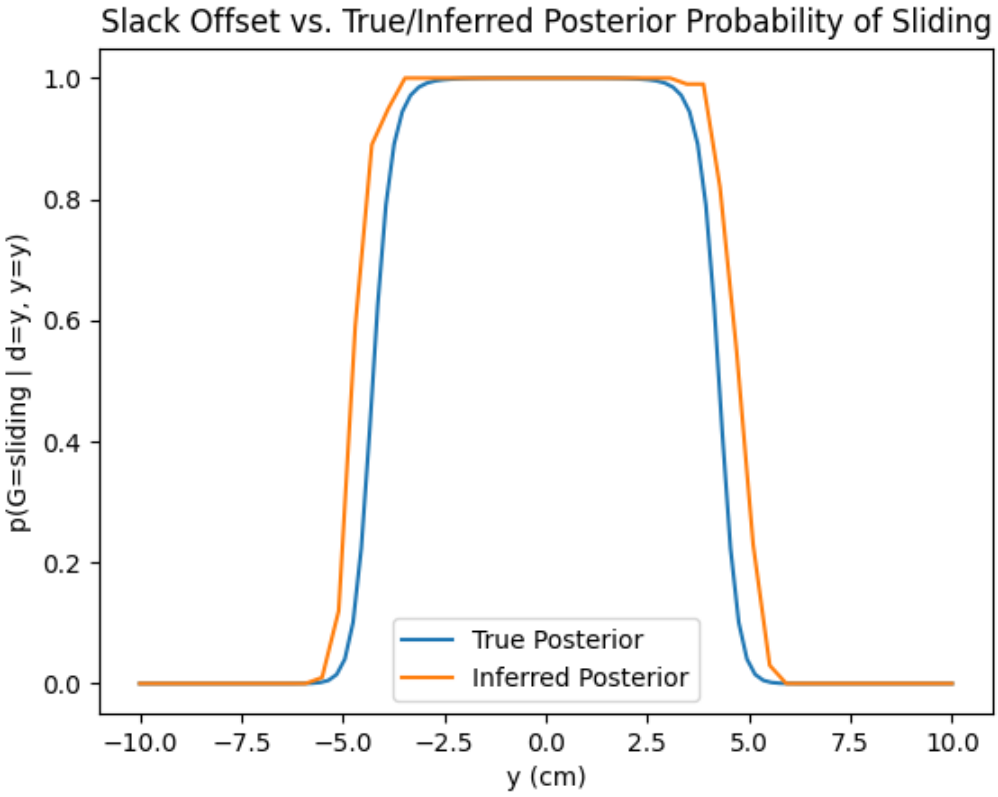


Figure 4-3: RJMCMC structure inference results. For each value of d, we infer the approximate probability of sliding as the average structure of 100 particles run with 3 iterations of our RJMCMC algorithm applied to the top object.

4.2.3 RJMCMC Structure Inference

Finally, we include a brief comparison of our RJMCMC inference procedure to the enumerated posterior. We replicate the scenario shown in Figure 4-1 with $\sigma_{\text{slack}} = 1$ cm, and overlay the inferred posterior probability of a sliding structure. As we can see, RJMCMC accurately recovers the posterior in this situation, which is a successful first test of the correctness of our algorithmic implementation.

4.3 Contact Slack Model

The second analysis we conduct examines the enumerated posterior over the model for slack offset d. Having a good slack model is important for having a good structure model; if the slack model provides a poor explanation for any observed gap between "contacting" objects, the structured model will also provide a poor explanation for the scene. Recall our prior model over slack is $p_{\text{norm}(0,\sigma_{\text{slack}})}(d)$.

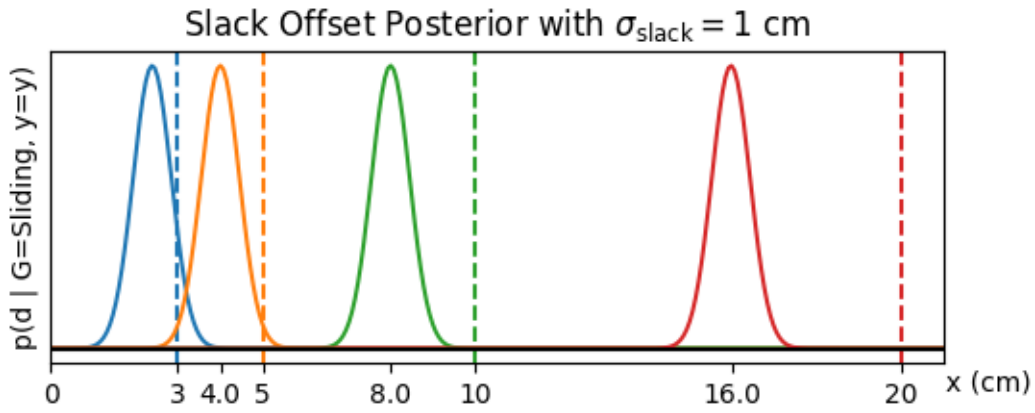


Figure 4-4: Posterior distribution on slack offset d. The dashed lines represent the set of observed y values, while the curve of the same color represents the slack offset posterior, given the observed y and sliding structure.

4.3.1 Varying the Gap between Sliding Objects

We set up an experimental scene with two objects, to visualize the enumerated slack offset posterior for d, given a fixed sliding structure. We restrict our objects to share the same observed pose, except for a one-dimensional vertical offset y between

their contacting faces. In our prior model we set $\sigma_{\text{slack}} = 1$ cm and $\sigma_{\text{inlier}} = 0.5$ cm. Figure 4-4 shows the posteriors corresponding to various settings of the observed offset y. Because our $p(d|G = \text{sliding}, y = y) = p_{\text{obs}}(d|y = y) \cdot p_{\text{slack}}(d)$ is a product of normal distributions, the posterior is also a normal distribution with mean in-between the mean of the two model distributions. This means the most probable gap between two objects given a sliding structure is counter-intuitively in-between 0 and the observed gap y; this may explain why we see such sharp cutoffs in the probability of a sliding structure in Figures 4-1 and 4-2. As the observed gap between two objects becomes larger, the model over slack offset quickly becomes a poor explanation of the observation. A more reasonable model would place high mass around $d = y$, as this would explain the observation most accurately.

4.3.2 Hyperprior over Slack Standard Deviation

To improve the model, we consider adding a hyperprior over σ_{slack}, to more accurately allow for uncertainty in the size of the slack term. We introduce an exponential distribution with rate parameter $\lambda = 0.2$ over the slack standard deviation $\sigma_{\text{slack}} \sim \text{Exp}(0.2)$. Figure 4-5 shows the joint posterior over σ_{slack} and the slack offset d. The figure shows that the mode of the posterior is located much closer to the observation than without a hyperprior over σ_{slack}. Indeed, as we see in Figure 4-5, the posterior exhibits much more reasonable behavior when letting $\sigma_{\text{slack}} = \sigma^*_{\text{slack}}$. We conclude that adding a hyperprior over slack is a viable way to increase the accuracy of the model with a sliding structure, especially when the observed gap between objects is large.

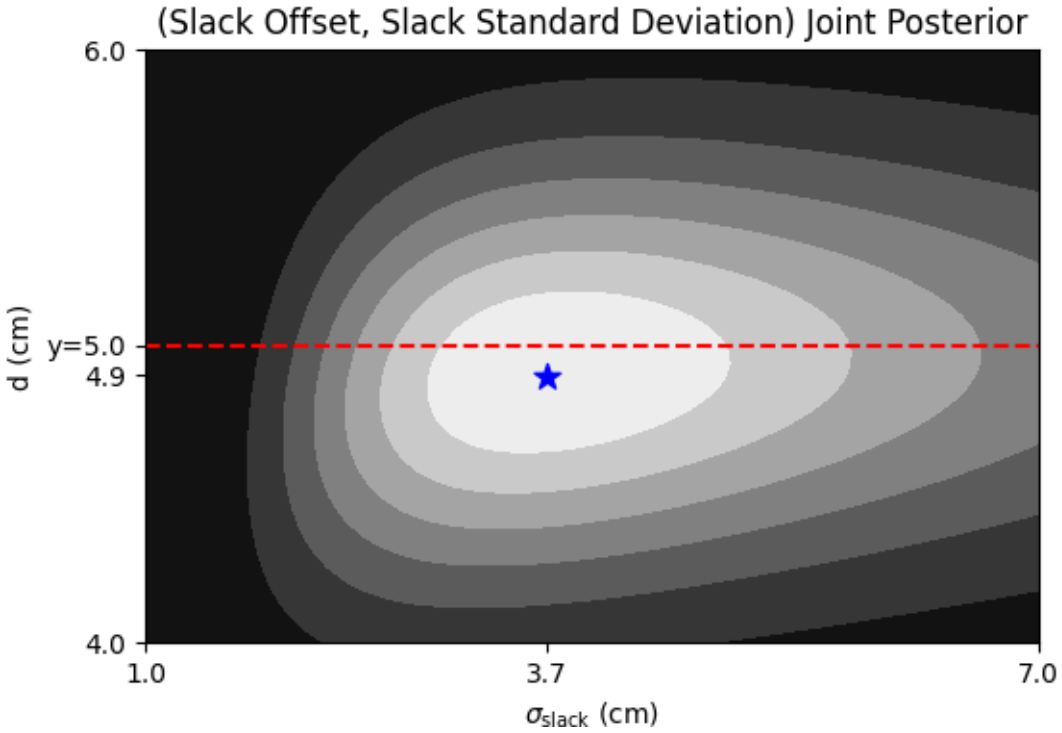


Figure 4-5: Joint posterior $p(d, \sigma_{\text{slack}}|y = 5.0)$. The dashed red line indicates the observed position $y = 5.0$ of the top object. The blue star indicates the maximum a posteriori $(\sigma^*_{\text{slack}}, d^*)$.

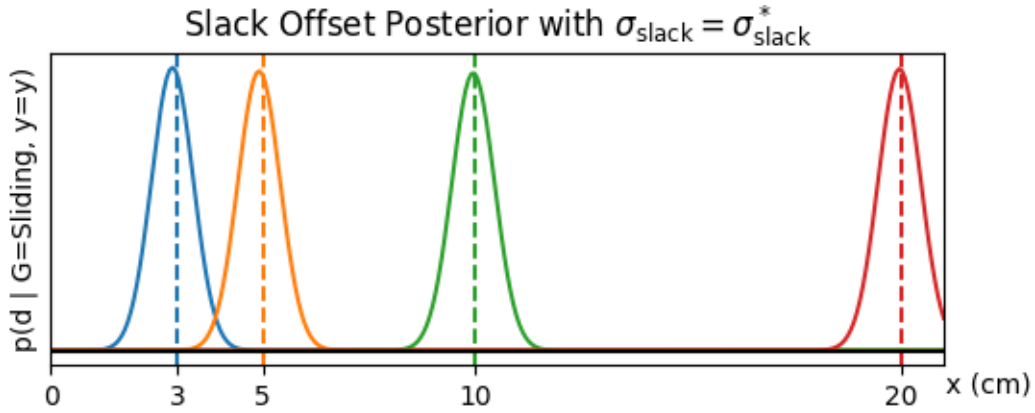


Figure 4-6: Slack offset posterior for d, conditioned on $\sigma_{\text{slack}} = \sigma^*_{\text{slack}}$. The dashed lines represent the set of observed y values, while the curve of the same color represents the posterior over slack given the observed y and sliding structure.

4.4 Neural Outlier Model

A big part of our model is leveraging a bottom-up feature detector like a neural network to leverage A simple form of intuitive physical reasoning we model is by discouraging large changes in distance over a single time step via our normally-distributed dynamics model. Our prior observational model is a simple mixture of a uniform outlier distribution over poses in a finite bounded box region, with component $p(\text{outlier}) = 0.1$, and a direct product of a normal and VMF inlier distribution, with component $p(\text{inlier}) = 0.9$. Ideally, the product of this model and our dynamics model would allow our scene graph model to confidently predict that large leaps in our latent position over a single time step are less probable than that specific observation being an outlier "flicker" detection, sampled randomly within the scene.

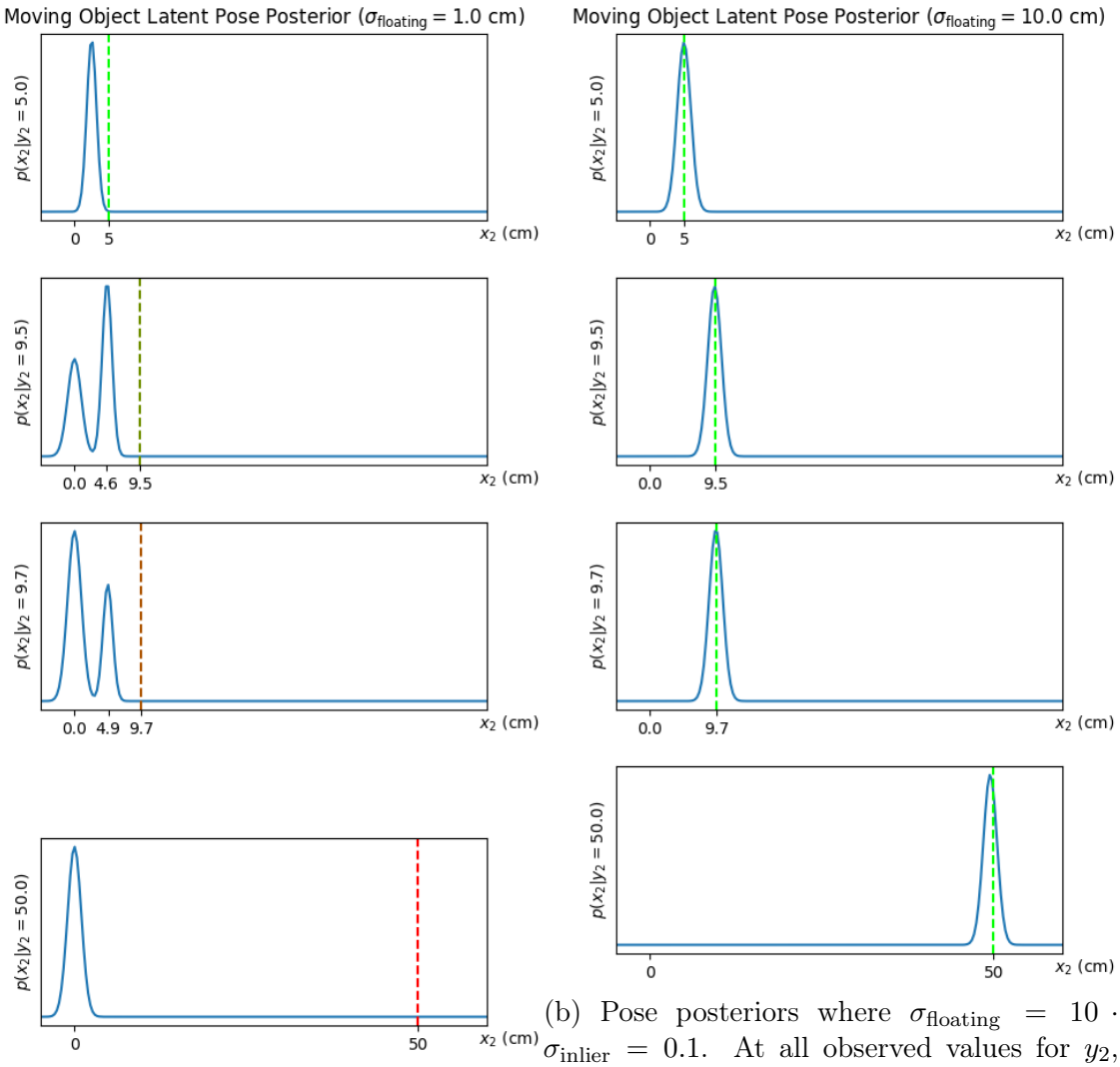
Figure 4-7 shows a scene in which an object's observed pose jumps from a latent position at the origin, $x_1 = 0$, to one located a distance y_2. On the left, when the dynamics and observational model are equivalent width, the posterior predicts that the detection is more likely to be an outlier than not at around $y_2 = 9.6$. On the right, when the dynamics is significantly more broad than the observational model, the posterior predicts that the detection is more likely to be an inlier, all the way up to $5 \cdot \sigma_{\text{floating}} = 50$ cm away from the previous position. The prior probability that an object travels a distance than $5 \cdot \sigma_{\text{floating}}$ away from the mean is less than $5.73e - 7$, which is dramatically less than the prior probability of an object being an outlier $p(\text{outlier}) = 0.1$.

The model on the right is counter-intuitively claiming that the object made an extraordinarily unlikely jump. If we examine the log probability density functions evaluated at the modes of the latent pose joint distributions, for the inlier and outlier

models respectively, we begin to see why:

$$\begin{aligned}
\log p(x_2 = 50, y_2 = 50, \text{inlier} | x_1 = 0) = \\
\log p(\text{inlier}) + \\
\log p(x_2 = 50 | x_1 = 0, \text{inlier}) + \\
\log p(y_2 = 50 | x_1 = 0, x_2 = 50, \text{inlier}) = \\
-0.105 - 1.783 + 17.625 = 15.737 \\
\log p(x_2 = 0, y_2 = 50, \text{outlier} | x_1 = 0) = \\
\log p(\text{outlier}) + \\
\log p(x_2 = 0 | x_1 = 0, \text{outlier}) + \\
\log p(y_2 = 50 | \text{outlier}) = \\
-2.303 + 10.717 - (3 \cdot 0. + 2.289) = 6.125
\end{aligned}$$

The dynamics model's density weights heavily in favor of the observation being an outlier. However, the inlier pose distribution is so tightly concentrated compared to the outlier pose model, that it dominates the calculation of the overall model weight. The posterior tells us that the model would rather have the latent pose make an incredibly unlikely jump, if it means it can explain the observed pose confidently using the inlier observational model. This is another occurrence of the Bayesian Occam's Razor, where the most concentrated component of the model (in this case the observation model), determines where the majority of the posterior mass is distributed.



(a) Pose posteriors where $\sigma_{\text{floating}} = \sigma_{\text{inlier}} = 0.01$.

(b) Pose posteriors where $\sigma_{\text{floating}} = 10 \cdot \sigma_{\text{inlier}} = 0.1$. At all observed values for y_2, the model strongly classifies the detection as an inlier.

Figure 4-7: Posteriors over the latent pose of an object $p(x_2|y_2) = p(x_2|y_2, x_1 = 0)$, under two different settings of the dynamics parameter. The noisy observational model's inlier distribution is given by $\sigma_{\text{inlier}} = 0.01$. The prior probability of an outlier is 0.1. The dashed line represents the observed pose y_2, with color representing the probability that y_2 is an outlier (totally green means $p(\text{inlier}) = 1$, while totally red means $p(\text{outlier}) = 1$).

4.5 Lessons for Improving Scene Graph Priors

We can take three big lessons from these experiments regarding the improvement of our scene graph model. The first lesson is that the qualitative behavior of the posterior is highly sensitive to the proper tuning of hyperparameters. As we saw in the structure posterior analysis, hyperparameter settings had a crucial impact on the performance of the model. Furthermore, we saw how introducing dynamics over continuous parameters impacts the posterior probability over discrete structure. Interactions like these can be highly counter-intuitive, and difficult to explore using synthetic tests like we've done here. It's for this reasons that we believe in the future development of scene graph models, training model hyperparameters and analyzing model performance using real-world data will be a vital step in improving accuracy. Next chapter we explore real-world data sets, and how they can be used to test and improve scene graph models.

The second lesson is the importance of considering how the *relative* concentrations of different component distributions in the model will affect the posterior. More specifically, the Bayesian Occam's Razor suggests that distributions with very high concentrations of their mass will tend to have an overinflated impact on where mass ends up in the posterior. In the simplest case, the relative concentration of distributions in different model structures have a large impact on which model structures are more probable in the posterior. In the extreme case, this can lead to entire parts of the prior model being neglected in favor of assigning high weights to areas that maximize the density of the especially peaky model component. Uur scene graph model has a large family of possible model structures, with different continuous parameter distributions of varying concentration and dimension. We cannot emphasize enough that *all* of these distributions, and their relative concentration, matter for the calculation of the structure posterior, and thus must be considered *jointly* in order to have a truly accurate understanding of the posterior in the full model.

Chapter 5

Learning and Testing from Real-World Data

Enumerative inference on synthetic traces can provide low-level confirmation about basic properties of our model, under simple qualitative conditions. However, real-world analysis provides a much richer set of possibilities in terms of analyzing and improving our model. We collect a set of real-world videos containing common household items from the YCB object dataset, along with annotated values for all of the variables in the static (single frame) version of our model. We then use this data in a gradient search to train a subset of hyperparameters that have thus far been hand-selected through trial-and-error, but may not be properly tuned to model real-world data. We also measure the performance of our proposed structure inference algorithm on predicting the ground-truth structure from a set of object poses.

5.1 Data Specification

We collect a set of experiments with static objects to analyze the model in the static case. We add noise via occluders that temporarily block the camera's line of sight to the objects. The raw video was collected using the Intel RealSense D435 RGB/Depth camera. Only the RGB information is used for the neural detector, but we collect depth data to enable plane detection, and for future tests on models containing

likelihood over depth data. Observed poses are generated using the publicly provided nVidia DOPE detector GitHub repository. Ground truth outlier status is manually annotated for each frame in the collected videos. Note that using the vertices, we can also use the scene graph structure to determine which detections are false positives or negatives.

Ground truth structure is manually annotated once per scene, and the ground truth poses are estimated as the average over all an object's detected *inlier* poses. While this obscures systematic bias across all neural detections in the data set, in nonetheless provides a useful relative analysis of the stability of the neural detector in the presence of varying levels of occlusion. This data set generation process is easily scalable, as all it requires manually is the collection of the video, a once-per-scene annotation of the static ground truth structure, and a per-frame annotation of which neural detections were decently well-localized inliers. Table 5.1 lists the data collected for each scene, while Figure 5-1 shows some representative samples from our data set.

Name	**Data**	**Description**
Observed poses	(String, List[Pose])	nVidia Deep Object Pose Estimator neural detections
Ground truth outlier status	(String, List[Bool])	Per-Frame, per-object flag indicating if an object's neurally generated pose estimate is a noisy outlier
Ground truth scene graph	Scene Graph	Observed static structure, and object poses estimated as the average of inlier detections.

Table 5.1: Description of data collected in our real-world experiments with YCB objects.

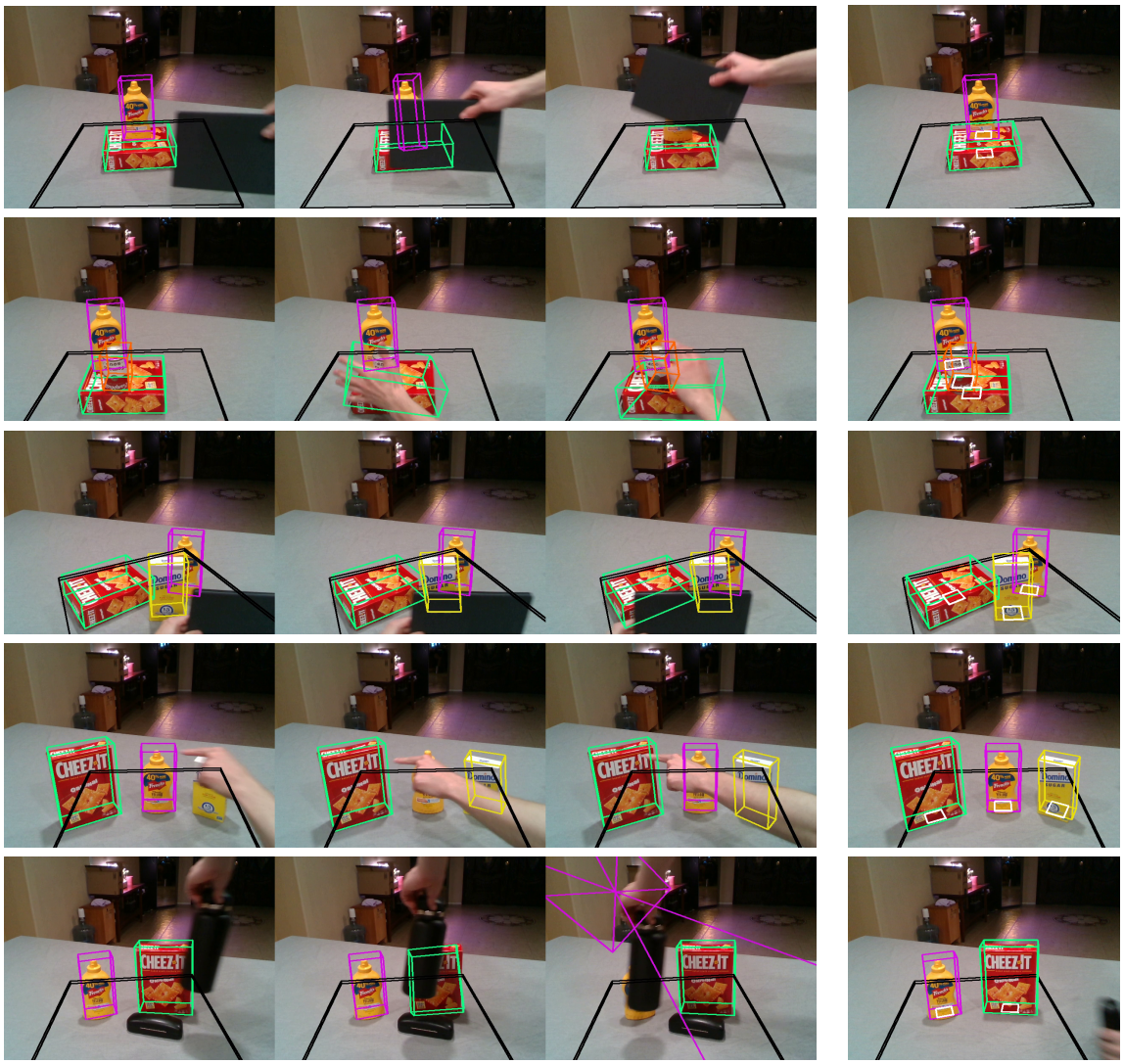


Figure 5-1: A selection of frames from 5 of the 8 captured videos. (Left) We use a dynamic occluder to generate noise in the observed neural network detections, in which objects have their poses estimated incorrectly, or even flicker out of existence. (Right) Each scene is represented by a single static scene graph, with manually annotated structure, and object poses estimated as the average of the inlier neural detections.

5.2 Metrics

5.2.1 Modeling: Marginal Likelihood

The most immediate metric measuring the evidence for our model given the data is the log likelihood. Since we've collected complete data for our model, the likelihood simply ends up being the joint probability density of the trace. In Gen, given a collection of traces $t \in \mathcal{T}$ and hyperparameters θ, the log-likelihood is:

$$\ell(\mathcal{T}; \theta) = \sum_{t \in \mathcal{T}} \log p(t_i; \theta)$$

5.2.2 Structure Inference: Graph Edit Distance

When evaluating the performance of structure inference, we would like a measure of the similiary of the most probable inferred structure to the ground truth. This gives a measure of how far our algorithm is from ground truth when tasked with classifying the structure of an unambiguous scene. In particular, we simplify our metric to how well our algorithm correctly determines which objects in a scene are "contacting". To do this, we reduce the structure of the predicted scene graph G_1 and target scene graph G_2 to a simple set of undirected edges. The graph operations we consider "edits" are edge addition and edge removal. Then, the edit distance is just the size of the symmetric distance $E_1 \triangle E_2$ of the edge sets, or more explicitly

$$\text{Edit}(E_1, E_2) = |E_1 \triangle E_2| = |(E_1 \cup E_2) \setminus (E_1 \cap E_2)|$$

5.3 Learning Hyperparameters with Gradient Ascent

We can perform a data-driven maximum, we can now use gradient-based optimization to automatically tune the static model hyperparameters to our data, and measure the increase in log-likelihood. We select 3 hyperparamters $\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \sigma_3)$ to optimize over: the standard deviation $\sigma_1 = \sigma_{\text{inlier}}$ for the observed 3D positions, the standard deviation $\sigma_2 = \sigma_{\text{slack}}$ for the 1D slack offsets, and the standard deviation

$\sigma_3 = \sigma_{\text{slidingXY}}$ for the latent (x, y) contact parameters. We'd like to find

$$\max \ell(\mathcal{T}; \boldsymbol{\sigma}) = \sum_{t \in \mathcal{T}} \log \ p(t; \boldsymbol{\sigma})$$

To do so, we perform a scheduled gradient ascent over the hyperparameters with respect to the objective function. We initialize $\boldsymbol{\sigma}_0 = (0.01, 0.01, 0.25)$ (meters). We run 1,000 iterations of gradient descent, and for each iteration t we update parameters according to

$$\boldsymbol{\sigma}_t = \boldsymbol{\sigma}_{t-1} + \alpha_0 r^t \cdot \nabla \boldsymbol{\sigma}_{t-1}$$

where $\alpha_0 = 1 \times 10^{-8}$ and $r = 0.999$. Gen's dynamic DSL supports automatic calculation of gradients of the joint density evaluated on the conditioned addresses, with respect to model hyperparameters, and for performing updates on parameters with a provided learning rate. Thus our procedure is mostly automated once we have the set of traces $\mathcal{T}$.

Figure 5-2 shows the results of gradient descent. The algorithm converges rather rapidly for σ_{inlier} and σ_{slack}, but over a longer time we see $\sigma_{\text{slidingXY}}$ begin to converge as well. This longer convergence time suggests that $\sigma_{\text{slidingXY}}$ may have a smaller impact overall on the likelihood compared to the other two parameters.

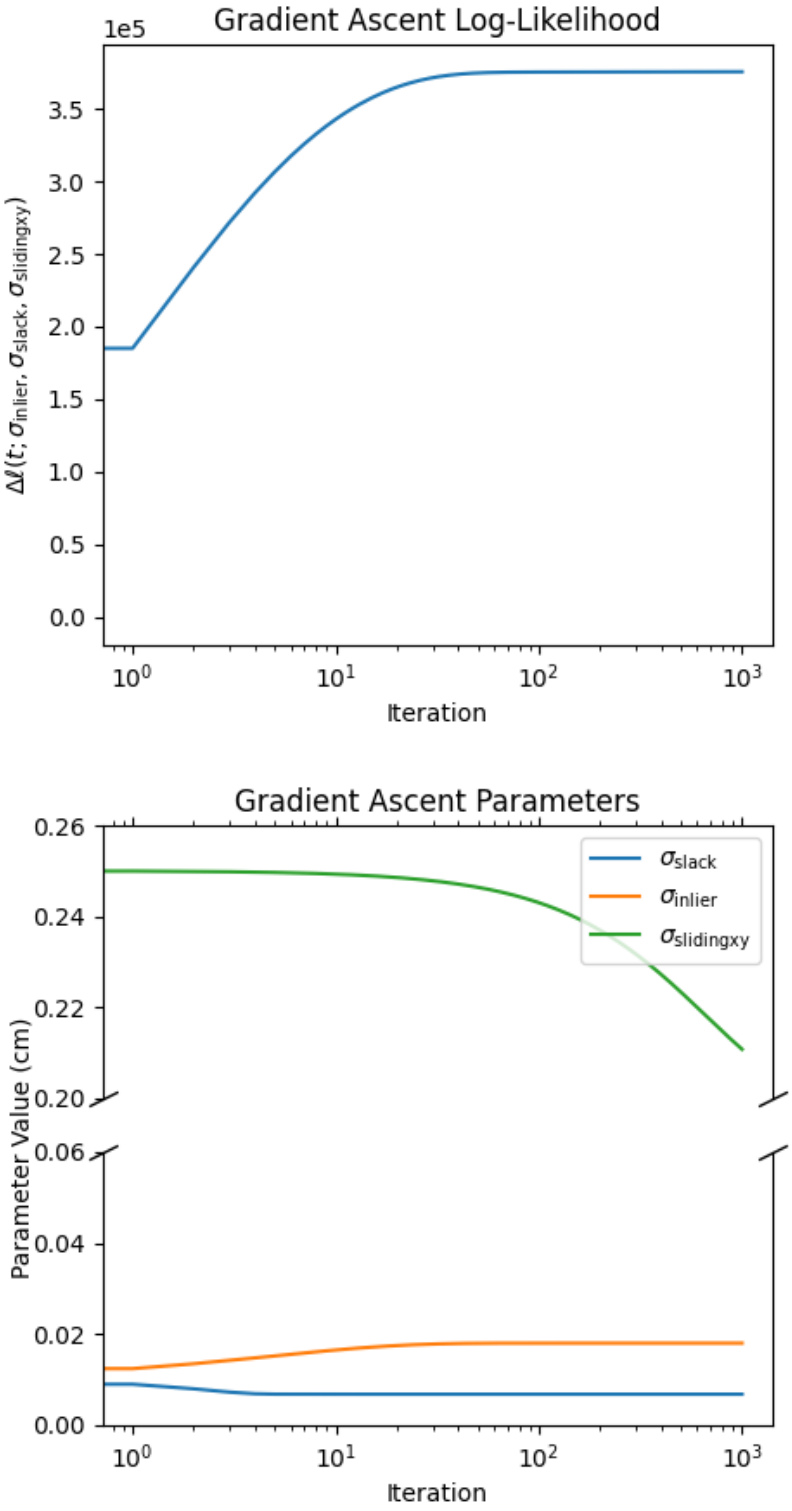


Figure 5-2: Results of gradient ascent on $\boldsymbol{\sigma}$. (Top) shows the change in log likelihood $\Delta\ell(\mathcal{T}; \boldsymbol{\sigma}) = \ell(\mathcal{T}; \boldsymbol{\sigma}_t) - \ell(\mathcal{T}; \boldsymbol{\sigma}_0)$. (Bottom) shows the values of the parameters $\boldsymbol{\sigma}_t$.

5.4 Inlier Detection Observational Model

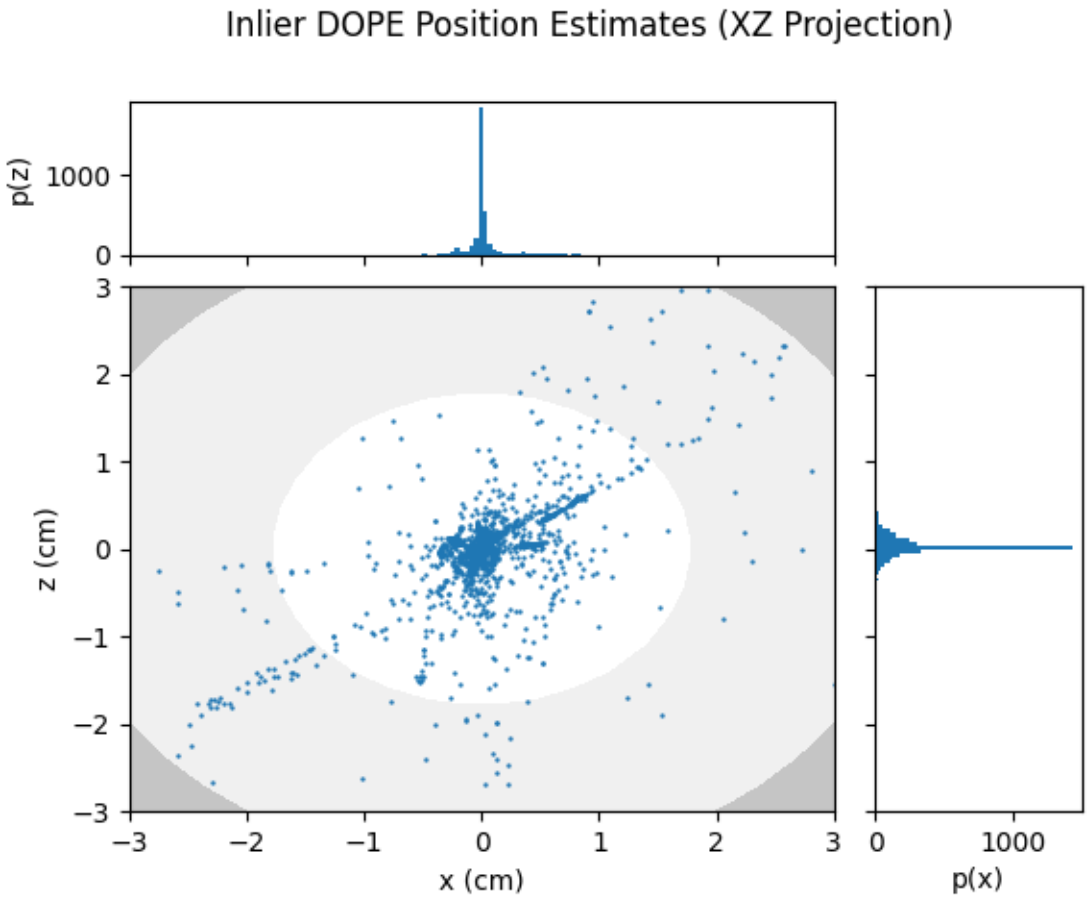


Figure 5-3: Scatter plot of the projection of nVidia DOPE's observed pose estimates to the x and z position axes, along with corresponding marginal histograms for each axis respectively. The plotted contour is the overlaid noisy observation model for inlier detections (in the projection, a 2D multivariate normal distribution), with the standard deviation hyperparameter set to the maximum-likelihood value for σ_{slack} that we optimized for above.

The lowest-level data in our static model are observations of neural object detections. A-priori, we modeled this data as a simple mixture of an "inlier" and "outlier" distribution. We presupposed the noise for detected inlier positions would be normally distributed, with standard deviation σ_{slack}. In the last section, we jointly fit this hyperparameter with two others, which gives us a tuned version of our inlier model.

Figure 5-3 provides a comparison of a simple observational model in the positional dimensions, fit to maximum likelihood with respect to inlier pose detections. We note

a few details from this plot, and what they suggest for the observational model.

The first is that the distribution systematically overestimates the number of observations in the "middle" part of the space. A large number of detected poses are either highly concentrated around the mean detection, or are much further out in the tails of the distribution. A normal distribution may not be the most accurate model of the behavior of the neural detector; a distribution with heavier tails, such as a Cauchy, is likely to better model of the detected positions.

The second is the presence of "clusters" of erroneous detections that suggest failures in the neural detectors are strongly correlated across time. Indeed, the high concentration of poses around the average inlier detection means that there could be substantial correlated systematic error in an object's detection across an entire scene, that is not shown in Figure 5-3. For example, the average inlier pose detection for the Domino sugar box presented on the right side of the fourth row in Figure 5-1 is visibly mis-estimated. Assuming that observational noise is falsely uncorrelated can bias our posterior latent pose to be centered around a systematically wrong detection. It may instead be possible to model the correlation between these errors, and use additional information from the image, like the presence of occluders, to obtain a more accurate estimate of the true object pose.

5.5 Testing Structure Inference

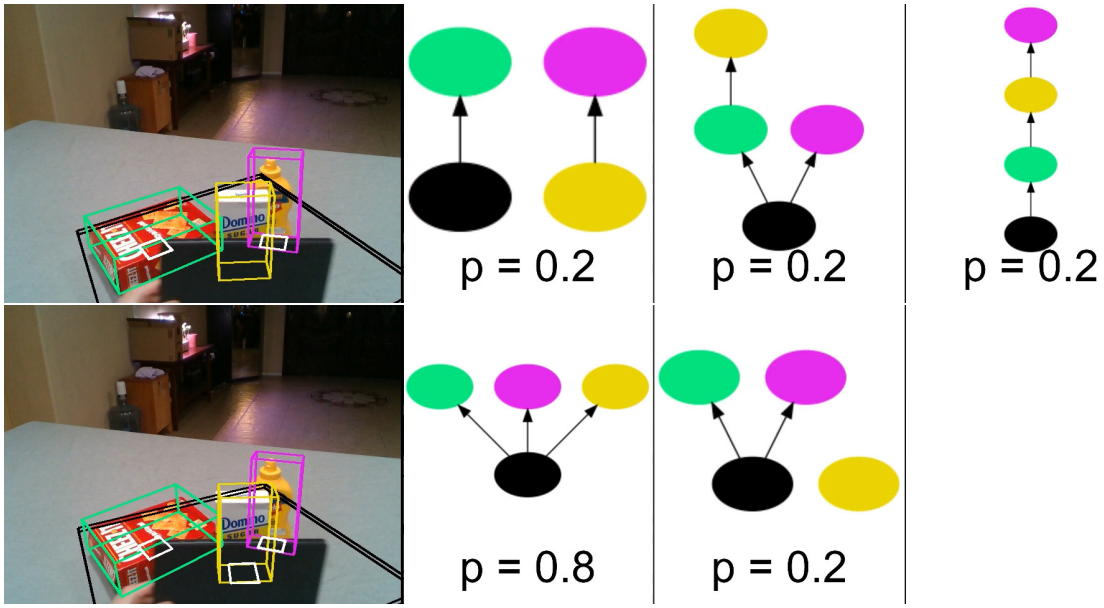


Figure 5-4: Example of the performance on RJMCMC on inferring the structure posterior in a frame from scene #5. (Top) The uniform prior over scene graph structure contains a random assortment of inaccurate classifications. (Bottom) The RJMCMC structure inference procedure predicts the correct ground-truth structure with high probability.

Finally, we use our real-world data to evaluate the performance of our RJMCMC procedure for structure inference, on the posterior for our unoptimized and optimized models. We want to see how well our procedure is able to classify the ground truth structure, given the correct latent poses of the objects in a scene. However, we only have the ground truth poses for 8 unique static scenes, which limits the number of independent trials we can perform.

To ameliorate this, we augment the data with some sampled noise. When performing continuous parameter inference our latent pose posterior will have some non-zero width, meaning we should allow for some variability in the estimated underlying pose, without significantly changing the structure posterior. To simulate this, we augment our data by very slightly perturbing the latent poses for each frame, without qualitatively changing the underlying scene graph that best explains these latent poses;

the position is perturbed by gaussian noise sampled with a standard deviation of 0.5 cm, and the orientation is perturbed by VMF noise sampled with concentration parameter 4000. This adds enough variability to give a more robust test of our structure inference algorithm, especially when comparing the unoptimized and optimized models.

For each frame, we set the static model trace to the underlying scene graph in 5 independent particles. For each particle, we then resample the scene graph structure G and discrete parameters Z from a uniform prior, without changing the absolute 6DoF poses or observations for each object. We then sweep our reversible jump structure move across all objects O for 5 iterations. Figure 5-5 shows the graph edit distance between the ground-truth structure and the most frequent structure among the particles, in the unoptimized and optimized case respectively. Figure **??** shows an example of the prior structure and inferred posterior respectively for a frame in sequence #5. Inference in both cases consistently finds a structure within two edges of the ground-truth, and most often finds the correct structure.

The optimized procedure performs marginally better than the unoptimized case, although the difference is minimal. Given this, and the fact that our hyperparameters didn't change much in the gradient ascent procedure, suggests that our hand-tuned values were already decent to begin with, at least with respect to the structure posterior. Recall that in our synthetic analysis of these hyperparameters shown in Figure 4-1, the posterior probability of an edge between two objects transitions rather sharply between 0 and 1, depending on the distance of their contacting faces. We might then expect that for a range of "decently-tuned" hyperparameter values, the structure posterior would perform roughly the same, with a sharp decrease in model performance once the hyperparmeters fell out of this range.

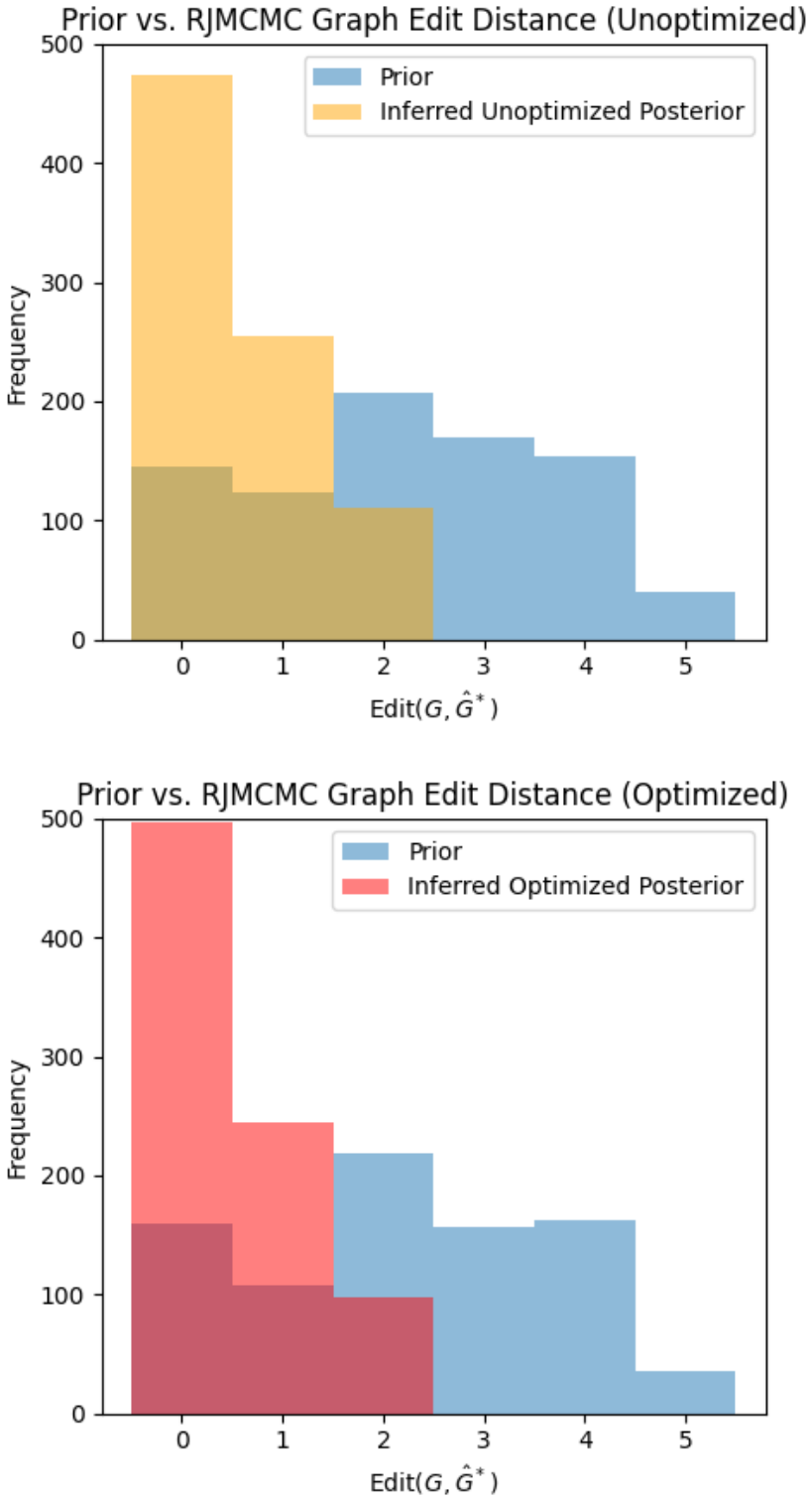


Figure 5-5: Graph edit distance between the ground truth structure G and the mode $\hat{G}^*$ of the inferred structure posterior, for the unoptimized model (top) and optimized model (bottom).

Chapter 6

Conclusion

TODO: replace me

Appendix A

Tables

Table A.1: Armadillos

Armadillos	are
our	friends

Appendix B

Figures

Figure B-1: Armadillo slaying lawyer.

Figure B-2: Armadillo eradicating national debt.

Bibliography

[1] Varun Jampani, Sebastian Nowozin, Matthew Loper, and Peter V. Gehler. The informed sampler: A discriminative approach to bayesian inference in generative computer vision models. *CoRR*, abs/1402.0859, 2014.

[2] Tejas D Kulkarni, Pushmeet Kohli, Joshua B Tenenbaum, and Vikash Mansinghka. Picture: A probabilistic programming language for scene perception. In *Proceedings of the ieee conference on computer vision and pattern recognition*, pages 4390–4399, 2015.

[3] Abhijit Kundu, Yin Li, and James M Rehg. 3d-rcnn: Instance-level 3d object reconstruction via render-and-compare. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3559–3568, 2018.

[4] Jonathan Tremblay, Thang To, Balakumar Sundaralingam, Yu Xiang, Dieter Fox, and Stan Birchfield. Deep object pose estimation for semantic robotic grasping of household objects. *CoRR*, abs/1809.10790, 2018.

[5] Yu Xiang, Tanner Schmidt, Venkatraman Narayanan, and Dieter Fox. Posecnn: A convolutional neural network for 6d object pose estimation in cluttered scenes. *arXiv preprint arXiv:1711.00199*, 2017.

[6] Ilker Yildirim, Tejas D Kulkarni, Winrich A Freiwald, and Joshua B Tenenbaum. Efficient analysis-by-synthesis in vision: A computational framework, behavioral tests, and comparison with neural representations. In *Proceedings of the thirty-seventh annual conference of the cognitive science society*, 2015.

[7] Ben Zinberg, Marco Cusumano-Towner, and Vikash K. Mansingkha. Structured differentiable models of 3d scenes via generative scene graphs. *Workshop on Perception as Generative Reasoning, NeurIPS 2019, Vancouver, Canada.*, 2019.

@articlegreen2009reversible, title=Reversible jump MCMC, author=Green, Peter J and Hastie, David I, journal=Genetics, volume=155, number=3, pages=1391–1403, year=2009, publisher=Citeseer

@inproceedingsCusumano-Towner:2019:GGP:3314221.3314642, author = Cusumano-Towner, Marco F. and Saad, Feras A. and Lew, Alexander K. and Mansinghka,

Vikash K., title = Gen: A General-purpose Probabilistic Programming System with Programmable Inference, booktitle = Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, series = PLDI 2019, year = 2019, isbn = 978-1-4503-6712-7, location = Phoenix, AZ, USA, pages = 221–236, numpages = 16, url = http://doi.acm.org/10.1145/3314221.3314642, doi = 10.1145/3314221.3314642, acmid = 3314642, publisher = ACM, address = New York, NY, USA, keywords = Markov chain Monte Carlo, Probabilistic programming, sequential Monte Carlo, variational inference,

@articlemarkley2007averaging, title=Averaging quaternions, author=Markley, F Landis and Cheng, Yang and Crassidis, John L and Oshman, Yaakov, journal=Journal of Guidance, Control, and Dynamics, volume=30, number=4, pages=1193–1197, year=2007

@INPROCEEDINGSEllson03graphvizand, author = John Ellson and Emden R. Gansner and Eleftherios Koutsofios and Stephen C. North and Gordon Woodhull, title = Graphviz and dynagraph – static and dynamic graph drawing tools, booktitle = GRAPH DRAWING SOFTWARE, year = 2003, pages = 127–148, publisher = Springer-Verlag