Supplementary material: Bayesian Object Models for Robotic Interaction with Differentiable Probabilistic Programming

Anonymous Author(s)
Affiliation
Address
email

In this brief document, we provide additional details about our approach.

1 Differentiable tree sampler

First, we prove that our differentiable tree sampler is guaranteed to produce a tree structured random variable. Recall that the DP-DAG strategy for sampling a DAG represents the graph structured random variable $\mathcal{G}$ by its factorized adjacency matrix $A = PUP^T$. Here, P is a permutation matrix and U is an upper-triangular matrix. For A to characterize a tree, we must explicity forbid any edge among siblings, or from a descendant to an ancestor in U.

For a proof sketch, recall that in our opportunistic rounding scheme, we disallow all edges (k, j) (for $k > i$) for all edges (i, j) in the canonical graph (i.e., with adjacency matrix U). By contradiction, let us assume that an edge (k, j) is added to the graph while another edge (i, j) exists. This would imply that a new path $i \mapsto k \mapsto j$ is created in the graph. This violates a basic property of a tree; only a unique path may exist across each pair of nodes (whereas we now have two such paths). This proves that our sampling strategy is guaranteed to return a tree.

Caveat: The sampling strategy we present is not an unbiased strategy (the opportunistic rounding step strongly favours edges between earlier entries in the canonical graph).

2 Implementation details

For model specification and gradient-based probabilistic inference, we leverage the pyro [1] (and numpyro [2]) probabilistic programming language (PPL). Pyro allows us to specify generative (probabilistic) models in python code using popular automatic differentiation API (e.g., pytorch [3], JAX [4]). Additionally, the use of a modern PPL enables us to use the wide variety of (Bayesian) inference engines (including ones that do not leverage autodifferentiation). In our initial experiments, we found gradient-free inference engines to work only for low-dimensional systems (3-5 parameters) and for likelihood functions involving only a small number of timesteps (less than 10). We, therefore, focus only on the gradient-based inference engines provided by pyro.

2.1 Inference engines

SVI: The stochastic variational inference (SVI) solver proceeds by maximizing the evidence lower bound (ELBO). The *stochasticity* in variational inference arises from the number of samples from the variational distribution used to compute the likelihood (and by extension the ELBO). In our experiments, we use 4 samples to compte the expected ELBO.

HMC and NUTS: For our gradient-based Monte Carlo solvers (Hamiltonian Monte Carlo and No-U-Turn Sampler), we leverage numpyro for efficient inference. Pyro with a pytorch backend has been extensively benchmarked against PyMC3 [5] and numpyro [2], and has been found to be significantly slower.

2.2 Prior distributions

Choice of priors for continuous variables: For the continuous-valued random variables in our probabilistic model, recall that we use a standard normal distribution as our prior. We also experimented with the standard uniform distribution $\text{Unif}(0, 1)$, but found marginal gains in inference time when using Gaussian priors.

Reparameterization trick: Several physical variables of interest in our model have varying constraints on parameter ranges. For instance, elasticity and stiffness parameters are on the scale of $[1000, 10000]$, while coefficients of static friction are in the $[0, 1]$ range. As such both variational and Monte Carlo inference engines are numerically unstable for such large variations in parameter ranges. We therefore reparameterize all parameters to a canonical range (-1 through 1).

Computing statistics over reparameterization ranges: To carry out the above reparameterization, we use the held out instances (e.g., the *train* split comprising PartNet [6] 3D assets.

Hierarchical priors: Our framework, optionally, allows for the flexibility of auto-tuning the observation noise variable if needed. This may be done by introducing a half-normal distribution that samples the standard-deviation of the observation noise parameter.

References

[1] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 2018.

[2] D. Phan, N. Pradhan, and M. Jankowiak. Composable effects for flexible and accelerated probabilistic programming in numpyro. *International Conference on Probabilistic Programming*, 2020.

[3] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[4] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

[5] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck. Probabilistic programming in python using pymc3. *PeerJ Computer Science*, 2:e55, 2016.

[6] K. Mo, S. Zhu, A. X. Chang, L. Yi, S. Tripathi, L. J. Guibas, and H. Su. PartNet: A large-scale benchmark for fine-grained and hierarchical part-level 3D object understanding. In *CVPR*, 2019.