Turing: a language for flexible probabilistic inference

Hong Ge
University of Cambridge

Kai Xu
University of Edinburgh

Zoubin Ghahramani
University of Cambridge
Uber

Abstract

Probabilistic programming is becoming an attractive approach to probabilistic machine learning. Through relieving researchers from the tedious burden of hand-deriving inference algorithms, not only does it enable development of more accurate and interpretable models but it also encourages reproducible research. However, successful probabilistic programming systems require flexible, generic and efficient inference engines. In this work, we present a system called `Turing` for flexible composable probabilistic programming inference. `Turing` has a intuitive modeling syntax and supports a wide range of sampling based inference algorithms. Most importantly, `Turing` inference is composable: it combines Markov chain sampling operations on subsets of model variables, e.g. using a combination of a Hamiltonian Monte Carlo (HMC) engine and a particle Gibbs (PG) engine. This composable inference engine allows the user to easily switch between black-box style inference methods such as HMC, and customized inference methods. Our aim is to present `Turing` and its composable inference engines to the community and encourage other researchers to build on this system to help advance the field of probabilistic machine learning.

1 Introduction

Probabilistic model-based machine learning [Ghahramani, 2015, Bishop, 2013] has been used successfully for a wide range of problems and new applications are constantly being explored. For each new application, however, it is currently necessary first to derive the inference method, e.g. in the form of variational or Markov chain Monte Carlo (MCMC) algorithm, and then implement it in application-specific code. Worse yet, building models from data is often an iterative process, where a model is proposed, fit to data and modified depending on its performance. Each of these steps is time-consuming, error-prone and usually requires expert knowledge in mathematics and computer science, an impedance for researchers who are not experts. In contrast, deep learning methods have benefited enormously from easy-to-use frameworks based on automatic differentiation that implement end-to-end optimisation. There is a real potential for automated probabilistic inference methods (in conjunction with existing automated optimisation systems) to revolutionise machine learning practice.

Proceedings of the 21st International Conference on Artificial Intelligence and Statistics (AISTATS) 2018, Lanzarote, Spain. PMLR: Volume 84. Copyright 2018 by the author(s).

Probabilistic programming languages [Goodman et al., 2008, Wood et al., 2014, Lunn et al., 2000, Minka et al., 2014, Stan Development Team, 2014, Murray, 2013, Pfeffer, 2001, 2009, Paige and Wood, 2014, Murray et al., 2017] aim to fill this gap by providing a very flexible framework for defining probabilistic models and automating the model learning process using generic inference engines. This frees researchers from writing complex models by hand and enables them to focus on designing a suitable model using their insight and expert knowledge, and accelerates the iterative process of model modification. Moreover, probabilistic programming languages make it possible to implement and publish novel learning and inference algorithms in the form of generic inference engines. This enables fair direct comparison between new and existing learning and inference algorithms on the same set of problems, something that is sorely needed by the scientific community. Furthermore, open problems that cannot be solved by state-of-the-art algorithms can be published in the form of challenging problem sets, allowing inference experts to easily identify open research questions in the field.

In this work, we introduce a system for probabilistic machine learning called `Turing`. `Turing` is an expressive probabilistic programming language developed with a focus on intuitive modelling syntax and inference efficiency. The organisation of the remainder of this paper is as follows. Section 2 sets up the problem and notation. Section 3 describes the proposed inference engines and Section 4 describe some techniques used for the implementation of proposed engines. Section 5 discusses some related work. Section 6 concludes.

2 Background

In probabilistic modelling, we are often interested in the problem of simulating from a probability distribution $p(\theta \mid y, \gamma)$. Here, θ could represent the parameters of interest, y some observed data and γ some fixed model hyper-parameters. The target distribution $p(\theta \mid y, \gamma)$ arises from conditioning a probabilistic model $p(y, \theta \mid \gamma)$ on some observed data y.

2.1 Model as computer programs

One way to represent a probabilistic model is by using a computer program. Perhaps the earliest and most influential probabilistic programming system so far is BUGS [Lunn et al., 2000]. The BUGS language dates back to 1990's. In BUGS, a probabilistic model is encoded using a simple programming language conveniently resembling statistical notations. After specifying a BUGS model and conditioning on some observed data, Monte Carlo samples can be automatically drawn from the model's posterior distribution. Algorithm 2.1 shows the generic structure of a probabilistic program.

Algorithm 2.1. A generic probabilistic program.

Input: data y and hyper-parameter γ
Step 1: Define global parameters

$$\theta^{\text{global}} \sim p(\cdot \mid \gamma); \tag{1}$$

Step 2: For each observation y_n, define (local) latent variables and compute likelihoods

$$\theta_n^{\text{local}} \sim p(\cdot \mid \theta_{1:n-1}^{\text{local}}, \theta^{\text{global}}, \gamma) \tag{2}$$
$$y_n \sim p(\cdot \mid \theta_{1:n}^{\text{local}}, \theta^{\text{global}}, \gamma) \tag{3}$$

where $n = 1, 2, \ldots, N$.

Above model variables (or parameters) are divided into two groups: θ_n^{local} denotes model parameters (or latent variables) specific to observation y_n, such as a mixture indicator for a data item in mixture models, and θ^{global} denote global parameters.

Currently there are two main approaches to probabilistic programming. The first approach is based on the idea that probabilistic programs should only support the family of models we can perform efficient inference. Although motivated from a pragmatic point of view, this approach has leads to a fruitful collection of software systems including BUGS, Stan [Stan Development Team, 2014], and Infer.NET [Minka et al., 2014].

The second approach to probabilistic programming relaxes constraints imposed by existing inference algorithms, and attempts to introduce languages that are flexible enough to encode arbitrary probabilistic models.

2.2 Inference for probabilistic programs

Probabilistic programs can only realize their flexibility potential when accompanied with efficient inference engines. To explain how inference in probabilistic programming works, we consider the the following HMM example with K states:

$$\begin{aligned} \pi_k &\sim \mathcal{D}\text{ir}(\theta) \\ \phi_k &\sim p(\gamma) \qquad (k = 1, 2, \ldots, K) \\ z_t \mid z_{t-1} &\sim \mathcal{C}\text{at}(\cdot \mid \pi_{z_{t-1}}) \\ y_t \mid z_t &\sim h(\cdot \mid \phi_{z_t}) \quad (t = 1, 2, \ldots, N) \end{aligned} \tag{4}$$

Here $\mathcal{D}$ir and $\mathcal{C}$at denote the Dirichlet and Categorical distribution respectively. The complete collection of parameters in this model is $\{\pi_{1:K}, \phi_{1:K}, z_{1:T}\}$. An efficient Gibbs sampler with the following series of three steps is often used for Bayesian inference:

Step 1: Sample $z_{1:T} \sim z_{1:T} \mid \phi_{1:K}, \pi_{1:K}, y_{1:T}$;
Step 2: Sample $\phi_k \sim \phi_k \mid z_{1:T}, y_{1:T}, \gamma$;
Step 3: Sample $\pi_k \sim \pi_k \mid z_{1:T}, \theta$ $(k = 1, \ldots, K)$.

2.3 Computation graph based inference

One challenge of performing inference for probabilistic programs is about how to obtain the computation graph between model variables. This is in contrast with graphical models, where the dependence structure (or computation graph) is normally static and known ahead of time. For example, consider variables in the HMM model, the chain $z_1, z_2, \ldots, z_T$ is dependent, while other variables are independent given this chain. However, when the HMM model is represented in the form of a probabilistic program (i.e.$\theta = \{\pi_{1:K}, \phi_{1:K}, z_{1:T}\}$, cf Algorithm 2.1), we no longer have the computation graph between model parameters.

For certain probabilistic programs, it is possible to construct the computation graph between variables through static analysis. This is the approach taken by the BUGS language [Lunn et al., 2000] and infer.NET [Minka et al., 2014]. Once the computation graph is constructed, a Gibbs sampler or message passing algorithm [Minka, 2001, Winn and Bishop, 2005] can

Sampler	Support discrete variables?	Require gradients?	Require adaption?	Support universal programs?	Composable MCMC operator?
HMC	No	Yes	Yes	No	Yes
NUTS	No	Yes	Yes	No	Yes
IS	Yes	No	No	Yes	No
SMC	Yes	No	No	Yes	No
PG	Yes	No	No	Yes	Yes
PMMH	Yes	No	No	Yes	Yes
IPMCMC	Yes	No	No	Yes	Yes

Table 1: Supported Monte Carlo algorithms in Turing (v0.4).

be applied to each random node of the computation graph. However, one caveat of this approach is that the computation graph underlying a probabilistic program needs to be fixed during inference time. For programs involving stochastic branches, this requirement may not be satisfied. In such cases, we have to resort to other inference methods.

2.4 Hamiltonian Monte Carlo based inference

For the family of models whose log probability is pointwise computable and differentiable, there exists an efficient sampling method using Hamiltonian dynamics. Developed as an extension of the Metropolis algorithm, Hamiltonian Monte Carlo [Neal et al., 2011, HMC] uses Hamiltonian dynamics to produce minimally correlated proposals. Within HMC, the slow exploration of the state space, originating from the diffusive behavior of MH's random-walk proposals, is avoided by augmenting the state space of the target distribution $p(\theta)$ with a d-dimensional vector r. The resulting joint distribution is as follows:

$$p(\theta, r) = p(\theta)p(r), \qquad p(r) = \mathcal{N}(0, I_D) \tag{5}$$

HMC operates through alternating between two types of proposals. The first proposal randomizes r (also known as the momentum variable), leaving the state θ unchanged. The second proposal changes both θ and r using simulated Hamiltonian dynamics as define by

$$H(\theta, r) = E(\theta) + K(r) \tag{6}$$

where $E(\theta) \equiv -\log p(\theta)$, $K(r)$ is a 'kinetic energy' such as $K(r) = r^T r/2$. These two proposals would produce samples from the joint distribution $p(\theta, r)$. The distribution $p(\theta, r)$ is separable, so the marginal distribution of θ is the desired distribution $p(\theta)$. Hence, we can obtain a sequence of samples for θ by simply discarding the momentum variables r.

The limitation of HMC is that it cannot sample from distributions that are not differentiable, or involving discrete variables. Probabilistic programming languages with stochastic branches [Goodman et al., 2008, Mansinghka et al., 2014, Wood et al., 2014], such as conditionals, loops and recursions, poses a substantially more challenging Bayesian inference problem because inference engines have to manage varying number of model dimensions, dynamic computation graph and so on.

2.5 Simulation based inference

Currently, most inference engines for universal probabilistic programs (those involving stochastic branches) use forward-simulation based sampling methods such as rejection sampling (RS), sequential Monte Carlo (SMC), and particle MCMC [Andrieu et al., 2010]. Goodman et al. [2008] first proposed to use RS to performance inference in universal probabilistic programming languages. Although mathematically sound, RS is notorious for its poor performance in high dimensional problems. Wood et al. [2014] were first to use SMC and particle MCMC to sample from complex target (posterior) distributions defined by universal probabilistic programs. The key to this application of particle MCMC is to note that a probabilistic program implicitly defines a high-order Markovian state space model, with a potentially unbounded order of dependency between model variables. It is worth noting that in order to simplify implementation, almost all existing implementations of these sampling algorithms use the (conditional) prior as proposals.

3 Composable MCMC inference

Our proposed composable inference method makes use of the HMC algorithm in Section 2.4 and the particle Gibbs (PG) algorithm in Section 2.5. In order to describe the proposed method for probabilistic programs, we consider Latent Dirichlet Allocation (LDA) as a working example.

Algorithm 3.1. Turing code for the LDA model.

```
@model lda(K,M,N,w,d,β,α) = begin
  θ = Vector{Vector{Real}}(M)
  for m = 1:M
    θ[m] ∼ Dirichlet(α)
  end

  ϕ = Vector{Vector{Real}}(K)
  for k = 1:K
    ϕ[k] ∼ Dirichlet(β)
  end

  z = tzeros(Int, N)
  for n = 1:N
    z[n] ∼ Categorical(θ[d[n]])
    w[n] ∼ Categorical(ϕ[z[n]])
  end
end
```

In algorithm 3.1, variables θ, ϕ and z denote model parameters, variables K,M, N, d, β and α denote hyperparameters and w denote observed data. Once a model is defined, providing data and performing inference is intuitive.

```
model = lda(K,V,M,N,w,d,β,α)
sample(model, engine)
```

Here `model` is the variable name referencing a instantiated model with data, `engine` is an MCMC engine that we want to use. For example, in order to run a particle Gibbs algorithm to sample all model parameters, we could use:

```
spl = PG(n,m)
sample(model, spl)
```

This would run a PG sampler with n particles for m iterations. Alternatively, if we want to run a blocked Gibbs sampler on the LDA model in Turing, we could use:

```
spl2 = Gibbs(1000,PG(10,2,:z),HMC(2,0.1,5,:ϕ,:θ))
sample(model, spl2)
```

It is worth noting that in the `spl2` engine we have split model variables into two subsets and assigned these subsets to two samplers PG and HMC. In general, we could split of variables in many different ways. In engine `spl`, we split them according to whether a parameter is differentiable w.r.t. to the target distribution so that we can run a HMC engine on variable θ and ϕ, and a PG engine on variable z.

In this work, we assume the split of these variables is performed manually by users and fed to the inference engine. It is worth pointing out the split of variables does not need to be mutual exclusive (i.e. we can split parameters into overlapping subsets) as long as the union of all subsets contains all model parameters. Given a split, we can sample the variables in a series of MCMC steps, e.g.:

Step 1: Sample z using a particle Gibbs sampler

$$z \sim z \mid \theta, \alpha, \beta, K, V, M, N, w, d \tag{7}$$

Step 2: Sample θ using a Hamiltonian Monte Carlo sampler

$$\theta \sim \theta \mid z, \alpha, \beta, K, V, M, N, w, d \tag{8}$$

In general, even in cases where we can sample all model parameters using the same engine, splitting variables could be beneficial. For example, we could split model parameters in several subsets, and run a separate HMC engine with a different configuration on each subset of parameters: `Gibbs(HMC(:`θ_1`), HMC(:`θ_2`)`. Such split HMC algorithm could be helpful for faster exploration of target distributions at low computation cost in logistic regression [Neal et al., 2011].

3.1 A family of MCMC operators

Turing supports a wide range of sampling algorithms include Hamiltonian Monte Carlo, particle Gibbs, No-U-Turns, particle Marginal MH, and interactive particle MCMC (see Table 1). Each algorithm can be used as a component algorithm for building a desirable MCMC algorithm.

Importantly, the HMC are implemented in a generic way such that they support models with dynamic dimensions, constrained variables (e.g. truncated Gaussian) and vectorized computation of certain independent random variables. We also implemented the NUTS variant of HMC. Some further implementation details are presented in section 4.2. Below we show a few example engines built using these MCMC operators.

- `Gibbs(HMC(stepSize, nLeapfrog, :θ), PG(nParticle, nIter, :z))`
- `Gibbs(NUTS(stepSize, nLeapfrog, :θ, :z), PG(nParticle, nIter, :z))`
- `Gibbs(PMMH(SMC(nParticles, :z), :θ), HMC(stepSize, nLeapfrog, :θ, :z), PG(nParticle, nIter, :z))`

One advantage of composing MCMC algorithms is that it can combine the flexibility and efficiency of different sampling methods. For example, particle Gibbs is a generic sampler that can be applied to arbitrary programs including those with stochastic branches, while HMC is a sampler that is most efficient when sampling differentiable model parameters. HMC alone is not a valid inference algorithms for programs with discrete variables, but when used together with PG, we have both the universality of PG and efficiency of HMC. Furthermore, new samplers can be straightforwardly implemented in Turing. In fact, some MCMC operators are already contributed by community developers, such as PMMH [Andrieu et al., 2010] and IPMCMC [Rainforth et al., 2016] in Table 1.

4 Implementation and Experiments

4.1 The Turing library

`Turing` is implemented as a library for the `Julia` language. `Turing` models are defined by normal Julia programs supported by two probabilistic operations: 'assume' and 'observe'. Both of these operations are indicated by the tilde operator, e.g. $x \sim \texttt{Normal}(0, 1)$, where the left hand side is a variable name and the right hand side is a distribution. Distributions are declared in their standard mathematical form, e.g. $\texttt{Normal}(0, 1)$ or $\texttt{Beta}(2, 3)$.

Since `Turing` programs are normal `Julia` programs, they can utilize the rich numerical and statistical libraries of Julia. For example, the distributions package provides a comprehensive collection of probability functions. The forward mode automatic differentiation library supports all continuous distributions found within `Julia`'s distribution package.

4.1.1 Efficient particle Gibbs implementation

One critical implementation detail about particle Gibbs engine is the duplication of particles during the resampling step of SMC. We define running program states as the sequence of memory states (ie $(\theta, \mathbf{z}_{1:N})$) that arises during the sequential execution of probabilistic programs. Monte Carlo based inference engines operates on the space of program states, and requires manipulating (e.g. fork, discard or mutate) program states to implement certain operations such as accepting/rejecting MCMC proposals or resampling SMC particles.

We use *coroutine* for implementing particle Gibbs. Coroutines can be viewed as a generalization of a functions, with the property that they can be suspended and resumed at multiple points. In essence coroutines are a lightweight tool for designing programs in terms of several interacting processes without introducing the overhead or concurrency problems associated with running multiple processes simultaneously.

4.1.2 Automatic differentiation

Simulating Hamiltonian dynamics for the HMC step requires the gradient of $\log p(\theta \mid \mathbf{z}_{1:N}, \gamma)$. These gradients can be obtained automatically when given a computer program defining $\log p(\theta \mid \mathbf{z}_{1:N}, \gamma)$, through automatic differentiation (AD) techniques [Baydin et al., 2015]. In this work, we first make use of a technique called vector mode forward differentiation for its simplicity and efficiency. The main concept behind the vector mode forward differentiation is the *multidimensional dual number*, whose behavior on scalar functions is defined by:

$$f(\theta + \sum_{i=1}^{D} y_i \epsilon_i) = f(\theta) + f'(\theta) \sum_{i=1}^{D} y_i \epsilon_i \tag{9}$$

where $\epsilon_i \epsilon_j = 0$ for $i \neq j$ $(i, j = 1, \ldots, D)$. Through storing more ϵ components we can perform a vector forward-mode implementation of the AD algorithm developed by Khan and Barton [2015].

More generally, for n input vector of length D and a chunk size N, it takes $\lceil \frac{D}{N} \rceil$ passes through function f to compute $\texttt{grad}\, f(\theta)$. So, using bigger chunk size N reduces the passes through f, but at the cost of additional memory.

Vector mode forward AD is very efficient for small models (e.g. models with less than 50 parameters). However, when models get bigger, reverse mode AD becomes more efficient since it only requires evaluating the target function once (probability distribution functions have only one output). For these reasons, Turing supports both forward mode and reverse mode AD. Since the AD implementations used by Turing are generic, any `Julia` library can straightforwardly make their function differentiable through very minimal or no changes of its code. This means any `Julia` library can make uses of `Turing`'s MCMC engines[1].

4.1.3 Vectorized random variables

Turing also supports vectorized sampling for i.i.d variables using the following syntax:

```
rv = Vector(10)
rv ~ [Normal(0, 1)]
```

Here `rv` is defined as a vector with each element following a Normal distribution.

Underlying this vectorized random variable syntax, `Turing` calls the corresponding random number generator that generates multiple random numbers or evaluates the density on multiple inputs, in parallel. This vectorization feature can substantially speed-up runtime of probabilistic models with i.i.d. variables, which are common in probabilistic modeling.

[1]Supplementary materials contain a notebook, `diff-eq.ipynb` illustrating how to use `Turing` in conjunction with `DifferentialEquation.jl` to perform Bayesian inference on the parameters of a differential equation when given noisy observations.

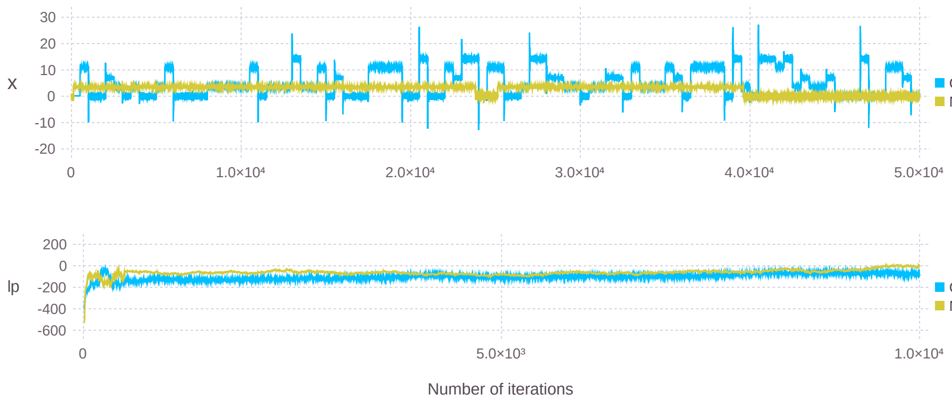

Figure 1: Trace plot of 50,0000 samples from a mixture of Gaussian (top) and the log joint-probability for 10,000 iterations of the SV models run by NUTS and Gibbs engines (bottom). Here NUTS, for both examples, uses 10,000 iterations to adapt the step-size; PG-as-Gibbs-component uses 5 particles for the GMM example and 50 for the SV model. Note that thinning is performed here in Gibbs, i.e. intermediate samples within Gibbs steps are dropped, for the SV model, but is disabled, i.e. intermediate PG and HMC samples are collected, for the mixture example.

4.1.4 Constrained random variables

Random variables can follow particular distributions with constrained support. Before performing any MH based sampling algorithm (like HMC) on models containing such constrained variables, `Turing` transforms them into un-constrained Euclidean space.

More specifically, there are three types of constraints. The first constraint type contains univariate random variables with bounds, e.g. random variables that follow a Gamma distribution are positive, or those following a Beta distribution are between 0 and 1. The second constraint type includes multivariate variables with simplex constraints, i.e. elements of the vector should sum up to 1. A typical example is the Dirichlet distribution. Finally the third type of constraints comes from matrix-variate distributions. For instance, random variables following a Wishart distribution should be positive-definite.

4.1.5 MCMC output analysis

In `Turing`, the following statistics for each MCMC chain can be calculated by calling the `describe` function: 1) mean, 2) standard deviation, 3) naive standard error, 4) Monte Carlo standard error (MCSE), 5) effective sample size (ESS), and 6) quantiles of 2.5%, 25.0%, 50.0%, 75.0% and 97.5%. In addition to these basic quantities, highest posterior density intervals can be computed by `hpd`, cross-correlations by `cor`, lag-autocorrelations by `autocor`, state space change rate (per iteration) by `changerate` and deviance information criterion by `dic`.

4.2 Finding the right inference engine

To demonstrate how to explore different inference engines for a certain model, we performed an illustrative comparison of the `NUTS` and `Gibbs(PG, HMC)` engine on several probabilistic models. We only report two scenarios that we found to be challenging for the NUTS engine to perform well. The models used together with inference engine configuration are briefly described in the coming section. Full code to reproduce the results is provided as supplementary material.

4.2.1 Models and inference engine setup

Stochastic Volatility Model: the collection of parameters is $\{\phi, \sigma, \mu, h_{1:N}\}$. All these parameters are differentiable with respect to the target distribution, so the NUTS algorithm is directly applicable.

$$\begin{aligned}
&\mu \sim \mathcal{C}\mathrm{a}(0,10)),\ \phi \sim \mathcal{U}\mathrm{n}(-1,1),\ \sigma \sim \mathcal{C}\mathrm{a}(0,5), \quad (\sigma > 0)\\
&h_1 \sim \mathcal{N}(\mu, \sigma/\sqrt{1-\phi^2}),\ h_n \sim \mathcal{N}(\mu + \phi(h_{n-1} - \mu), \sigma)\\
&y_n \sim \mathcal{N}(0, \exp(h_n/2)) \qquad (n = 2, 3, \ldots, N).
\end{aligned} \tag{10}$$

Here $\mathcal{C}$a and $\mathcal{U}$n denote the Cauchy and Uniform distribution respectively. In our experiments, we use the following inference engines

```
spl1 = NUTS(1e4, 1e3, 0.65)
spl2 = Gibbs(1e4, PG(5, 1, :h),
             NUTS(1,1e3,0.65,:μ,:ϕ,:σ))
```

More specifically, for NUTS 1000 adaption steps are

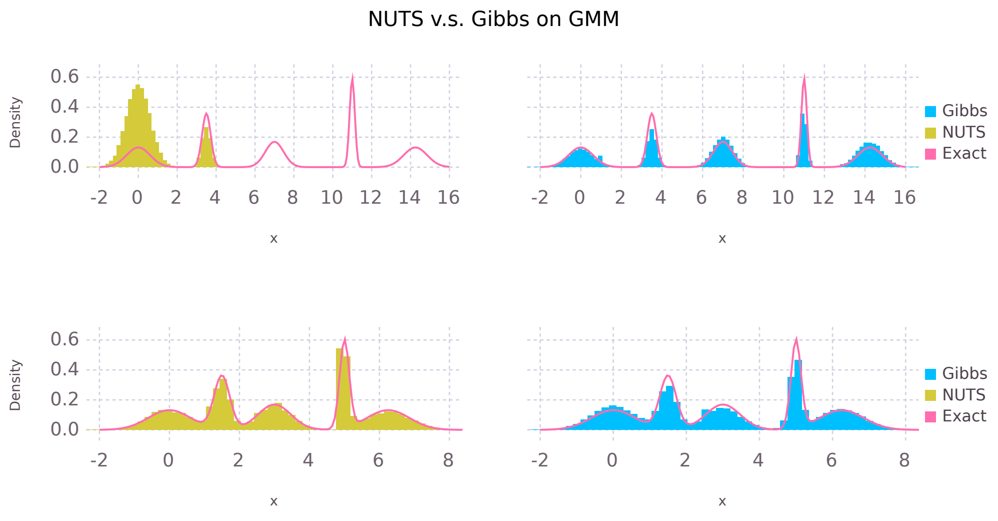

Figure 2: Density estimation of 50,000 samples from a GMM with 5 Gaussian components by NUTS and Gibbs. Here each Gibbs step consists of a PG step using 5 particles, and an NUTS step with 500 iterations. Bars are normalized histogram of samples and the solid line is the exact density of the underlying GMM.

used for warm-up, 0.65 is the adaption target for dual averaging.

Gaussian Mixture Model: the collection of parameters of interest is $\{z, \theta\}$, where parameter θ is differentiable, parameter z is not. To run the NUTS algorithm, we integrate out z and sample θ only.

$$\begin{aligned} \mu = (\mu_{1:K}), \quad \sigma = (\sigma_{1:K}), \quad \pi = (p_{1:K}) \\ z \sim \mathcal{C}\text{at}(\pi), \quad \theta \sim \mathcal{N}(\mu_z, \sigma_z) \end{aligned} \tag{11}$$

We use the following inference engines:

```
spl3 = NUTS(5e4, 1000, 0.65)
spl4 = Gibbs(5e4,PG(5,1,:z),
                  NUTS(5e2,1e3,0.65,:θ))
```

In this group of experiment, the above GMM model with different sets of parameters are used: the first GMM model has 5 mixtures that are close to each other, while the second as well separated mixtures, each covering a small region of the sampling space. For each GMM, we collect 10,000 samples using using either the Gibbs or the NUTS engine.

4.2.2 Results

Figure 1 shows a trace of parameter θ for GMM model (top), and a trace of the samples' log-likelihoods for the SV model (bottom). Figure 2 shows the estimated histogram for each engine together with the exact density. For the SV model, the trace shows that `NUTS` and `Gibbs(PG, HMC)` engine have converged to similar values of log-likelihoods. However, for the GMM model, the `NUTS` engine seem get stuck in some modes of the target distribution, as shown in the top panel of fig. 1. This behavior can also be seen from the samples histogram in fig. 2. The `NUTS` engine only explores 2 modes for the GMM whose components are far from each other, while leaving the other 3 modes unidentified. The `Gibbs` engine seems to successfully explore all high probability regions.

4.3 A simple runtime comparison between Stan and Turing

Table 2 shows some illustrative benchmarking results between `Turing` and `Stan`, using the same HMC algorithm on a variety of popular machine learning models[2]. Overall, Turing is between 0.7 to 20 times slower than Stan. In general, these runtime numbers are very sensitive to specific model implementations: the same LDA model can be implemented in several different ways in `Turing`, because in `Turing` a user can use all language features and numerical capabilities from `Julia`. After trying a few different implementations for LDA, we could find a version that is substantially faster than others. In fact, the fastest version is faster then the corresponding high optimized `Stan` implementation. We believe that this is an important advantage of building a probabilistic programming system through embedding

[2]This benchmark can be reproduced using the script in the benchmark folder of `Turing`'s repository.

Table 2: Runtime comparison between Turing vs Stan using the same HMC sampling algorithm. For Turing, both forward mode (F) and reverse mode (R) are used. For Stan, only reverse mode AD is used. Ratio (R) is the runtime ratio between reverse mode Turing and reverse mode Stan, while Ratio (F) is the runtime ratio between forward mode Turing and reverse mode Stan. For models with more than 100,000 dimensions, we were unable to run HMC with forward AD. These runtime numbers should not be considered as serious benchmarking results since the runtime is highly model and implementation dependent.

Model	Dimensionality	Time (R)		Ratio (R)	Ratio (F)
		Turing (s)	Stan (s)		
High-dimensional Gaussian	100,000	351.4 ± 15.06	90.31 ± 0.23	4.38	—
Latent Dirichlet Allocation	550	156.8 ± 7.79	205.3 ± 2.41	0.76	7.74
Naive Bayes	400	630.4 ± 2.65	37.27 ± 0.42	16.91	152.21
Stochastic Volatility	100,003	12.04 ± 0.65	0.58 ± 0.02	20.87	—
Hidden Markov Model	275	274.97 ± 2.97	21.85 ± 0.09	12.58	324.67

in a high performance general-purpose programming language than creating new languages which are often limited by language features and numerical capabilities.

5 Related Work

The closest related work to composable inference is that of Mansinghka et al. [2014] and Zinkov and Shan [2016], which share the goal of composing MCMC algorithms for probabilistic inference. Our particle Gibbs engine is a new implementation of Wood et al. [2014], Paige and Wood [2014], but with a more efficient mechanism (coroutines) for representing particles. In addition, our particle Gibbs could be used in conjunction with other inference engines including HMC.

We purposely designed `Turing`'s syntax to be similar to `BUGS` [Lunn et al., 2000] and `Stan` [Gelman et al., 2015], however `Turing` is more expressive than BUGS and Stan because it supports stochastic branches. Moreover, `Turing` has a much smaller code base because many functionalities of `Turing`, such as automatic–differentiation and probability functions are based on `Julia`'s rich numerical capacities.

6 Conclusion

In this work we have presented a machine learning language that supports flexible composable probabilistic inference. The proposed language supports a wide range of basic Markov chain sampling algorithms. Some of these sampling algorithms can be used as black-box style inference methods while others can be used as part of a composed inference engine. In order to illustrate the utility of the proposed language, we empirically compared several inference engines for several popular machine learning models.

In particular, we compared a HMC engine that implements the NUTS algorithm [Hoffman and Gelman, 2014], and a Gibbs engine, which samples model parameters through iterating a series of particle Gibbs and HMC updates. The Gibbs engine, although motivated to sample model parameters that are not differentiable w.r.t. the target distribution, turns out to be a helpful complement for the HMC engine for sampling target distributions with isolated modes. Another advantage of the composed Gibbs engine is that it can be universal, i.e. applicable to arbitrary probabilistic models including those involving discrete variables and stochastic control flows.

Furthermore, the current Gibbs engine implementation in `Turing` is suboptimal and requires a full sweep of the model function in order to update one variable. This inefficiency can be addressed by leveraging dynamic computational graphs to explicitly represent dependency relationships between model variables. We leave this development and application as an avenue for future work.

Acknowledgement

HG and ZG acknowledge support from the Alan Turing Institute (EPSRC Grant EP/N510129/1) and EPSRC Grant EP/N014162/1, and donations from Google and Microsoft Research. Since Turing is an open source project, we have received many contributions from volunteers around the world. We would like to thank all contributors for their valuable work.

References

Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767*, 2015.

Christopher M Bishop. Model-based machine learning. *Phil. Trans. R. Soc. A*, 371(1984):20120222, 2013.

Andrew Gelman, Daniel Lee, and Jiqiang Guo. Stan a probabilistic programming language for bayesian inference and optimization. *Journal of Educational and Behavioral Statistics*, 2015.

Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459, 2015.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*, 2008. URL `http://danroy.org/papers/church_GooManRoyBonTen-UAI-2008.pdf`.

Matthew D Hoffman and Andrew Gelman. The no-U-turn sampler: Adaptively setting path lengths in hamiltonian Monte Carlo. *The Journal of Machine Learning Research*, 15(1):1593–1623, 2014.

Kamil A Khan and Paul I Barton. A vector forward mode of automatic differentiation for generalized derivative evaluation. *Optimization Methods and Software*, 30(6):1185–1212, 2015.

David J Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. WinBUGS – a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and computing*, 10(4):325–337, 2000.

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

T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. Infer.NET 2.6, 2014. URL `http://research.microsoft.com/infernet`. Microsoft Research Cambridge.

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.

Lawrence M. Murray. Bayesian state-space modelling on high-performance hardware using LibBi, 2013.

Lawrence M Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B Schön. Delayed sampling and automatic rao-blackwellization of probabilistic programs. *arXiv preprint arXiv:1708.07787*, 2017.

Radford M Neal et al. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2(11), 2011.

Brooks Paige and Frank Wood. A compilation target for probabilistic programming languages, 2014.

Avi Pfeffer. IBAL: A probabilistic rational programming language. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence*, pages 733–740. Morgan Kaufmann Publ., 2001.

Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, 137, 2009.

Tom Rainforth, Christian Naesseth, Fredrik Lindsten, Brooks Paige, Jan-Willem Vandemeent, Arnaud Doucet, and Frank Wood. Interacting Particle Markov Chain Monte Carlo. In *International Conference on Machine Learning*, pages 2616–2625, 2016.

Stan Development Team. Stan: A C++ library for probability and sampling, version 2.5.0, 2014. URL `http://mc-stan.org/`.

John Winn and Christopher M Bishop. Variational message passing. *Journal of Machine Learning Research*, 6(Apr):661–694, 2005.

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, 2014. URL `http://www.robots.ox.ac.uk/~fwood/anglican/assets/pdf/Wood-AISTATS-2014.pdf`.

Robert Zinkov and Chung-chieh Shan. Composing inference algorithms as program transformations. *arXiv preprint arXiv:1603.01882*, 2016.