Bean Machine: A Declarative Probabilistic Programming Language For Efficient Programmable Inference

Nazanin Tehrani NAZANINKT@FB.COM
Nimar S. Arora NIMARORA@FB.COM
Yucen Lily Li YUCENLI@FB.COM
Kinjal Divesh Shah KSHAH97@FB.COM
David Noursi DCALIFORNIA@FB.COM
Michael Tingley TINGLEY@FB.COM
Narjes Torabi NTORABI@FB.COM
Sepehr Masouleh SEPEHRAKHAVAN@FB.COM
Eric Lippert ERICLIPPERT@FB.COM
Erik Meijer ERIKM@FB.COM
Facebook Inc, USA

Abstract

A number of imperative Probabilistic Programming Languages (PPLs) have been recently proposed, but the imperative style choice makes it very hard to deduce the dependence structure between the latent variables, which can also change from iteration to iteration. We propose a new declarative style PPL, Bean Machine, and demonstrate that in this new language, the dynamic dependence structure is readily available. Although we are not the first to propose a declarative PPL or to observe the advantages of knowing the dependence structure, we take the idea further by showing other inference techniques that become feasible or easier in this style. We show that it is very easy for users to program inference by composition (combining different inference techniques for different parts of the model), customization (providing a custom hand-written inference method for specific variables), and blocking (specifying blocks of random variables that should be sampled together) in a declarative language. A number of empirical results are provided where we backup these claims modulo the runtime inefficiencies of unvectorized Python. As a fringe benefit, we note that it is very easy to translate statistical models written in mathematical notation into our language.

Keywords: Probabilistic Programming; Programmable Inference; Declarative Structure.

1. Introduction

Probabilistic Programming Languages (PPLs) provide a formal language for expressing statistical models and a general-purpose inference engine for performing statistical inference. In a typical scenario, a user specifies their problem in the PPL as a statistical model $P(\cdot)$, supplies observed values y for a subset of random variables Y, and queries for the posterior distributions of another set of variables X, $P(X \mid Y = y)$. In theory, PPLs should allow users to focus on modeling their problem rather than implementing and debugging inference algorithms. However, this vision has not come to fruition because general-purpose inference engines lag far behind model-specific handwritten inference algorithms.

We propose a new declarative universal PPL, Bean Machine, which offers three inference techniques to help close the performance gap between general-purpose and model-specific handwritten inference: compositional inference, block inference, and custom proposers. These *programmable* *inference* techniques can be executed efficiently in Bean Machine due to its declarative paradigm, which explicitly tracks the model dependency structure.

Specifically, Bean Machine models specify a distribution over possible *worlds*, where each world is a graph data structure that consists of the *directed graphical model* (Pearl and Russell, 2000) over a set of random variables and the assignment of values to these variables. The language is Universal and allows for the directed acyclic graph (DAG) of the random variables (or model dependence structure) as well as the number of variables to vary between worlds. Our language is an extension to BLOG (Milch et al., 2005) and allows much greater flexibility in specifying the conditional probability distribution of random variables using arbitary Python code.

In contrast, other imperative Universal PPLs, such as Church (Goodman et al., 2008), Pyro (Bingham et al., 2019), Turing (Ge et al., 2018), and Gen (Cusumano-Towner et al., 2019), represent the statistical model as a monolithic program that explicitly draws a sample for each random variable that is encountered in the execution of the program. Imperative models specify the distribution over possible execution *traces* of the program, where a trace includes the assignment of values to random variables as well as the linear control flow, but not the explicit DAG of variables. Now, both styles of PPLs can specify the same distribution over random variables, but the lack of an explicit DAG leads to limitations during inference.

In order to understand the benefits of the explicit DAG one must consider the state of the art implementation of trace-based inference, Lightweight MH (Wingate et al., 2011) (LMH). In this technique, a variable is chosen at random from the current trace and its value is updated through some proposal distribution (typically the parent conditional prior as described in Arora et al. (2010)). The rest of the trace downstream of the updated variable is then re-executed to either sample new random variables that are needed or to update the probability of variables whose distribution has changed. The new trace is then accepted or rejected using an MH update rule (Metropolis et al., 1953). Now, in a world-based MH inference, which we call Single-Site MH (SSMH), the overall steps are very similar. However, since the world includes the DAG, we have direct knowledge of the random variables whose distribution would change as a consequence of the proposed update. Hence we only need to update the distribution of the children of the updated random variable. Overall, the DAG helps to reduce the cost of each MH update, i.e. the cost of proposing a new value and deciding to accept or reject it, from O(size of trace) to O(size of the Markov blanket of the updated variable).

Gen circumvents this issue through a user-specified *argdiff* structure, allowing the inference engine to safely skip parts of execution; however, an incorrect *argdiff* may cause failures. C3 (Ritchie et al., 2016) proposes another solution that requires that models be written in a specific style that allows the DAG to be inferred. This could be considered similar to our approach except that we are advocating a declarative language rather than an optional modeling style. The classic language, BUGS (Spiegelhalter et al., 1996), also used a very similar approach of component-wise Gibbs sampling, but that language is constrained enough that the dynamic DAG is easy to compute.

We note that SSMH was first deployed in BLOG and the runtime advantage of SSMH versus LMH was previously noted in C3. Our contribution is to develop three techniques for programmable inference that are applicable to any language or modeling style that supports SSMH, and we demonstrate these techniques in the context of a new declarative PPL, Bean Machine.

Compositional inference allows different MH proposer algorithms to be used for different variables in the model, providing two major advantages. The first is the ability to select, compose, and tune the most efficient set of proposers for the model. The second is that it allows state-of-the-art gradient-based proposers such as Hamiltonian Monte Carlo (HMC) (Duane et al., 1987; Neal, 1993), No U-Turn Sampler (Hoffman and Gelman, 2014), and Newtonian Monte Carlo (NMC) (Arora et al., 2020), to be used on models with mixed discrete and continuous variables, which otherwise are only applicable to continuous random variables. The imperative Turing language (Ge et al., 2018) also implements compositional inference; however, it does not have the dynamic graph structure needed for efficient MH updates.

Block inference allows a set of variables to be specified to undergo MH updates together. Proposing new values for all variables in the model followed by a single accept/reject decision often leads to low acceptance rates, as a bad proposal for any individual variable may result in a low overall acceptance probability. Single-site inference can improve this behavior by separately proposing values for each variable and accepting or rejecting these proposals individually. However, in cases where a model contains multiple highly-correlated variables, the optimal proposal strategy is to group the correlated variables and MH update them together.

Custom proposers allow users to incorporate domain knowledge to achieve satisfactory performance in models where even state-of-the-art gradient-based proposers are inadequate. Together with its compositional inference engine, Bean Machine allows for custom proposers to be implemented on a per-variable basis. Customized inference was first deployed in BLOG for the citation matching problem in Milch and Russell (2006). However, this required wholesale replacement of the inference algorithm with a custom one, which defeats much of the point of generic inference.

We demonstrate that the combination of these techniques allows the user to obtain efficient inference performance with a minimal amount of handwritten code.

Finally, the syntax of our PPL, embedded in the Python language, is targeted toward statisticians who are familiar with conventional mathematical notation for statistical models. We show examples where statistical models translate very easily—often line by line—into Bean Machine syntax. As an aside, the name Bean Machine pays homage to a device consisting of pegs arranged in a quincunx pattern. Beans dropped from the top deflected randomly and collected at the bottom in a manner consistent with the law of large numbers (Galton, 1894). In effect, that device was the first PPL.

2. Embedding Bean Machine in Python

In Bean Machine, a random variable is declared by adding a `@random_variable` decorator to a Python function. A decorated function with signature $f(a_1, \cdots, a_r)$ defines a variable for all possible values of the tuple $(a_1, \cdots, a_r)$. The function body defines the variable's dependency function, a code block that specifies how to compute its distribution. This function can be arbitrary Python code, may reference other random variables, and must return a probability distribution object. A distribution object is any Python object implementing a method to draw a sample, a method to evaluate a sample's log probability, and an attribute describing the distribution's domain[1]. The arguments $a_1, \cdots, a_r$, represent indices or subscripts for identifying the random variable.

Figure 1 uses the Hidden Markov Model (HMM) to compare three different syntaxes. In Bean Machine's syntax in Figure 1b, the function `mu(k)` defines a variable for each value of k—analogous to the variable μ_k in the mathematical notation. The call to `Normal` returns a distribution object representing a normal distribution with mean `alpha` and variance `beta`. The variable `y(i)` depends upon the variables `x(i)`, `mu(x(i))`, and `sigma(x(i))`.

Figure 1c shows the imperative version of the same model. The variables `mu(k)`, `sigma(k)`, and `theta(k)` are sampled first. Based upon their values, the variables `x(i)` and `y(i)` are then

1. Bean Machine currently support, reals, real vectors, half-spaces, sim- plexes, integers, and bounded integer domains.

$$\begin{aligned}
\mu_k &\sim \text{Normal}(\alpha, \beta)\\
\sigma_k &\sim \text{Gamma}(\nu, \rho)\\
\theta_k &\sim \text{Dirichlet}(\kappa)\\
x_i &\sim \begin{cases}\text{Categorical}(init) & \text{if } i = 0\\ \text{Categorical}(\theta_{x_{i-1}}) & \text{if } i > 0\end{cases}\\
y_i &\sim \text{Normal}(\mu_{x_i}, \sigma_{x_i})
\end{aligned}$$

(a) Mathematical notation

```
@random_variable
def mu(k):
  return Normal(alpha, beta)
@random_variable
def sigma(k):
  return Gamma(nu, rho)
@random_variable
def theta(k):
  return Dirichlet(kappa)
```

```
@random_variable
def x(i):
  if i == 0:
    return Categorical(init)
  elif i>0:
    return Categorical(theta(x(i−1)))
@random_variable
def y(i):
  return Normal(mu(x(i)), sigma(x(i)))
```

(b) Bean Machine

```
mu, sigma, theta, x, y = {}, {}, {}, {}, {}

with Model():
  for k in range(K):
    mu[k] = Normal(f‘‘mu[{k}]”, alpha, beta)
    sigma[k] = Gamma(f‘‘sigma[{k}]”, nu, rho)
    theta[k] = Dirichlet(f‘‘theta[{k}]”, kappa)

  x[0] = Categorical(‘‘x[0]”, init)
  for i in range(1, N):
    x[i] = Categorical(f‘‘x[{i}]”, theta[x[i−1]])
    y[i] = Normal(f‘‘y[{i}]”, mu[x[i]], sigma[x[i]], observed=data[i])
```

(c) An imperative version.

Figure 1: HMM model in three languages.

sampled. Unlike in Figure 1b, the calls to `Normal`, `Gamma`, `Dirichlet`, and `Categorical` directly return a numerical sample from their respective distributions. The strings passed to the distribution functions are used by the inference engine to identify random variables across different program executions. Note that the imperative model definition includes many inference-specific details, such as the order of the sampling of variables and the length of the sequence N, Imperative models also commonly include the observations, as shown by in the “observed” attribute for `y(i)`. However, Bean Machine keeps the model specification completely separate from any details of inference. To perform inference in Bean Machine, the user provides a list of the random variables to query with the data for the observed random variables. Bean Machine then instantiates the minimal number of variables necessary to perform inference for the desired query.

Gen also allows for symbolic naming of random variables, and allows for observations to be bound to the observed variables after specifying the model. However, in Gen, as in other imperative languages, it is the model writer's responsibility to explicitly sample all the random variables that are observed or queried during inference.

The semantics of Bean Machine are very similar to those established in the BLOG paper. We adapt the Contingent Bayes Net notation in Arora et al. (2010) for a formal description. A Bean Machine model consists of a set of random variables $\mathcal{V}$ and a dependency function $\mathcal{T}_X$ for each variable $X \in \mathcal{V}$ such that $\mathcal{T}_X$ is a pure function that can only refer to variables in $\mathcal{V}$ or other pure functions in the host language Python and that the return value of $\mathcal{T}_X$ must be a probability distribution. A world σ is an assignment of values to a finite subset of variables, $vars(\sigma)$, such that σ_X is the value of variable X in σ. We will refer to $\sigma_{\mathcal{T}_X}$, as the assignment of values to the subset of variables in σ that are referenced in the execution of $\mathcal{T}_X$ and $p_X(\cdot|\sigma_{\mathcal{T}_X})$ as the resulting distribution. A world σ is self-supporting if it contains $vars(\sigma_{\mathcal{T}_X})$ for all $X \in \sigma$, i.e. all variables referenced in the execution of the dependency function of X are already assigned a value in σ. A world is well-defined if the parent edges from each variable X to $vars(\sigma_{\mathcal{T}_X})$ induced by the world form an acyclic graph. A model is well-defined if all self-supporting worlds, σ, that are possible over the model are well-defined. We further require that the domain of $p_X(\cdot|\sigma_{\mathcal{T}_X})$ for all X in $\mathcal{V}$ be the same in all worlds σ. The a-priori probability of any self-supporting world σ is given by,

$$p(\sigma) \quad = \prod_{X \in vars(\sigma)} p_X(\sigma_X|\sigma_{\mathcal{T}_X}).$$

The task of inference is simply to find the posterior distribution over all minimal self-supporting worlds that are consistent with the observations and that include the queries. Unlike BLOG, we don't have an explicit number statement or an origin statement or explicit object types. Number statements are simply syntactic sugar since we can always introduce a random variable to represent the number of objects of some type. For example, in the HMM model we could change `K` from a constant to a function `K()` defined as `K = random_variable(lambda: Poisson(5))`, where we use the `random_variable` decorator in the equivalent functional form.

An important advantage of Bean Machine's declarative syntax is that the dependency function of each variable is clearly demarcated. In the imperative version, without performing a data flow analysis on the code, it is not obvious what the parents of the `x(i)` or `y(i)` variables are. In Bean Machine, the parents of a random variable can be identified by simply executing its dependency function and recording all referenced variables. For example, the parents of `x(3)` can be computed by executing its dependency function with the argument $i = 3$. This will cause execution to go to the `elseif` branch and reference `x(2)`. If we suppose that `x(2)` is sampled to be 7 during that inference iteration, then the next variable to be referenced is `theta(7)`. The `@random_variable` decorator allows the engine to intercept these references to maintain the dependency graph. Note that the dependency graph is dynamic and may change from one inference iteration to another.

3. Inference

The goal of Bayesian inference is to compute the posterior distribution conditioned upon observed values in a given model. Bean Machine's inference engine is built upon MCMC using the MH rule and outputs the posterior as a set of samples. We follow the methodology first proposed in BLOG, where inference is performed over minimal self-supporting worlds. The world is represented using a graph structure containing the most recently sampled values for all variables, and the DAG model dependency structure. The fundamental algorithm underlying our inference engine is Single-Site Metropolis Hastings (SSMH), described in Algorithm 1, which we then extend with compositional inference, block inference, and custom proposers.

In SSMH, an MH update is performed on a single random variable at a time. This algorithm is an extension of Pearl (1987) to open universe models. Given a minimal self-supporting world σ, on each inference iteration, we iterate through the unobserved random variables in shuffled order, and perform an MH update on each individual variable X in turn. A single MH update consists of proposing a new value for the variable X, and then making an accept/reject decision using the standard MH rule. The parent conditional prior distribution, $p_X(\cdot \mid \sigma_{\mathcal{T}_X})$, is typically the default choice for the proposal distribution.

For gradient-based single-site methods such as Hamiltonian Monte Carlo (HMC) or Newtonian Monte Carlo (NMC), because of Bean Machine's single-site nature, when resampling X, we only need to consider the term $\pi_X(x)$ given by,

$$\pi_X(x) = p_X(x \mid \sigma_{\mathcal{T}_X}) \prod_{Y \in \{Y \mid X \in \sigma_{\mathcal{T}_Y}\}} p_Y(\sigma_Y | \sigma_{\mathcal{T}_Y}).$$

We note that $\frac{\partial \log \pi_X(x)}{\partial x} = \frac{\partial \log p(\sigma)}{\partial x}$, and all of the quantities in the definition of $\pi_X(x)$ can be computed by executing the dependency function of X and the dependency function of the children of X. Thus the runtime complexity of computing the proposal distribution for X, $\mathcal{Q}_X(\cdot|\sigma)$ or of performing an MH update on one variable is proportional to the size of its Markov blanket (i.e. its parents, children, and parents of its children) which is typically smaller than the full graph. Because single-site only requires local updates, second-order gradient-based inference methods such as NMC are tractable in Bean Machine.

The model dependency structure is dynamic and may change when new values are proposed for the variables. Thus, an MH proposal consists of the new value for the variable along with the updated dependency structure. We commit both changes to the world if a proposal is accepted.

Algorithm 1 Single-Site Metropolis Hastings

```
Input: evidence $\mathcal{E}$ and queries $\mathcal{R}$
Given: a family of proposal distributions $\mathcal{Q}$.
Create: initial world $\sigma$ initialized with $\mathcal{E}$ and extended to include $\mathcal{R}$
repeat
    assign $V = vars(\sigma) - \mathcal{E}$
    for $X$ in $V$ do
        Sample $x' \sim \mathcal{Q}_X(\cdot|\sigma)$
        Clone $\sigma$ to $\sigma'$ and set $\sigma'_X = x'$
        Recompute $\sigma'_{\mathcal{T}_Y}$ for $Y \in$ children of $X$ in $\sigma'$
        Make $\sigma'$ minimal and self-supporting.
        $\alpha = min\left[1, \frac{p(\sigma')\mathcal{Q}_X(\sigma_X|\sigma')}{p(\sigma)\mathcal{Q}_X(\sigma'_X|\sigma)}\right]$
        $u \sim$ Uniform(0, 1)
        if $u < \alpha$ then
            $\sigma = \sigma'$
        end if
    end for
    Output sample $\sigma_{\mathcal{R}}$
until Desired number of samples.
```

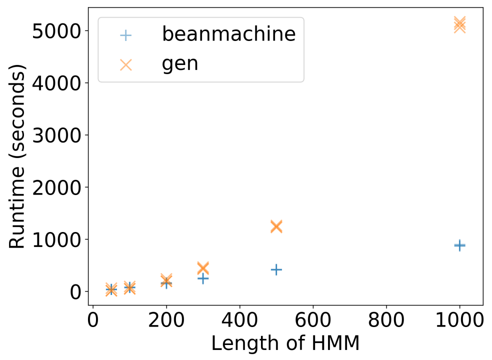

Figure 2: Runtime of HMM model with observed θ, μ and σ with K=3 and 1000 single-site MCMC samples.

HMM Experiment for SSMH The runtime of SSMH is proportional to the number of variables in the world times the size of the Markov blanket for each variable. Figure 2 shows Bean Machine's inference runtime on the HMM implementation in Figure 1 with respect to the length N of the HMM. As the size of the Markov blanket for each variable is constant, Bean Machine achieves a constant time MH update per variable, resulting in an overall linear runtime.

For reference, we compare Bean Machine to Gen's single-site inference performance. Without an explicit DAG, Gen re-executes its entire imperative model, resulting in a linear single variable MH update. Therefore, as N increases, the runtime increases quadratically.

4. Programmable Inference

Bean Machine supports three programmable techniques: compositional, block, and custom.

Composition Different variables require different proposal algorithms to explore the space efficiently. For continuous random variables, gradient-based proposers result in improved exploration over vanilla MH. For bounded discrete random variables, other techniques such as *Uniform sampling*, where new samples are drawn with equal probability across all values, may lead to better performance. Because Bean Machine's single-site inference engine already MH updates each variable individually, it is straightforward to extend the framework to use a different proposal algorithm per variable. This is similar to Turing's approach to compositional inference, except Bean Machine benefits from a fast single-site implementation due to its declarative syntax.

Blocking Single-site inference is not always suitable for models with highly correlated variables as no single variable can be accepted individually. Referring back to the HMM example in Figure 1, if the proposed value for the hidden state `x(i)` changes from `k` to `k'`, then `y(i)` is no longer a child of `mu(k)` or `sigma(k)`, but is now a child of `mu(k')` and `sigma(k')`. The likelihood of the world with only the change in `x(i)` will be very low, as `mu(k)`, `sigma(k)`, `mu(k')` and `sigma(k')` were all sampled with the assumption that `y(i)` was observed from hidden state `k`.

Bean Machine's solution is to propose new values for `mu`'s and `sigma`'s based on the change for `x(i)`. First, a new value for `x(i)` is proposed and the world is updated to reflect `y(i)` as an observation of hidden state k'. Then, new values are sequentially proposed for the affected mu's and sigma's. Note that the only `mu`'s and `sigma`'s impacted by the proposal from k to k' are within the Markov blanket of the original value for `x(i)` or the Markov blanket of the proposed value for `x(i)`, which is easily available through Bean Machine's DAG model dependency structure. Finally, the new proposals for `x(i)`, `mu(k)`, `mu(k')`, `sigma(k)`, and `sigma(k')` are treated as a single block and accepted/rejected together using Metropolis Hastings.

Block inference allows Bean Machine to overcome the limitations of single-site because highly correlated variables are updated together, allowing for worlds with higher probabilities. Furthermore, the inference has efficient runtimes because it is limited to resampling variables within the Markov blanket. In Figure 3, we show the succinct syntax of specifying block inference with the line `mh.add_sequential_propose([x, mu, sigma])`. This line of code specifies that in addition to the single-site proposers for all variables, an additional block proposer, which first samples `x`, and then updates all `mu`'s and `sigma`'s in the Markov blankets, will be introduced.

METHOD	K	n_{EFF}
BLOCK	25	109
NON-BLOCK	25	89
BLOCK	50	93
NON-BLOCK	50	30

Table 1: HMM results (median of 3 trials with N=200 and 100 samples)

```
mh = SingleSiteCompositionalInference()
mh.add_sequential_proposer([x, mu, sigma])
queries = [x(N − 1)]
          + [theta(k) for k in range(K)]
          + [mu(k) for k in range(K)]
          + [sigma(k) for k in range(K)]
obs = {y(i): data[i] for i in range(N)}
samples = mh.infer(queries, obs)
```

Figure 3: Code for invoking block inference and specifying queries and observations for HMM

In Table 1, the effective sample size (n_{eff}) for the HMM model is shown for inference with and without blocked moves. From the table, it is clear that block inference outperforms inference without blocking. Additionally, as the number of hidden states K increases, the performance of non-blocked inference progressively decreases; this is because each hidden state is explaining fewer observations, so a change in `x(i)` without a corresponding change in `mu` leads to a less plausible world. In Lightweight MH, which does not resample any `mu`'s after `x(i)`, the inference was completely unable to generate any new samples when the number of observations N exceeded 30, even with only three hidden states K.

Custom Proposers The final programmable inference method provided by Bean Machine allows users to supply handwritten proposal algorithms for specific random variables. This enables users to easily incorporate domain knowledge in a targeted and limited manner to inference.

To show the power of custom proposers, we implemented a PPL model for locating seismic events (Arora et al. (2013); Arora and Russell (2017)). We further simplified it as shown in Figure 4a to have exactly one event with attributes given by the random variable `event_attr()`. The variable `is_detected(station)` represents whether a given station detects this event. At the subset of detecting stations, the attributes of the detection are given by `det_attr(station)`. The inference problem is to find the event attributes given the detection attributes.

This problem is hard even for state-of-the-art gradient-based proposers due to the non-convexity of its posterior distribution. Luckily, there is domain knowledge within seismology to mathematically *invert* the most likely attributes of an event given the detection attributes at a single station. Due to the noise in the data, this predicted location can be inaccurate. But, with enough stations, it is likely that one of the predictions will be close to the true values. With this in mind, we used Bean Machine's easily implementable proposer interface (Figure 4b) to write a custom proposer for the `event_attr()` variable, which inspects the `det_attr(s)` children variables and uses a Gaussian mixture model proposer around the inverted locations for each of the detections.

```
@random_variable
def event_attr():
    return SeismicEventPrior()

@random_variable
def is_detected(station):
    prob = calculate_prob(station, event_attr())
    return Bernoulli(prob)

@random_variable
def det_attr(station):
    det_loc = calculate_det(station, event_attr())
    return Laplace(det_loc, scale)
```

(a) Seismic 2D Model

```
class SeismicProposer(Proposer):
    def propose(self, variable, curr_world):
        # return a new value for variable and log probability of
        # proposing the new value in the current world
        det_attrs = [child.value for child in variable.children
            if child.dependency_fn = det_attr]
        event_attrs = [seismic_invert(det) for det in det_attrs]
        self.gmm = construct_GMM_dist(event_attrs)
        new_value = self.gmm.sample()
        return new_value, self.gmm.log_prob(new_value)
    def post_process(self, variable, new_world):
        # return the log probability of proposing the
        # original value of the variable in the new world
        return self.gmm.log_prob(variable.value)
```

(b) Custom proposer

Figure 4: Seismic2D model and corresponding custom proposer

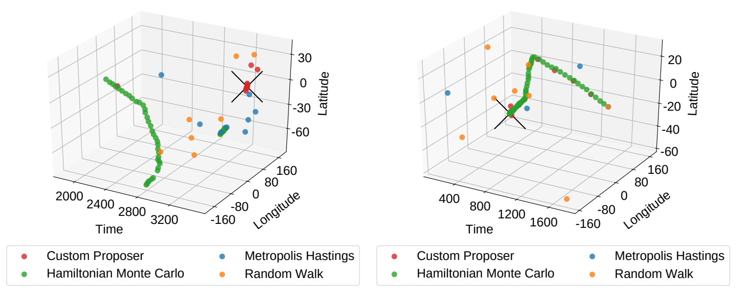

Figure 5: Samples for two events with known detections

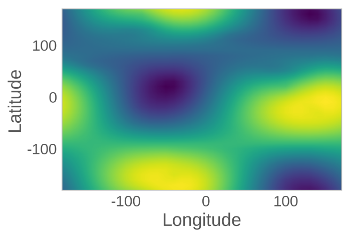

Figure 6: True multimodal posterior for the first event in Figure 5

Our experiments in Figure 5 show that the custom proposer converges to the true location marked with an X much more quickly than other methods, including gradient-based methods such as HMC. For the first event, Figure 6 reveals its multimodality in longitude and latitude when holding the time attribute constant at its true value. The custom proposer is able to find the true location of the event due to the domain knowledge, while HMC, reliant on gradient information, can take many iterations before moving from one mode to the other.

5. Experiments on Standard Models

5.1 Bayesian Logistic Regression

Bean Machine is implemented using the PyTorch (Paszke et al., 2017) tensor library to speed up dense numerical operations and avoid Python's interpreter overhead. These fast tensor operations are best exploited by inference on vectorized models with a few high dimensional random variables. Figure 7 demonstrates this for the Bayesian Logistic Regression (BLR) model.

$$\alpha \sim \mathcal{N}(0, 10) \quad \alpha \in \mathbb{R}$$
$$\beta \sim \mathcal{N}(0, 2.5) \quad \beta \in \mathbb{R}^K$$
$$\sigma \sim \text{LogNormal}(0, \rho) \quad \sigma \in \mathbb{R}^K$$
$$x_i \sim \mathcal{N}(0, \sigma) \quad x_i \in \mathbb{R}^K$$
$$\mu_i = \alpha + x_i^T \beta$$
$$y_i \sim \text{Bernoulli}(\text{logit} = \mu_i)$$

(a) Mathematical version

```
@random_variable
def alpha():
    return Normal(0, 10)
@random_variable
def beta():
    return Normal(0, 2.5, shape=(K,))
@random_variable
def sigma():
    return LogNormal(0, RHO, shape=(K,))
```

```
@random_variable
def x(i):
    return Normal(0, sigma())
def mu(i):
    return alpha() + dot(x(i), beta())
@random_variable
def y(i):
    return Bernoulli(logit = mu(i))
```

(b) Bean Machine version

Figure 7: Bayesian Logistic Regression (BLR) Model.

For evaluation purposes, we sample a dataset of N pairs of x and y values from the model and use half for inference and the other half for evaluating predictive log-likelihood (PLL). Figure 8 plots the growth of PLL versus samples across various inference engines for a small problem size, $N = 20\text{K}$. Table 2 shows the corresponding inference runtime and n_{eff} stats. Bean Machine, Stan (Carpenter et al., 2017) and NumPyro (Phan et al., 2019) are all using NUTS. Pyro is using Stochastic Variational Inference, and Bootstrap is using the basic logistic regression module in SciPy. This experiment demonstrates that Bean Machine and Stan both converge similarly, however, because it is a small model, the Bean Machine runtime is dominated by Python interpreter overhead. Note that Pyro does not converge to the same PLL within 1000 samples. Convergence difficulty in the BLR model is due to the high value of hyper-prior ρ, which introduces differential scaling along each data dimension.

Figure 9 and Table 3 show the same BLR experiment for a larger problem size, $N =$2M. Comparisons are limited to Stan and Bean Machine. The results show that Bean Machine continues to scale well on this model and performs better than Stan in runtime due to its use of PyTorch tensor library.

5.2 Annotation Model

Our final set of experiments is on a classic annotation model previously presented in Passonneau and Carpenter (2014). The model is specified in Figure 10 and is used to estimate the prevalence, π, of

C possible categories of items given noisy labels provided by labelers with an unknown confusion matrix unique to each labeler. θ_l is the confusion matrix of the l'th labeler such that θ_{lmj} is the probability that this labeler will label an item with true category m as j. Similar to the BLR model, we generate N triples of labeler, item, and label with half given to inference to deduce the prevalence π and confusion matrices $\theta_l\cdot$.

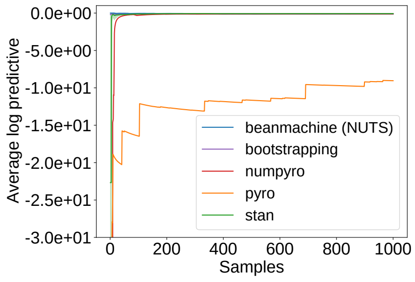

Figure 8: BLR predictive log likelihood with N=20000, K=10, ρ=10.

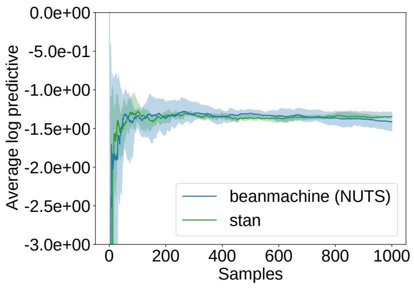

Figure 9: BLR predictive log likelihood with N=2M, K=10, ρ=10.

METHOD	RUNTIME(S)	n_{EFF}
BOOTSTRAPPING	70	2968
BEAN MACHINE	196	446
NUMPYRO	15.73	381
PYRO(SVI)	4055.7	4
STAN	27.33	1037

Table 2: BLR: N=20000, K=10, ρ=10.

METHOD	RUNTIME(S)	n_{EFF}
BEAN MACHINE	4293.4	164
STAN	13903	1486

Table 3: BLR: N=2M, K=10, ρ=10.

$$
\begin{aligned}
\pi &\sim \text{Dirichlet}\left(\frac{1}{C}, \ldots, \frac{1}{C}\right) \\
z_i &\sim \text{Categorical}(\pi) \\
\theta_{lm} &\sim \text{Dirichlet}(\alpha_m) \\
y_{li} &\sim \text{Categorical}(\theta_{lz_i})
\end{aligned}
$$

(a) Mathematical version

```
@random_variable
def pi():
    return Dirichlet(ones(C) / C)
@random_variable
def z(item):
    return Categorical(pi())
```

```
@random_variable
def theta(labeler, true_class):
    return Dirichlet(ALPHA[true_class])
@random_variable
def label(labeler, item):
    return Categorical(theta(labeler, z(item)))
```

(b) Bean Machine version

Figure 10: Annotation Model.

Bean Machine's declarative syntax shines on this example: simultaneously concisely capturing the model's complex and dynamic dependency structure, while also making these dependencies explicitly available to the inference engine. Single-site inference exploits this structure and allows the use of appropriate inference techniques on each of these variables, including second-order methods rarely viable in coarse models with high dimensional variables. In this model, Bean Machine uses NMC for the continuous-valued variables `pi()` and `theta(..)` and uniform sampling for each of the discrete `z(..)` variables. In contrast, Stan, which doesn't have single-site inference, requires the users to manually integrate out `z` in the model and it concatenates all of the `pi` and `theta` variables into one vector for *global* inference.

The results comparing the two PPLs are shown in Figure 11 and Table 4. This experiment shows Bean Machine is able to obtain results comparable to Stan. The coordinate-wise moves in single-site inference allows for an effective exploration of the posterior. We acknowledge that un-vectorized Python is slow, but the methods that we have outlined in this paper are applicable to declarative PPLs written in Julia or C++ where such performance limitations do not apply.

METHOD	RUNTIME(S)	n_{eff}
SINGLE-SITE	8560	622
GLOBAL	104	487

Table 4: Annotation Model with N=1000, 50 labelers, and C=3. Global results are from Stan and Single-Site are from Bean Machine

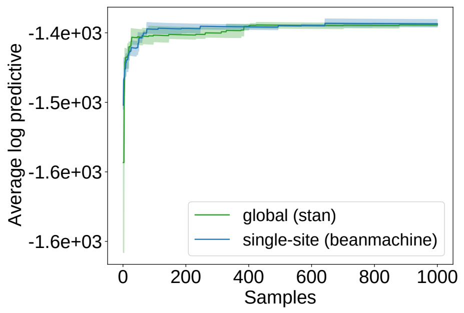

Figure 11: Annotation Model PLL with N=1000, 50 labelers, and C=3.

6. Conclusion

Our work has highlighted the importance of language style – declarative versus imperative – in PPL research. We have shown that the the problem of deducing the dynamic dependence graph that current imperative PPL research is grappling with was previously solved by declarative PPLs. In addition to inventing a new universal declarative PPL, we have shown that some of the recent work on composable and programmable inference can be incorporated and extended very easily in our language. In future, we are working on using information from the dynamic dependence graph to efficiently vectorize the basic inference operations.

References

N. S. Arora and S. Russell. Seismic 2-D. `https://github.com/nimar/seismic-2d/blob/master/description.pdf`, 2017. Accessed: 2020-1-10.

N. S. Arora, R. d. S. Braz, E. B. Sudderth, and S. Russell. Gibbs sampling in open-universe stochastic languages. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, pages 30–39. AUAI Press, 2010.

N. S. Arora, S. Russell, and E. Sudderth. NET-VISA: Network processing vertically integrated seismic analysis. *Bulletin of the Seismological Society of America*, 103(2A):709–729, 2013.

N. S. Arora, N. K. Tehrani, K. D. Shah, M. Tingley, Y. L. Li, N. Torabi, D. Noursi, S. A. Masouleh, E. Lippert, and E. Meijer. Newtonian Monte Carlo: single-site MCMC meets second-order gradient methods. *arXiv preprint arXiv:2001.05567*, 2020.

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. *The Journal of Machine Learning Research*, 20(1):973–978, 2019.

B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

M. F. Cusumano-Towner, F. A. Saad, A. K. Lew, and V. K. Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 221–236, 2019.

S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth. Hybrid Monte Carlo. *Physics letters B*, 195(2): 216–222, 1987.

F. Galton. *Natural inheritance*. Macmillan and Company, 1894.

H. Ge, K. Xu, and Z. Ghahramani. Turing: A language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics*, pages 1682–1690, 2018.

N. Goodman, V. Mansinghka, D. M. Roy, K. Bonawitz, and J. Tenenbaum. Church: a language for generative models with non-parametric memoization and approximate inference. In *Uncertainty in Artificial Intelligence*, 2008.

M. D. Hoffman and A. Gelman. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15(1):1593–1623, 2014.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

B. Milch, B. Marthi, S. Russell, D. Sontag, D. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *IJCAI International Joint Conference on Artificial Intelligence*, pages 1352–1359, 2005.

B. C. Milch and S. J. Russell. *Probabilistic models with unknown objects*. PhD thesis, University of California, Berkeley, 2006.

R. M. Neal. Bayesian learning via stochastic dynamics. In *Advances in Neural Information Processing Systems*, pages 475–482, 1993.

R. J. Passonneau and B. Carpenter. The benefits of a model of annotation. *Transactions of the Association for Computational Linguistics*, 2:311–326, 2014.

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.

J. Pearl. Evidential reasoning using stochastic simulation of causal models. *Artificial Intelligence*, 32(2): 245–257, 1987.

J. Pearl and S. Russell. Bayesians networks. *Handbook of Brain Theory and Neural Networks*, pages 157–160, 2000.

D. Phan, N. Pradhan, and M. Jankowiak. Composable effects for flexible and accelerated probabilistic programming in numpyro. *arXiv preprint arXiv:1912.11554*, 2019.

D. Ritchie, A. Stuhlmüller, and N. Goodman. C3: Lightweight incrementalized MCMC for probabilistic programs using continuations and callsite caching. In *Artificial Intelligence and Statistics*, pages 28–37, 2016.

D. Spiegelhalter, A. Thomas, N. Best, and W. Gilks. BUGS 0.5: Bayesian inference using Gibbs sampling manual (version ii). *MRC Biostatistics Unit, Institute of Public Health, Cambridge, UK*, pages 1–59, 1996.

D. Wingate, A. Stuhlmüller, and N. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 770–778, 2011.