Nested Compiled Inference for Hierarchical Reinforcement Learning

Tuan Anh Le **Atılım Güneş Baydin** **Frank Wood**
Department of Engineering Science
University of Oxford
{tuananh, gunes, fwood}@robots.ox.ac.uk

1 Introduction

Probabilistic programming languages (PPLs) allow the representation of probability distributions as computer programs by means of stochastic language primitives and conditioning of variable values on observations [8, 7]. Probabilistic programs are generative models as they generate samples from a joint distribution $p(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ are latent variables and $\mathbf{y}$ are output (or observed) data. Inference in PPLs amounts to computing the posterior $p(\mathbf{x}|\mathbf{y})$ conditioned on observed data. PPLs allow decoupling model specification from inference, which can be handled by techniques including Markov chain Monte Carlo [18], variational methods [19], expectation propagation [16], and sequential Monte Carlo [20]. Inference is often computationally expensive. Amortized inference [5] strategies are devised to store and reuse past inferences so that future inferences run faster.

2 Inference Compilation

We outline a framework [13] for using deep neural networks to amortize the cost of inference in universal PPLs—languages such as Church [6], Venture [14], and Anglican [20] that are Turing-complete and can represent any computable probability distribution [4]. We call this framework "inference compilation" because our method, given only a probabilistic program, automatically constructs a neural network architecture and trains it using a stream of training data from the model, producing a specialized *compilation artifact* that subsequently performs efficient approximate inference in the original model specified by the probabilistic program. This work can be seen from two perspectives where we (1) *incorporate deep learning techniques into probabilistic programming to amortize the cost of inference*, and, at the same time, (2) lay out a framework for using *universal PPLs for defining generative models in deep learning*, where the specified neural networks are trained using an "infinite" stream of synthetic data from the model, and we get supervision for free. Seen in this light, our efforts reside at the intersection of Bayesian approaches and deep learning.

Sequential importance sampling. During inference, our technique uses compilation artifacts to parameterize proposal distributions $q(\mathbf{x}|\mathbf{y}; \phi)$ in a sequential importance sampling scheme [1, 3], where ϕ represents the trainable artifact weights.

Objective function. We use the Kullback–Leibler divergence $D_{\text{KL}}\left(p(\mathbf{x}|\mathbf{y}) \,||\, q(\mathbf{x}|\mathbf{y}; \phi)\right)$ as a measure of closeness between $p(\mathbf{x}|\mathbf{y})$ and $q(\mathbf{x}|\mathbf{y}; \phi)$ and minimize the loss $\mathcal{L}(\phi) = -\frac{1}{M}\sum_{m=1}^{M} \log q(\mathbf{x}^{(m)}|\mathbf{y}^{(m)}; \phi)$ on execution traces $(\mathbf{x}^{(m)}, \mathbf{y}^{(m)}) \sim p(\mathbf{x}, \mathbf{y}),\ m = 1, \ldots, M$.

Training data. During compilation, minibatches of program traces $(\mathbf{x}^{(m)}, \mathbf{y}^{(m)})$ are generated on-the-fly from the probabilistic model and streamed to a stochastic gradient descent procedure using Adam [11] for optimizing neural network weights ϕ.

Workshop on Bayesian Deep Learning, NIPS 2016, Barcelona, Spain.

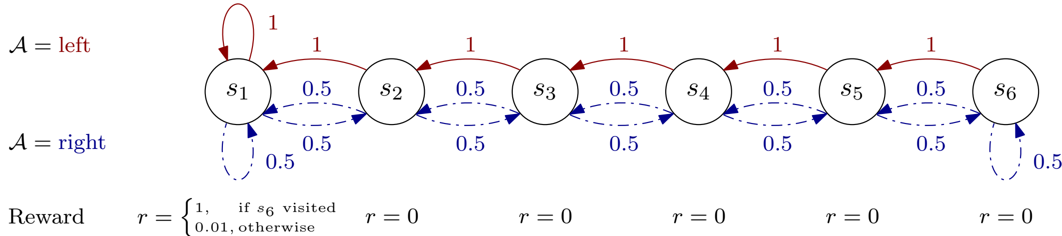

Figure 1: The stochastic decision process.

Neural network architecture. Compilation artifacts comprise a non-program-specific stacked LSTM [10] core and program-specific observation embedding, sample embeddings, and proposal layers specified by the probabilistic program. The artifacts are implemented in Torch [2] and our framework uses ZeroMQ [9] for interfacing with Anglican [20] during compilation and inference. This setup allows distributed inference with GPU support across many machines.

3 Example

In our recent work [13] we demonstrate using PPLs for generative modeling in inverse graphics (Captcha solving) and mixture models (identifying the number and parameters of components in an observed mixture).

Here, we show how we can view hierarchical reinforcement learning (RL) [12] as a nested inference scheme in probabilistic programming where we cast policy learning of intrinsic goals as inference on what we call an *inner query*, which is used inside an *outer query* that performs inference over sub-goal selection. We then "compile away" the inferential task of the *inner query*.

We formulate a stochastic decision process (Figure 1) that highlights the need for temporally extended exploration (similar to Kulkarni et al. [12] and Osband et al. [17]), with states $\mathcal{S} = \{s_1, s_2, s_3, s_4, s_5, s_6\}$ and actions $\mathcal{A} = \{\text{left}, \text{right}, \text{end}\}$. The agent starts at s_2. If the action left is chosen, it moves to the left deterministically. The right action moves it to the left or right with equal probability. The end action ends the execution of a current policy and ends the game if the agent is executing the last policy.[1] The game ends automatically if the agent reaches s_1 regardless of its action. The agent is rewarded with $r = 0$ for end states other than s_1. Otherwise $r = 1$ or $r = 0.01$ depending on whether s_6 was visited or not.

As explored by Kulkarni et al. [12], this example illustrates the difficulties faced by current RL algorithms on problems with sparse feedback. In the same spirit, we aim to decompose the problem into smaller problems, which can be solved by a specialized controller that is then used inside a meta-controller. In particular, we define an *inner query* named `go-to`, which during inference is responsible for outputting the policy with reaching the `goal-state` being the intrinsic motivation. This intrinsic motivation is enforced by placing an approximate Bayesian computation (ABC) [15] likelihood over the last state of the simulation being the `goal-state`. We also simulate over all possible `start-state` values from $\mathcal{S}$. The query is written in Anglican[2] as shown in the following listing.

```
(defquery go-to [game goal-state]
  (let [policy (sample policy-dist)]
    (loop [start-state 1]
      (if (< start-state 7)
       (let [history (execute-policy game start-state policy)]
          (observe (dirac (last history)) goal-state)
          (recur (inc start-state)))))
    policy))
```

[1]This policy change is a state change in a state space that is a Cartesian product of the agent and simulation state spaces. Hence the stochastic decision process in Figure 1 can be thought of as one of many subprocesses—each corresponding to the agent's current active policy—of an underlying stochastic decision process with the end action moving between them.

[2]More details and syntax can be found at `http://www.robots.ox.ac.uk/~fwood/anglican/`.

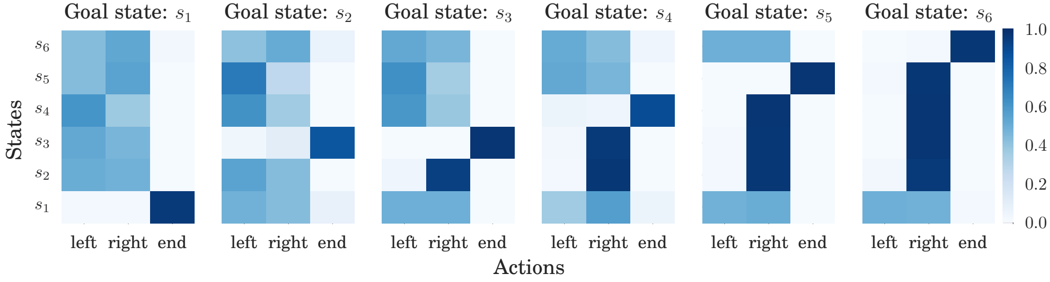

Figure 2: Policies from inference compilation of `go-to` with one particle; averaged over 100 runs.

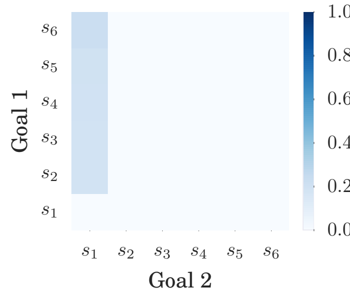

Figure 3: The empirical distribution over (`goal-1`, `goal-2`) from the outer query `play` run with sequential Monte Carlo with 100 particles and the inner query `go-to` run with inference compilation with one particle; averaged over 100 runs.

This is used inside an *outer query*[3] named `play`, which does inference over the intermediate high-reward goal `goal-1` and the end goal `goal-2` for the particular game dynamics specified by a given `game` variable —which, in this experiment, has high-reward state s_6 and terminal state s_1. This query returns two policies: one to reach the intermediate goal `goal-1`; and another to reach the end goal `goal-2`. The query is written in Anglican as shown in the listing below.

```
(defquery play [game]
  (let [goal-1 (sample (uniform-discrete 1 7))
        goal-2 (sample (uniform-discrete 1 7))
        policy-1 (sample ((conditional go-to) game goal-1))
        policy-2 (sample ((conditional go-to) game goal-2))
        history (execute-policy game 2 policy-1 policy-2)
    (observe (factor) (get-reward history))
    [goal-1 goal-2 policy-1 policy-2]))
```

Inference of the inner query is performed using our inference compilation framework. In Figure 2 we can see that the compilation artifact allows us to identify reasonable policies using only one particle of sequential importance sampling. In Figure 3, inference over the outer query using sequential Monte Carlo with 100 particles returns an empirical distribution over the subgoals (`goal-1`, `goal-2`) peaked at (s_6, s_1). Picking the maximum a posteriori policy pair (`policy-1`, `policy-2`), we obtain 0.17 average reward over 1000 runs.

The full theoretical underpinnings of the correspondence of our work to reinforcement learning are left for future work.

[3]The implementation here uses the Anglican special form `conditional` that takes in a query and returns a distribution object constructor, which, given the query parameter, returns a distribution object that represents the conditional distribution for that query and can be sampled from like any other primitive distribution (e.g. `normal`). For instance, `(conditional go-to)` is a distribution object constructor and `((conditional go-to) 5)` is a distribution object returning the posterior policy sampler for `go-to` with goal state s_6.

References

[1] M Sanjeev Arulampalam, Simon Maskell, Neil Gordon, and Tim Clapp. A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, 2002.

[2] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A MATLAB-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[3] Arnaud Doucet and Adam M Johansen. A tutorial on particle filtering and smoothing: Fifteen years later. *Handbook of Nonlinear Filtering*, 12(656–704):3, 2009.

[4] Cameron E Freer, Daniel M Roy, and Joshua B Tenenbaum. Towards common-sense reasoning via conditional simulation: Legacies of Turing in artificial intelligence. *Turing's Legacy: Developments from Turing's Ideas in Logic*, 42:195, 2014.

[5] Samuel J Gershman and Noah D Goodman. Amortized inference in probabilistic reasoning. In *Proceedings of the 36th Annual Conference of the Cognitive Science Society*, 2014.

[6] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

[7] Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2016-10-31.

[8] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Future of Software Engineering, FOSE 2014*, pages 167–181. ACM, 2014.

[9] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[11] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in Neural Information Processing Systems 29*, pages 3675–3683. 2016.

[13] Tuan Anh Le, Atılım Güneş Baydin, and Frank Wood. Inference compilation and universal probabilistic programming. *arXiv preprint arXiv:1610.09900*, 2016.

[14] Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

[15] Jean-Michel Marin, Pierre Pudlo, Christian P Robert, and Robin J Ryder. Approximate Bayesian computational methods. *Statistics and Computing*, 22(6):1167–1180, 2012.

[16] Thomas P Minka. Expectation propagation for approximate bayesian inference. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, pages 362–369. Morgan Kaufmann Publishers Inc., 2001.

[17] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped DQN. In *Advances in Neural Information Processing Systems 29*, pages 4026–4034. 2016.

[18] David Wingate, Andreas Stuhlmüller, and Noah D Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, pages 770–778, 2011.

[19] David Wingate and Theophane Weber. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299*, 2013.

[20] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.