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 \( p(x, y) \), where \( x \) are latent variables and \( y \) are output (or observed) data. Inference in PPLs amounts to computing the posterior \( p(x | y) \) 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 \( q(x|y; \phi) \) in a sequential importance sampling scheme [1, 3], where \( \phi \) represents the trainable artifact weights.

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

Training data. During compilation, minibatches of program traces \( (x^{(m)}, y^{(m)}) \) 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 \( \phi \).

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

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 $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$ and actions $A = \{\text{left}, \text{right}, \text{end}\}$. The agent starts at $s_2$. If the action $\text{left}$ is chosen, it moves to the left deterministically. The $\text{right}$ action moves it to the left or right with equal probability. The $\text{end}$ action ends the execution of a current policy and ends the game if the agent is executing the last policy. 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 $S$. The query is written in Anglican 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))
```

Figure 1: The stochastic decision process.
This is used inside an outer query\(^3\) named play, which does inference over the intermediate high-reward goal \(\text{goal-1}\) and the end goal \(\text{goal-2}\) 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 \(\text{goal-1}\); and another to reach the end goal \(\text{goal-2}\). The query is written in Anglican as shown in the listing below.

\[
\text{(defquery play [game]}
\begin{align*}
\text{(let [goal-1 (sample (uniform-discrete 1 7))} \hfill \\
\text{goal-2 (sample (uniform-discrete 1 7))} \hfill \\
\text{policy-1 (sample ((conditional go-to) game goal-1))} \hfill \\
\text{policy-2 (sample ((conditional go-to) game goal-2))} \hfill \\
\text{history (execute-policy game 2 policy-1 policy-2)} \hfill \\
\text{(observe (factor) (get-reward history))} \hfill \\
\text{[goal-1 goal-2 policy-1 policy-2])})}
\end{align*}
\]

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 \((\text{goal-1}, \text{goal-2})\) peaked at \((s_6, s_1)\). Picking the maximum a posteriori policy pair \((\text{policy-1}, \text{policy-2})\), 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 \text{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. \text{normal}). For instance, \((\text{conditional go-to})\) is a distribution object constructor and \(((\text{conditional go-to}) 5)\) is a distribution object returning the posterior policy sampler for go-to with goal state \(s_6\).
References


