Compiling Stan to Generative Probabilistic Languages

GUILLAUME BAUDART, MIT-IBM Watson AI Lab, IBM Research
JAVIER BURRONI, UMass Amherst
MARTIN HIRZEL, MIT-IBM Watson AI Lab, IBM Research
KIRAN KATE, MIT-IBM Watson AI Lab, IBM Research
LOUIS MANDEL, MIT-IBM Watson AI Lab, IBM Research
AVRAHAM SHINNAR, MIT-IBM Watson AI Lab, IBM Research

Stan is a popular declarative probabilistic programming language with a high-level syntax for expressing graphical models and beyond. Stan differs by nature from generative probabilistic programming languages like Church, Anglican, or Pyro. In this paper, we present a comprehensive compilation scheme to compile any Stan model to a generative language. We use this result to build a compiler from Stan to Pyro and extend extend Stan with support for explicit variational inference guides and deep probabilistic models. Overall, our paper clarifies the relationship between declarative and generative probabilistic programming languages and is a step towards making deep probabilistic programming easier.

1 INTRODUCTION

Probabilistic Programming Languages (PPLs) are designed to describe probabilistic models and run inference on these models. There exists a variety of PPLs [1, 5, 8, 9, 12, 13, 20–23, 25, 28–30, 36, 37]. *Declarative Languages* like BUGS [20], JAGS [30], or Stan [5] focus on efficiency, constraining what can be expressed to a subset of models for which fast inference techniques can be applied. This family enjoys broad adoption by the statistics and social sciences communities [4, 10, 11]. *Generative languages* like Church [12], Anglican [36], WebPPL [13], Pyro [1], and Gen [7] focus on expressivity and allow the specification of intricate models with rich control structures and complex dependencies. Generative PPLs are particularly suited for describing *generative models*, i.e., stochastic procedures that simulate the data generation process. Generative PPLs are increasingly used in machine-learning research and are rapidly incorporating new ideas, such as Stochastic Gradient Variational Inference (SVI), in what is now called Deep Probabilistic Programming [1, 37].

While the semantics of probabilistic languages have been extensively studied [14, 15, 18, 35], to the best of our knowledge little is known about the relation between the two families. This paper presents a comprehensive compilation scheme that can be used to compile any Stan program to a generative PPL. This makes it possible to leverage the rich set of existing Stan models for testing, benchmarking, or experimenting with new features or inference techniques.

In addition, recent probabilistic languages offer new features to program and reason about complex models. Our compilation scheme combined with conservative extensions of Stan can be used to make these benefits available to Stan users. As a proof of concept, this paper shows how to extend Stan with support for deep probabilistic models by compiling Stan to Pyro. This approach has the following advantages: (1) Pyro is built on top of PyTorch [27]. Programmers can thus seamlessly import neural networks designed with the state-of-the-art API provided by PyTorch. (2) Variational inference was central in the design of Pyro. Programmers can easily craft their own inference guides to run variational inference on deep probabilistic models. (3) Pyro also offers alternative inference methods, such as NUTS [16] (No U-Turn Sampler), an optimized Hamiltonian Monte-Carlo (HMC) algorithm which is the preferred inference method for Stan. We can thus validate the results of our approach against the original Stan implementation on classic probabilistic models.

```
data { int N; int<lower=0,upper=1> x[N]; }
parameters { real<lower=0,upper=1> z; }
model {
  z ~ beta(1, 1);
  for (i in 1:N) x[i] ~ bernoulli(z); }
```

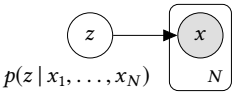


Fig. 1. Biased coin model in Stan.

To summarize, this paper makes the following contributions: (1) A comprehensive compilation scheme from Stan to a generative PPL (§2). (2) A compiler from Stan to Pyro with support for explicit variational inference guides and deep probabilistic models (§3).

2 COMPILATION

This section shows how to compile a declarative language that specifies a joint probability distribution like Stan [5] to a generative PPL like Church, Anglican, or Pyro. Translating Stan to a generative PPL also demonstrates that Stan's expressive power is at most as large as that of generative languages, a fact that was not clear before our paper.

As a running example, consider the biased coin model shown in Figure 1. This model has observed variables x_i, $i \in [1:N]$, which can be 0 for tails or 1 for heads, and a latent variable $z \in [0,1]$ for the bias of the coin. Coin flips x_i are independent and identically distributed (IID) and depend on z via a Bernoulli distribution. The prior distribution of parameter z is Beta(1, 1).

2.1 Generative translation

Generative PPLs are general-purpose languages extended with two probabilistic constructs [14, 35, 38]: `sample(`D`)` generates a sample from a distribution D and `factor(`v`)` assigns a score v to the current execution trace. Typically, `factor` is used to condition the model on input data [36]. We also introduce `observe(`D`,`v`)` as a syntactic shortcut for `factor(`D_{pdf}`(`v`))` where D_{pdf} denotes the density function of D. This construct penalizes executions according to the score of v w.r.t. D which captures the assumption that the observed data v follows the distribution D.

Compilation. Stan uses the same syntax `v ~` D for both observed and latent variables. The distinction comes from the kind of the left-hand-side variable: observed variables are declared in the `data` block, latent variable are declared in the `parameters` block. A straightforward *generative translation* compiles a statement `v ~` D into `v = sample(`D`)` if `v` is a parameter or `observe(`D`, v)` if `v` is data. For example, the Stan code from Figure 1 is compiled into (using Python syntax):

```
def model(N, x):
  z = sample(Beta(1.,1.))
  for i in range(0, N):
    observe(Bernoulli(z), x[i])
  return z
```

2.2 Non-generative features

In Stan, a model represents the unnormalized density of the joint distribution of the parameters defined in the `parameters` given the data defined in the `data` block [5, 15]. A Stan program can thus be viewed as a function from parameters and data to the value of a special variable `target` that represents the log-density of the model. A Stan model can be described using classic imperative statements, plus two special statements that modify the value of `target`. The first one, `target+=`e, increments the value of `target` by e. The second one, e `~` D, is equivalent to `target+=` $D_{\text{lpdf}}(e)$ [15].

Table 1. Stan features: example, prevalence and compilation.

Feature	%	Example	Compilation
Left expression	7.7	`sum(phi) ~ normal(0, 0.001*N);`	`observe(Normal(0.,0.001*N), sum(phi))`
Multiple updates	3.9	`phi_y ~ normal(0,sigma_py);` `phi_y ~ normal(0,sigma_pt)`	`observe(Normal(0.,sigma_py), phi_y);` `observe(Normal(0.,sigma_pt), phi_y)`
Implicit prior	60.7	`real alpha0;` `/* missing 'alpha0 ~ ...' */`	`alpha0 = sample(ImproperUniform())`
Target update	16.3	`target += -0.5 * dot_self(` `  phi[node1] - phi[node2]);`	`factor(-0.5 * dot_self(` `  phi[node1] - phi[node2]))`

Unfortunately, these constructs allow the definition of models that cannot be translated using the generative translation defined above. Specifically, Table 1 lists the Stan features that are not handled correctly. A *left expression* is a case where the left-hand-side of ~ is an arbitrary expression. The *multiple updates* feature occurs when the same parameter appears on the left-hand-side of multiple ~ statements. An *implicit prior* occurs when there is no explicit ~ statement in the model for a parameter. A *target update* is a direct update to the log-density of the model.

The "%" column of Table 1 indicates the percentage of Stan models that exercise each of the non-generative features among the 502 files in https://github.com/stan-dev/example-models. The example column contains illustrative excerpts from such models. Since these are official and long-standing examples, we assume that they use the non-generative features on purpose. Comments in the source code further corroborate that the programmer knowingly used the features. While some features only occur in a minority of models, their prevalence is too high to ignore.

2.3 Comprehensive translation

The previous section illustrates that Stan is centered around the definition of `target`, not around generating samples for parameters, which is required by generative PPLs. The idea of the comprehensive translation is to add an initialization step to generate samples for all the parameters and compile all Stan ~ statements as observations. To initialize the parameters we draw from the uniform distribution in the definition domain of the parameters. For the biased coin example, this translation yields:

```
def model(N, x):
  z = sample(Uniform(0.,1.))
  observe(Beta(1.,1.), z)
  for i in range(0, N):
    observe(Bernoulli(z), x[i])
  return z
```

The compilation column of Table 1 illustrates the translation of non-generative features. Left expression and multiple updates are simply compiled into observations. Parameter initialization uses the uniform distribution over its definition domain. For unbounded domains, we introduce new distributions (e.g., `ImproperUniform`) with a constant density that can be normalized away during inference. Compiling the rest of the language does not raise additional difficulties (see §A).

Correctness. The semantics of Stan as described in [15] is the semantics of a classical imperative language that defines an environment containing, in particular, the value of the special variable `target`: the unnormalized log-density of the model. On the other hand, the semantics of a generative PPL as described in [35] defines a kernel mapping an environment to a measurable function. Our

compilation scheme adds uniform initializations for all parameters which comes down to the Lebesgue measure on the parameters space, and translates all ~ statements to `observe` statements. We can then show that succession of `observe` statements yields a distribution with the same log-density as the Stan semantics.

Implementation. The comprehensive compilation scheme can be used to compile any Stan program to a generative PPL leveraging the rich set of existing Stan models for testing, benchmarking, or experimenting with new features or inference techniques. As a proof of concept, we implemented a compiler from Stan to Pyro.

Built as a Python library on top of PyTorch, Pyro is described as a *deep universal probabilistic programming language* in the line of WebPPL. Probabilistic models may involve deep neural networks to capture complex relations between parameters. Pyro was also designed to natively support *variational inference* (see §3.1).

In Pyro, *sample* is `v = sample(name,`D`)`, and *observe* is `sample(name,`D`,obs=`e`)`. In both cases, the user must specify a unique `name` as a Python string. Translating *target update* requires overcoming the obstacle that Pyro does not directly expose the log-density accumulator to the programmer. Instead, we use the exponential distribution with parameter $\lambda = 1$ whose density function is $Exp_{\text{pdf}}(1)(x) = e^{-x}$. Observing a value $-v$ from this distribution multiplies the score by e^v, which corresponds to the update *target* += v. This approach is similar to the "zeros trick" from BUGS [19]. Finally, compared to Stan, Pyro does not use the context to lift constants to vectors (e.g., `x ~ normal(0, 1)` where `x` is of type `real[N]`). Shapes thus need to be computed by the compiler.

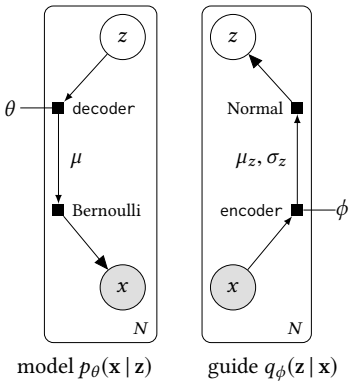
3 EXTENDING STAN: EXPLICIT VARIATIONAL GUIDES AND NEURAL NETWORKS

Recent probabilistic languages like Pyro offer new features to program and reason about complex models. This section shows that our compilation scheme combined with conservative syntax extensions can be used to lift these benefits for Stan users. Building on Pyro, we propose DeepStan, an extension of Stan with: (1) variational inference with high-level but explicit guides, and (2) a clean interface to neural networks written in PyTorch. From another perspective, we contribute a new frontend for Pyro that is high-level and self-contained, with hundreds of Stan models ready to try (§C evaluates DeepStan on multiple examples).

3.1 Explicit variational guides

Variational Inference (VI) tries to find the member $q_{\theta^*}(z)$ of a family $\mathcal{Q} = \{q_\theta(z)\}_{\theta \in \Theta}$ of simpler distributions that is the closest to the true posterior $p(z \mid \mathbf{x})$ [2]. Members of the family $\mathcal{Q}$ are characterized by the values of the *variational parameters* θ. The fitness of a candidate is measured using the Kullback-Leibler (KL) divergence from the true posterior, which VI aims to minimize $q_{\theta^*}(z) = \text{argmin}_{\theta \in \Theta} \text{KL}\Big(q_\theta(z) \,||\, p(z \mid \mathbf{x})\Big)$. Pyro natively support variational inference and lets users define the family $\mathcal{Q}$ (the *variational guide*) alongside the model. To make this feature available for Stan users, we extend Stan with two new optional blocks: `guide parameters` and `guide`. The `guide` block defines a distribution parameterized by the `guide parameters`. Variational inference then optimizes the values of these parameters to approximate the true posterior.

DeepStan inherits restrictions for the definition of the guide from Pyro: the guide must be defined on the same parameter space as the model, i.e., it must sample all the parameters of the model; and the guide should also describe a distribution from which we can directly generate valid samples without running the inference first, which prevents the use of `observe` statements. Our compiler checks these restrictions statically for early error reporting. Once these conditions are verified, the generative translation from Section 2.1 generates a Python function that can serve as a Pyro guide. The `guide parameters` block is used to generate Pyro `param` statements, which introduce




```
networks {
  Decoder decoder; Encoder encoder; }
data {
  int nz;
  int<lower=0, upper=1> x[28, 28]; }
parameters { real z[_]; }
model {
  real mu[_, _];
  z ~ normal(0, 1);
  mu = decoder(z);
  x ~ bernoulli(mu); }
guide {
  real encoded[2, nz] = encoder(x);
  real mu_z[_] = encoded[1];
  real sigma_z[_] = encoded[2];
  z ~ normal(mu_z, sigma_z); }
```

Fig. 2. Graphical models and DeepStan code of the Variational Auto-Encoder model and guide.

PyTorch learnable parameters. Unlike Stan parameters that define random variables for use in the model, guide parameters are learnable coefficients that will be optimized during inference.

The restrictions imposed on the guide do not prevent the guide from being highly sophisticated. For instance, the following section shows an example of a guide defined with a neural network.

3.2 Adding neural networks

One of the main advantages of Pyro is its tight integration with PyTorch which allows the authoring of *deep probabilistic models*, that is, probabilistic models involving neural networks. In comparison, it is impractical to define neural networks directly in Stan. To make this feature available for Stan users, we extend Stan with an optional block `networks` to import neural network definitions.

Neural networks can be used to capture intricate dynamics between random variables. An example is the *Variational Auto-Encoder* (VAE) illustrated in Figure 2. A VAE learns a vector-space representation z for each observed data point x (e.g., the pixels of an image) [17, 31]. Each data point x depends on the latent representation z in a complex non-linear way, via a deep neural network: the *decoder*. The leftmost part of Figure 2 shows the corresponding graphical model. The output of the decoder is a vector μ that parameterizes a Bernoulli distribution over each dimension of x (e.g., each pixel is associated to a probability of being present in the image).

The main idea of the VAE is to use variational inference to learn the latent representation. The guide maps each x to a latent variable z via another neural network: the *encoder*. The middle part of Figure 2 shows the graphical model of the guide. The encoder returns, for each input x, the parameters μ_z and σ_z of a Gaussian distribution in the latent space. Then inference tries to learn good values for the parameters θ and ϕ, simultaneously training the decoder and the encoder.

The right part of Figure 2 shows the corresponding code in DeepStan. A network is introduced by the name of its class and an identifier. This identifier can then be used in subsequent blocks, in particular the `model` block and the `guide` block. The network class must be implemented in PyTorch and the associated variable must be a valid instance of the class.

Neural networks can also be treated as probabilistic models. A *Bayesian neural network* is a neural network whose learnable parameters (weights and biases) are random variables, instead of concrete values [26]. Building on Pyro features, we make it easy for users to *lift* neural networks, i.e., replace concrete neural network parameters by random variables (see details in §B.1).

Tensor dimension analysis. Since model code spans both Stan and PyTorch, one challenge is to minimize redundancy while checking for errors and generating efficient code. As shown in Figure 2 we let users elide some of the concrete tensor dimensions by writing the wildcard "_" instead. Types that use the wildcard have their size and number of dimensions automatically filled in by the compiler. The DeepStan compiler needs to derive information about dimensions for two purposes: error checking and code generation. It deduces the tensor dimensions with a Hindley-Milner style polymorphic type analysis [24]. In Figure 2, the compiler is able to compute the shape of parameters `z` (`nz`), `mu` (28, 28), `mu_z` and `sigma_z` (`nz`) (see details in §B.2).

3.3 Experiments

We tested our compilation scheme on 6 classical probabilistic models taken from the set of Stan examples. We run inference on the generated Pyro code using NUTS (No U-Turn Sampler [16], an optimized HMC which is the preferred inference method for Stan, and verified that the results were consistent with those of Stan. We also showed on a simple multimodal example that, sometimes, using explicit VI gives comparable or even more accurate results (see §C.1).

Since Stan lacks support for deep probabilistic models, we compared the performance of the code generated by our compiler with hand-written Pyro code on the VAE described in §3.2 and a simple Bayesian neural network. In both cases, we found comparable results (see §C.2).

4 RELATED WORK

To the best of our knowledge, we propose the first comprehensive translation of Stan to a generative PPL. The closest related work has been developed by the Pyro team concurrently [3] to our work [6]. Their work focuses on performance and our work on completeness. Their proposed compilation technique corresponds to the generative translation presented in §2.1 and thus only handles a subset of Stan. Compared to our approach, they are also looking into independence assumptions between loop iterations to generate parallel code. Combining these ideas with our approach is a possible future direction. They do not extend Stan with VI and neural networks.

The goal of compiling Stan to Pyro is not to replace Stan's highly optimized inference engine [16], but rather to create a platform for experimenting with new ideas. As an example, we showed in §3.1 how to extend Stan with explicit variational guides. In the same vein, Pyro now offers tools for automatic variational guide synthesis [2] that can now be tested on existing Stan models.

In recent years, taking advantage of the maturity of DL frameworks, multiple deep probabilistic programming languages have been proposed: Edward [37] and ZhuSuan [33] built on top of TensorFlow, Pyro [1] and ProbTorch [34] built on top of PyTorch, and PyMC3 [32] built on top of Theano. All these languages are implemented as libraries. The users thus need to master the entire technology stack of the library, the underlying DL framework, and the host language. In comparison, DeepStan is a self-contained language and the compiler helps the programmer via dedicated static analyses (e.g., the tensor dimension analysis of §3.2).

5 CONCLUSION

This paper presents a comprehensive compilation scheme from Stan to any generative probabilistic programming language. We thus show that Stan is at most as expressive as this family of languages. To validate our approach we implemented a compiler from Stan to Pyro. Additionally, we designed and implemented extensions for Stan with explicit variational guides and an interface with PyTorch.

REFERENCES

[1] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. https://arxiv.org/abs/1810.09538

[2] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. 2017. Variational Inference: A Review for Statisticians. *J. Amer. Statist. Assoc.* 112 (2017), 859–877. Issue 518.

[3] Javier Burroni, Guillaume Baudart, Louis Mandel, Martin Hirzel, and Avraham Shinnar. 2018. Extending Stan for Deep Probabilistic Programming. https://arxiv.org/abs/1810.00873

[4] Bradley P Carlin and Thomas A Louis. 2008. *Bayesian methods for data analysis.* CRC Press.

[5] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software* 76, 1 (2017), 1–37.

[6] Jonathan P. Chen, Rohit Singh, Eli Bingham, and Noah Goodman. 2018. Transpiling Stan models to Pyro. In *The International Conference on Probabilistic Programming.*

[7] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-Purpose Probabilistic Programming System with Programmable Inference. In *Conference on Programming Language Design and Implementation (PLDI).* 221–236. https://doi.org/10.1145/3314221.3314642

[8] Luc De Raedt and Kristian Kersting. 2008. Probabilistic inductive logic programming. In *Probabilistic Inductive Logic Programming.* Springer, 1–27.

[9] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery.. In *IJCAI*, Vol. 7. Hyderabad, 2462–2467.

[10] Andrew Gelman and Jennifer Hill. 2006. *Data analysis using regression and multilevel/hierarchical models.* Cambridge university press.

[11] Andrew Gelman, Hal S Stern, John B Carlin, David B Dunson, Aki Vehtari, and Donald B Rubin. 2013. *Bayesian data analysis.* Chapman and Hall/CRC.

[12] Noah Goodman, Mansinghka Vikash, Daniel Roy, Keith Bonawitz, and Joshua Tenenbaum. 2008. Church: a Language for Generative Models. In *Conference on Uncertainty in Artificial Intelligence (UAI).* 220–229.

[13] Noah D. Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org Accessed February 2019.

[14] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *ICSE track on Future of Software Engineering (FOSE).* 167–181.

[15] Maria I. Gorinova, Andrew D. Gordon, and Charles A. Sutton. 2019. Probabilistic programming with densities in SlicStan: efficient, flexible, and deterministic. *PACMPL* 3, POPL (2019), 35:1–35:30.

[16] Matthew D. Homan and Andrew Gelman. 2014. The No-U-turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.* 15, 1 (Jan. 2014), 1593–1623. http://dl.acm.org/citation.cfm?id=2627435.2638586

[17] Diederik P. Kingma and Max Welling. 2013. Auto-Encoding Variational Bayes. https://arxiv.org/abs/1312.6114

[18] Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. System Sci.* 22, 3 (1981), 328–350.

[19] David Lunn, Chris Jackson, Nicky Best, Andrew Thomas, and David Spiegelhalter. 2012. *The BUGS Book: A Practical Introduction to Bayesian Analysis.* Chapman and Hall/CRC.

[20] David Lunn, David Spiegelhalter, Andrew Thomas, and Nicky Best. 2009. The BUGS project: Evolution, critique and future directions. *Statistics in medicine* 28, 25 (2009), 3049–3067.

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

[22] Andrew McCallum, Karl Schultz, and Sameer Singh. 2009. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems.* 1249–1257.

[23] Brian Milch, Bhaskara Marthi, Stuart J. Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. 2005. BLOG: Probabilistic Models with Unknown Objects. In *International Joint Conference on Artificial Intelligence (IJCAI).* 1352–1359.

[24] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (Dec. 1978), 348–375.

[25] Lawrence M. Murray and Thomas B. Schön. 2018. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control* 46 (2018), 29 – 43.

[26] Radford M. Neal. 2012. *Bayesian Learning for Neural Networks.* Vol. 118. Springer.

[27] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. In *AutoDiff Workshop.*

[28] Avi Pfeffer. 2001. IBAL: A Probabilistic Rational Programming Language. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 733–740.

[29] Avi Pfeffer. 2009. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report* 137 (2009), 96.

[30] Martyn Plummer et al. 2003. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, Vol. 124. Vienna, Austria.

[31] Danilo J. Rezende, Shakir Mohamed, and Daan Wierstra. 2014. Stochastic Backpropagation and Approximate Inference in Deep Generative Models. In *International Conference on Machine Learning (ICML)*. 1278–1286.

[32] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2015. Probabilistic Programming in Python Using PyMC3. https://arxiv.org/abs/1507.08050

[33] Jiaxin Shi, Jianfei. Chen, Jun Zhu, Shengyang Sun, Yucen Luo, Yihong Gu, and Yuhao Zhou. 2017. ZhuSuan: A Library for Bayesian Deep Learning. https://arxiv.org/abs/1709.05870

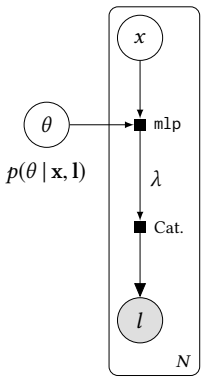
[34] N. Siddharth, Brooks Paige, Jan-Willem van de Meent, Alban Desmaison, Noah D. Goodman, Pushmeet Kohli, Frank Wood, and Philip Torr. 2017. Learning Disentangled Representations with Semi-Supervised Deep Generative Models. In *Conference on Neural Information Processing Systems (NIPS)*. 5927–5937.

[35] Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *European Symposium on Programming (ESOP)*. 855–879.

[36] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *Symposium on the Implementation and Application of Functional Programming Languages (IFL)*. 6:1–6:12.

[37] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *International Conference on Learning Representations (ICLR)*.

[38] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756* (2018).




```
networks { MLP mlp; }
data { int<lower=0, upper=1> img[28, 28];
       int<lower=0, upper=9> label; }
parameters { real mlp.l1.weight[_]; real mlp.l1.bias[_];
             real mlp.l2.weight[_]; real mlp.l2.bias[_]; }
model { real lambda[10];
        mlp.l1.weight ~ normal(0, 1);
        mlp.l1.bias   ~ normal(0, 1);
        mlp.l2.weight ~ normal(0, 1);
        mlp.l2.bias   ~ normal(0, 1);
        lambda = mlp(img);
        label ~ categorical_logits(lambda); }
guide parameters { real w1_mu[_]; real w1_sgma[_];
                   real b1_mu[_]; real b1_sgma[_];
                   real w2_mu[_]; real w2_sgma[_];
                   real b2_mu[_]; real b2_sgma[_]; }
guide { mlp.l1.weight ~ normal(w1_mu, exp(w1_sgma));
        mlp.l1.bias   ~ normal(b1_mu, exp(b1_sgma));
        mlp.l2.weight ~ normal(w2_mu, exp(w2_sgma));
        mlp.l2.bias   ~ normal(b2_mu, exp(b2_sgma)); }
```

Fig. 3. Graphical models and DeepStan code of the Bayesian MLP.

A COMPILING STAN'S BLOCKS

§2.3 has described how to translate Stan's `data`, `parameters`, and `model` blocks. Stan programs can have additional blocks for pre- and post-processing which we translate as follows. Functions in the `functions` block are translated to Python functions. We translate the `transformed data` block to a Python function that takes the data as arguments and returns the transformed data, and then introduce a new argument to the compiled model for receiving the transformed data. Stan separates the `transformed parameters` block from the `model` block to allow querying additional values after the inference. We merge the compiled code of the `transformed parameters` and `model` blocks and generate a new function that takes as arguments the data and a sample of the parameters and returns a sample of the transformed parameters. Finally, the `generated quantities` block is compiled into a function that takes as arguments the data, the transformed data, and a sample of the inferred parameters, and returns the generated quantities.

B STAN WITH EXPLICIT VARIATIONAL GUIDES AND NEURAL NETWORKS

B.1 Bayesian networks

DeepStan lets users lift parameters of a neural network to random variables to create a Bayesian neural network [26]. The left side of Figure 3 shows a simple classifier for handwritten digits based on a multi-layer perceptron (MLP) where all the parameters are lifted to random variables. Unlike the networks used in the VAE, the parameters (regrouped under the variable θ) are represented using a circle to indicate random variables. The inference starts from prior beliefs about the parameters and learns distributions that fit observed data. We can then generate samples of concrete weights and biases to obtain an ensemble of as many MLPs as we like. The ensemble can vote for predictions and can quantify agreement.

The right of Figure 3 shows the corresponding code in DeepStan. We let users declare lifted neural network parameters in Stan's `parameters` block just like any other random variables. Network parameters are identified by the name of the network and a path, e.g., `mlp.l1.weight`, following PyTorch naming conventions. The `model` block defines `normal(0,1)` priors for the weights and

biases of the two linear layers of the MLP. Then, for each image, the computed label follows a categorical distribution parameterized by the output of the network, which associates a probability to each of the ten possible values of the discrete random variable `label`. The `guide parameters` define μ and σ, and the `guide` block uses those parameters to propose normal distributions for the model parameters.

Compiling Bayesian neural networks. To lift neural networks we use Pyro `random_module`, a primitive that takes a PyTorch network and a dictionary of prior distributions and turns the network into a distribution of networks where each parameter is sampled from the corresponding prior distribution. We treat network parameters as any other random variables, that is, we apply the comprehensive translation from Section 2.3. This translation initializes parameters with a uniform prior and then compiles Stan ~ statements in the `model` block into Pyro observe statements.

```
priors = {'l1.weight': ImproperUniform(nh, nx),
          ...} # dict entries for remaining parameters
lifted_mlp = random_module("mlp", mlp, priors)()
params = dict(lift_mlp.named_parameters())
sample('l1.weight', Normal(zeros(nh, nx), ones(nh, nx)), obs=params['l1.weight'])
... # observe statements for remaining parameters
lambda_ = lifted_mlp(img)
sample("obs", Categorical(logits=lambda_), obs=label)
```

It is also possible to mix probabilistic parameters and non-probabilistic parameters. Our translation only lifts the parameters that are declared in the `parameters` block by only adding those to the `priors` dictionary.

Compiling the guide. We apply the compilation scheme described in Section 3.1. Variational parameters declared in the `guide parameters` block are compiled to learnable PyTorch parameters. Since they obey the restriction listed in Section 3.1, we can directly lift the network using the distribution defined in the `guide` block. Each ~ statement associated to a network parameter is added to the dictionary of priors used by `random_module`.

```
w1_mu = param("w1_mu", torch.randn((nh, nx)))
w1_sgma = param("w1_sgma", torch.randn((nh, nx)))
... # param declarations for remaining guide parameters
priors = {'l1.weight': Normal(w1_mu, exp(w1_sgma)),
          ...} # dict entries for remaining parameters
lifted_mlp = random_module("mlp", mlp, priors)()
```

B.2 Tensor dimension analysis

We designed the following type system for DeepStan:

$$
\begin{aligned}
\mathit{type} ::= {} & \texttt{real} \mid \texttt{int} \mid \mathit{type}[\mathit{dim}] \mid \texttt{vector}[\mathit{dim}] \mid \texttt{matrix}[\mathit{dim}, \mathit{dim}] \\
& \mid \mathit{typeVar} \mid \mathit{type}[?\mathit{dim}]\ \mathit{type}[^*] \mid \texttt{tensor}[\mathit{tensorExpr}, \mathit{intLit}] \\
\mathit{dim} ::= {} & \mathit{intLit} \mid \mathit{intExpr} \mid \mathit{dimVar} \mid \texttt{dim}[\mathit{tensorExpr}, \mathit{intLit}]
\end{aligned}
$$

There are two kinds of types: original Stan types vs. new generic types we introduced for our analysis. Original Stan types include `real`, `int`, arrays $\mathit{type}[\mathit{dim}]$, `vector`, and `matrix`. New generic types express various degrees of unknown information, which can then be deduced by unifying such types with other more specific types. A *typeVar* means nothing is known about the type. A $\mathit{type}[?\mathit{dim}]$ can unify with both arrays and vectors. A $\mathit{type}[^*]$ has an unknown number of

Table 2. Comparison of execution time and parameter distributions between Stan and DeepStan.

Model	Coin	Double Normal	Linear Regression	Seeds	8 Schools	Aspirin
Stan (s)	30.82 + 0.13	27.87 + 0.09	29.60 + 0.34	30.38 + 1.67	31.31 + 0.52	29.13 + 0.58
DeepStan (s)	199.18	66.62	311.26	883.86	305.77	404.90
Max SKL	theta: 0.0037	theta: 0.006	sigma: 0.005	alpha0: 0.058	mu: 0.017	shrinkage[1]: 0.007

dimensions. And a `tensor[`*tensorExpr*`, `*intLit*`]` represents a tensor returned by a neural network via *tensorExpr*, then subscripted *intLit* number of times. Analogously, there are also more-or-less generic dimensions *dim*.

The following example illustrates how our analysis uses generic types and unification to figure out the type of variable `z` in the VAE in Figure 2. The `parameters` block declares `real z[_]`, which our analysis represents with the generic type `real[*]`. But code generation needs concrete dimensions for initializing `z` correctly. The analysis needs to derive those concrete dimensions elsewhere. The statement `z ~ normal(0, 1)` yields no new information about the dimensions because it is auto-vectorized. The declaration `real encoded[2,nz]` tells the analysis that variable `encoded` has type `real[2][nz]`, a nested array of dimensions *intLit* 2 and *intExpr* `nz`. The assignment `mu_z = encoded[1]` unifies the type of `mu_z` with `real[nz]`. Finally, the statement `z ~ normal(mu_z, sigma_z)` unifies the type of `z` with the type of `mu_z`, so now `z` has type `real[nz]`. The code generator then uses that deduced type to put the concrete dimension size `nz` into the generated Python:

```
z = sample(ImproperUniform(shape=nz))
sample(dist.Normal(zeros(nz), 1), obs=z)
```

C EXPERIMENTS

This section evaluates DeepStan on multiple examples. For a subset of examples, we run inference on the generated Pyro code using NUTS (No U-Turn Sampler) [16], an optimized HMC which is the preferred inference method for Stan, and compare the results with Stan. We show that, sometimes, using explicit VI gives comparable or even more accurate results. Finally, for deep probabilistic models, we compare the generated Pyro code against hand-written code and find comparable results.

C.1 Comparison of Stan and DeepStan

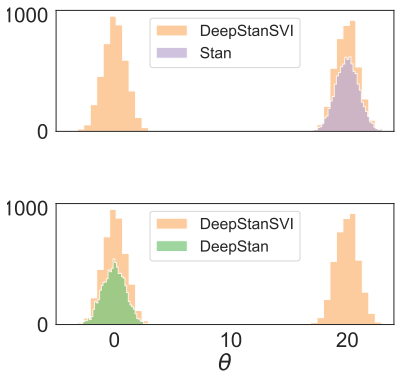
This section describes several experiments comparing inference output from Stan and DeepStan. The first two examples exercise our translation technique. *Coin* is the example presented Figure 1. *Double normal* is doing multiple updates of the same parameter `theta`.

The next four examples, *linear regression*, *seeds*, *8 schools*, and *aspirin* are taken from the Stan examples git repository.[1] *Seeds* is used for comparing SlicStan to Stan in [15], *8 schools* and *aspirin* are popular hierarchical examples.

The inference method is NUTS with 10,000 sampling steps, 300 warmup steps and 1 chain. For each parameter, we compare the posterior marginal distributions generated by Stan and DeepStan using a symmetrized version of the Kullback–Leibler (KL) divergence: $\mathrm{SKL}(P, Q) = \mathrm{KL}(P, Q) + \mathrm{KL}(Q, P)$, where P and Q are the two probability distributions being compared.

Table 2 summarizes the results of our experiments. The first two lines report the execution time for Stan and DeepStan averaged over 5 runs. For Stan we separate compilation and inference time.

[1] https://github.com/stan-dev/example-models




```
parameters {
  real cluster;
  real theta; }
model {
  real mu;
  cluster ~ normal(0, 1);
  if (cluster > 0) mu = 20;
  else mu = 0;
  theta ~ normal(mu, 1); }
guide parameters {
  real mc;
  real m1; real m2;
  real ls1; real ls2; }
guide {
  cluster ~ normal(mc, 1);
  if (cluster > 0) theta ~ normal(m1, exp(ls1));
  else theta ~ normal(m2, exp(ls2)); }
```

Fig. 4. DeepStan code and histograms of the multimodal example using Stan, DeepStan with NUTS, and DeepStan with VI.

Experiments were run on a MacBook Pro 6 cores i9 (2.9 GHz, 32 GB RAM). Stan first compiles the model to C++, which takes significant time, but the inference is impressively fast. In comparison, the compilation from DeepStan to Pyro is quasi-instantaneous, but the Pyro version of NUTS is slower. Remark that there is a new beta version of Pyro called NumPyro. We can expect pretty different performance results with this new version.

The last line compare the distribution inferred by Stan and DeepStan for the 32 parameters of the 6 models. Each entry is the parameter with the maximal SKL averaged over 5 runs. A SKL close to zero indicates that the distributions are very similar. We observe that in all cases the SKL approaches 0 when the number of samples increases. These results empirically validate that our translation from DeepStan to Pyro preserves the Stan semantics.

Explicit Variational Guide. The *multimodal* example shown in Figure 4 is a mixture of two Gaussians with different means but identical variance. The histograms in the left half of Figure 4 show that in both Stan and DeepStan, this example is particularly challenging for NUTS. Using multiple chains, NUTS finds the two modes, but the chains do not mix and the relative densities are incorrect. This is a known limitation of HMC. As shown in the code of Figure 4 we can provide a custom variational guide that will correctly infer the two modes (DeepStanSVI). Note, however, that this approach requires a-priori knowledge about the shape of the true posterior.

C.2 Deep probabilistic models

Since Stan lacks support for deep probabilistic models, we could not use it as a baseline. Instead, we compare the performance of the code generated by our compiler with hand-written Pyro code on the VAE described in §3.2 and a simple Bayesian neural network.

VAE. Variational autoencoders were not designed as a predictive model but as a generative model to reconstruct images. Evaluating the performance of a VAE is thus non-obvious. We use the following experimental setting. We trained two VAEs on the MNIST dataset using VI: one hand-written in Pyro, the other written in DeepStan. For each image in the test set, the trained VAEs compute a latent representation of dimension 5. We then cluster these representations using KMeans with 10 clusters. Then we measure the performance of a VAE with the pairwise F1 metric:

true positives are the number of images of the same digit that appear in the same cluster. For Pyro F1=0.41 (precision=0.43, recall=0.40), and for DeepStan F1=0.43 (precision=0.44, recall=0.42). These numbers shows that compiling DeepStan to Pyro does not impact the performance of such deep probabilistic models.

Bayesian MLP. We trained two versions of a 2-levels Bayesian multi-layer perceptron (MLP) where all the parameters are lifted to random variables (see §B.1): one hand-written in Pyro, the other written in DeepStan. We trained both models for 20 epochs on the training set. For each model we then generated 100 samples of concrete weights and biases to obtain an ensemble MLP. The log-likelihood of the test set is then computed for each MLP. We observe that the log-likelihood distribution is indistinguishable for the two models (SKL : 0.039) and the execution time is about the same. Again, these experiments show that compiling DeepStan models to Pyro has little impact on the model.

Additionally, we observe that changing the priors on the network parameters from `normal(0,1)` to `normal(0,10)` (see §B.1) increases the accuracy of the models from 0.92 to 0.96. This further validates our compilation scheme where priors on parameters are compiled to `observe` statements on deep probabilistic models.