Chair of Aerodynamics and Fluid Mechanics
Department of Mechanical Engineering
Technical University of Munich

Probabilistic Programming for Scientific Discovery

Lecture 2

Ludger Paehler
Lviv Data Science Summer School

July 29, 2020

Table of Contents

Outline

Approaches to Inference - the Inference Engines

- A typical probabilistic programming system consists of:
  - A domain-specific language (DSL), which enables the user to express his model using the language-specific primitives
  - Provides a library of inference algorithms, which enable inference on probabilistic models definable in the DSL.
  - Prevalent Monte-Carlo and variational inference approaches have their own specific sets of strength, it is hence important to understand the inference algorithms one utilizes

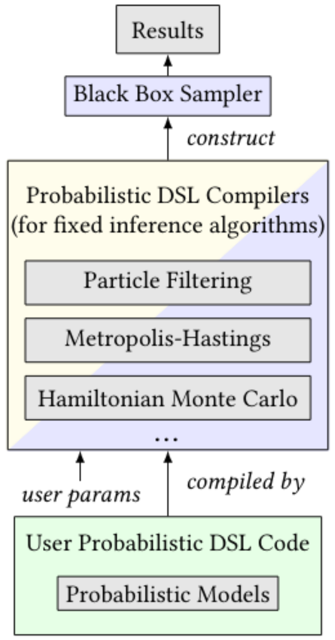


Figure: Structure of a typical probabilistic programming system. Source: *Gen: A General-Purpose Probabilistic Programming Systems with Programmable Inference*

Monte-Carlo Approaches to Inference

Random-Walk Metropolis Hastings [1] [2]

- Can be understood as a stochastic mode-finding algorithm looking
- Most simple Markov Chain Monte-Carlo algorithm
  - Default inference approach which will deliver a baseline solution
  - Constructs an ergodic and stationary Markov chain
- Often still used as a building block in modern algorithms
  - Simulate
  - Compute the acceptance probability
  - Sample from a uniform distribution to decide whether to accept or reject
- Original Markov Chain Monte-Carlo algorithm from which Langevin algorithms, and particle MCMC developed

[1] Gelman, A., Carlin, J.B., Stern, H.S., Dunson, D.B., Vehtari, A. and Rubin, D.B., 2013. Bayesian data analysis. CRC press.

[2] Gilks, W.R. and Richardson, S., S. and Spiegelhalter, D.(1996). Markov chain Monte Carlo in practice. London, UK: Chapman k Hall/CRC.

Monte-Carlo Approaches to Inference

Random-Walk Metropolis Hastings

- The Metropolis-Hasting acceptance ratio is given by

$$\alpha(y|x) = \min\left\{\frac{\pi(y)q(x|y)}{\pi(x)q(y|x)}, 1\right\}$$

- But simplifies under the random walk hypothesis to

$$= \min\left\{\frac{\pi(y)}{\pi(x)}, 1\right\}$$

- Which is then integrated with Monte-Carlo, Markov Chain Monte-Carlo or Gibbs Samplers, i.e.

$$\mathbb{E}_f[h(X)] = \int h(x)f(x)dx \longrightarrow \frac{1}{n}\sum_{i=1}^{n} h(x_i)$$

Monte-Carlo Approaches to Inference

Hamiltonian Monte-Carlo [3] [4]

- In high dimension random-walk Metropolis suffers from a vanishing acceptance probability, hence providing the impetus to exploit the information geometry of the gradient of the target probability density
- By endowing space with Hamiltonian structure we are then able to sample much more efficiently, where the gradient is channelled through the momentum part of the Hamiltonian
- One follows the Hamiltonian vector field to generate trajectories, which when projected back to the target parameter space generate efficient exploration trajectories
- Algorithm can be summarized as:
  1. Project onto the phase space
  2. Integrate Hamilton's equations to get projected exploration trajectories
  3. Project these trajectories back onto the target parameter space

[3] Neal, R.M., 2011. MCMC using Hamiltonian dynamics. Handbook of markov chain monte carlo, 2(11), p.2.
[4] Betancourt, M., 2017. A conceptual introduction to Hamiltonian Monte Carlo. arXiv preprint arXiv:1701.02434.

Monte-Carlo Approaches to Inference

Hamiltonian Monte-Carlo

- The Hamiltonian defines a joint distribution between position and momentum

$$\mathbb{P}(q,p) = \frac{1}{Z}\exp\left(-\frac{H(q,p)}{T}\right)$$

- Decomposing into momentum and position parts, we get a joint density

$$\mathbb{P}(q,p) = \frac{1}{Z}\exp\left(-\frac{U(q)}{T}\right)\exp\left(-\frac{K(p)}{T}\right)$$

- From which point on we can then express the posterior distribution as a canonical distribution using the potential energy function

$$U(q) = -\log\left[\pi(q)L(q|D)\right]$$

  with $\pi(q)$ being the prior density and $L(q|D)$ the likelihood function

Monte-Carlo Approaches to Inference

Hamiltonian Monte-Carlo

```
HMC = function (U, grad_U, epsilon, L, current_q)
{
  q = current_q
  p = rnorm(length(q),0,1)  # independent standard normal variates
  current_p = p

  # Make a half step for momentum at the beginning

  p = p - epsilon * grad_U(q) / 2

  # Alternate full steps for position and momentum

  for (i in 1:L)
  {
    # Make a full step for the position

    q = q + epsilon * p

    # Make a full step for the momentum, except at end of trajectory

    if (i!=L) p = p - epsilon * grad_U(q)
  }

  # Make a half step for momentum at the end.

  p = p - epsilon * grad_U(q) / 2

  # Negate momentum at end of trajectory to make the proposal symmetric

  p = -p

  # Evaluate potential and kinetic energies at start and end of trajectory

  current_U = U(current_q)
  current_K = sum(current_p^2) / 2
  proposed_U = U(q)
  proposed_K = sum(p^2) / 2

  # Accept or reject the state at end of trajectory, returning either
  # the position at the end of the trajectory or the initial position

  if (runif(1) < exp(current_U-proposed_U+current_K-proposed_K))
  {
    return (q)  # accept
  }
  else
  {
    return (current_q)  # reject
  }
}
```

Monte-Carlo Approaches to Inference

Hamiltonian Monte-Carlo

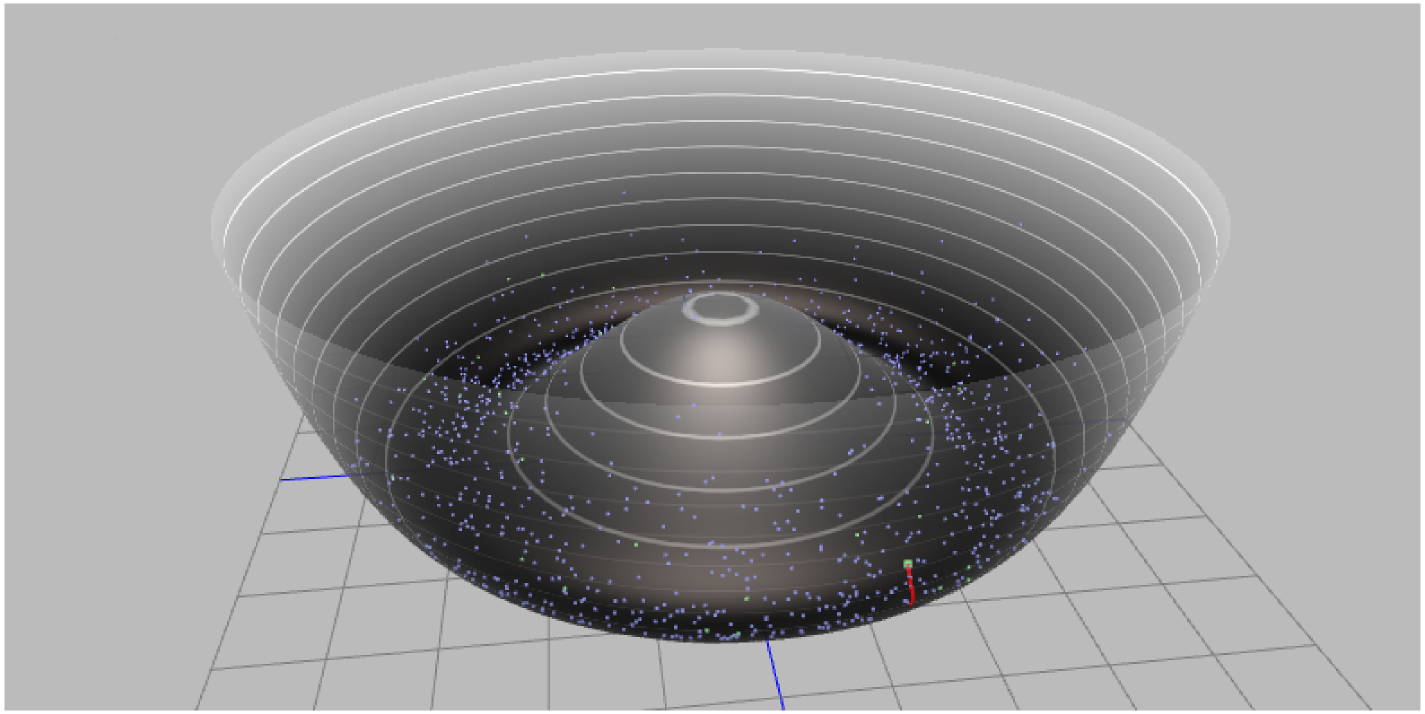
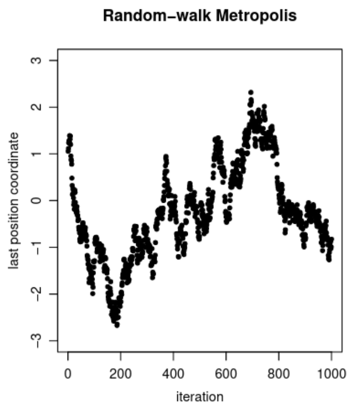


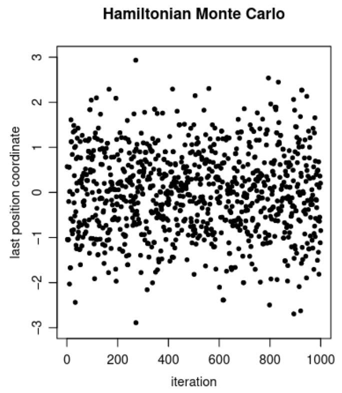
Figure: Information geometry of HMC. Source: *Hamiltonian Monte Carlo explained* by Alex Rogozhnikov

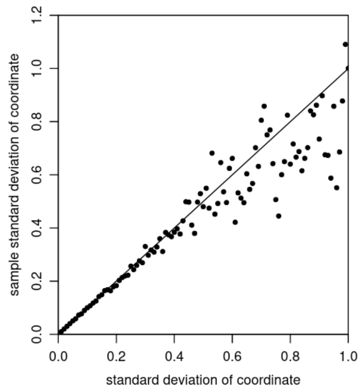
Chair of Aerodynamics and Fluid Mechanics
Department of Mechanical Engineering
Technical University of Munich

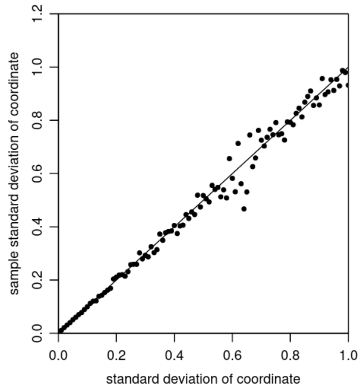
Monte-Carlo Approaches to Inference

Hamiltonian Monte-Carlo









Monte-Carlo Approaches to Inference

Stochastic-Gradient Langevin Dynamics [5] [6]

- Combines the stochastic approximation algorithms of Robbins & Monroe (see stochastic variational inference) with Langevin dynamics
- Langevin dynamics injects noise into the parameter updates s.t. the trajectory of the parameters will converge to the full posterior
- Can be understood as a two-phase approach
  1. Stochastic optimization
  2. Simulate samples from the posterior using Langevin dynamics
- Advantage over e.g. Hamiltonian dynamics is that this approach does not require sweeps over the whole dataset
- Key component of the approach is to switch from the stochastic approximation to posterior sampling at exactly the right time

[5] Welling, M. and Teh, Y.W., 2011. Bayesian learning via stochastic gradient Langevin dynamics. In Proceedings of the 28th international conference on machine learning (ICML-11) (pp. 681-688).

[6] Brosse, N., Durmus, A. and Moulines, E., 2018. The promises and pitfalls of stochastic gradient Langevin dynamics. In Advances in Neural Information Processing Systems (pp. 8268-8278).

Monte-Carlo Approaches to Inference

Stochastic-Gradient Langevin Dynamics

- Using Robbins-Monroe stochastic gradients, add Gaussian noise, which is to be balanced with the used step size

$$\Delta\theta_t = \frac{\epsilon_t}{2}\left(\nabla \log p(\theta_t) + \frac{N}{n}\sum_{i=1}^{n}\nabla \log p(x_{ti}|\theta_t)\right) + \eta_t, \quad \eta_t \sim \mathcal{N}(0, \epsilon_t)$$

- The preconditioned stochastic gradient Langevin dynamics are given by

$$\Delta\theta_t = \frac{\epsilon_t}{2}M\left(g(\theta_t + h_t(\theta_t)\right) + \eta_t, \quad \eta_t \sim \mathcal{N}(0, \epsilon_t M)$$

- Using the empirical covariance one V then differentiates between the stochastic approximation and posterior sampling phase, if $\alpha \ll 1$ then we are at the posterior sampling stage

$$\frac{\epsilon_t N^2}{4n}\lambda_{\max}\left(M^{\frac{1}{2}}V_s M^{\frac{1}{2}}\right) = \alpha \ll 1$$

Monte-Carlo Approaches to Inference

Stochastic-Gradient Langevin Dynamics [7]

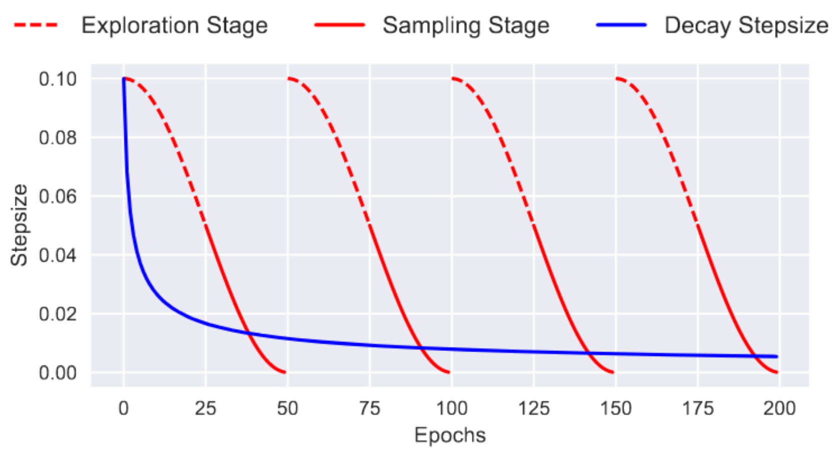
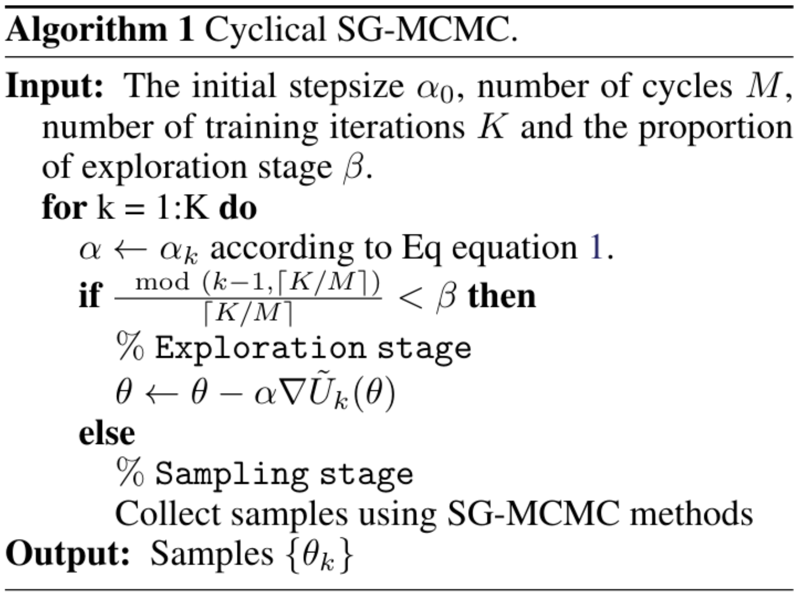


Figure: Cyclical step-size schedule to capture multimodal structures



Algorithm 1 Cyclical SG-MCMC.

```
Input: The initial stepsize $\alpha_0$, number of cycles $M$,
  number of training iterations $K$ and the proportion
  of exploration stage $\beta$.
  for k = 1:K do
    $\alpha \leftarrow \alpha_k$ according to Eq equation 1.
    if $\frac{\mathrm{mod}\ (k-1, \lceil K/M \rceil)}{\lceil K/M \rceil} < \beta$ then
      % Exploration stage
      $\theta \leftarrow \theta - \alpha \nabla \tilde{U}_k(\theta)$
    else
      % Sampling stage
      Collect samples using SG-MCMC methods
Output: Samples $\{\theta_k\}$
```

[7] Zhang, R., Li, C., Zhang, J., Chen, C. and Wilson, A.G., 2019. Cyclical stochastic gradient MCMC for Bayesian deep learning. arXiv preprint arXiv:1902.03932.

Monte-Carlo Approaches to Inference

Stochastic-Gradient Langevin Dynamics

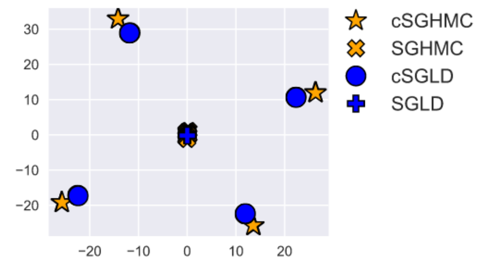


Figure: MDS on CIFAR-100

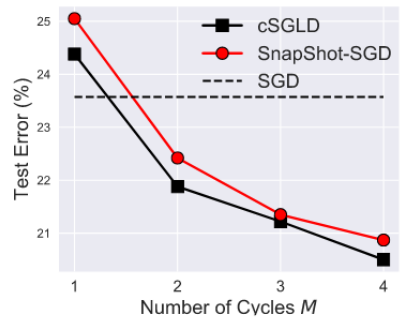


Figure: Comparison on CIFAR-100

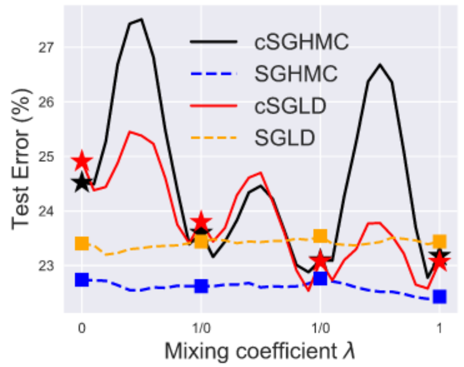


Figure: Interpolation on CIFAR-100

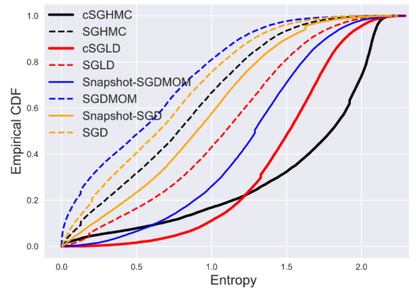


Figure: Empirical CDF on notMNIST

Variational Approaches to Inference

Variational Inference [8] [9] [10]

- Rephrases the problem from a Monte-Carlo sampling, which requires significant computation but converges to the true posterior to one, where we propose a family of distributions to then be optimized to be as close to the true posterior as possible using the Kullback-Leibler divergence
  - An approach to approximate densities, whereas MCMC is a tool to simulate from densities
  - Underestimates the variance of the posterior density
- Suited for large problems and problems of high complexity, where a Monte-Carlo inference would be computationally intractable
- Possible to make variational inference more efficient with a few markov chain monte carlo samples beforehand for a better initialization

[8] Blei, D.M., Kucukelbir, A. and McAuliffe, J.D., 2017. Variational inference: A review for statisticians. Journal of the American statistical Association, 112(518), pp.859-877.

[9] Zhang, C., Bütepage, J., Kjellström, H. and Mandt, S., 2018. Advances in variational inference. IEEE transactions on pattern analysis and machine intelligence, 41(8), pp.2008-2026.

[10] Salimans, T., Kingma, D. and Welling, M., 2015, June. Markov chain monte carlo and variational inference: Bridging the gap. In International Conference on Machine Learning (pp. 1218-1226).

Variational Approaches to Inference

Variational Inference

- Positing the family of approximate densities $\mathcal{D}$ and optimizing for the Kullback-Leibler divergence

$$q^{\star}(z) = \underset{q(z)\in\mathcal{D}}{\arg\min} \text{KL}(q(z)||p(z|x))$$

- The reach of the density family $\mathcal{D}$ governs the complexity of the optimization problem.
- The Kullback-Leibler divergence can be decomposed into

$$\text{KL}(q(z)||p(z|x)) = \mathbb{E}[\log q(z)] - \mathbb{E}[\log p(z|x)]$$

- The Kullback-Leibler divergence is intractable for complex distributions, we hence define a surrogate objective called the evidence lower-bound (ELBO). The ELBO is the negative KL divergence plus a constant

$$\text{ELBO}(q) = \mathbb{E}[\log p(z,x)] - \mathbb{E}[\log q(z)]$$

Variational Approaches to Inference

Variational Inference

- We hence have

$$\max \text{ ELBO} \sim \min \text{ KL}$$

- The exact relation between the KL divergence measure and the surrogate objective is then

$$\log p(x) = \text{KL}(q(z)||p(z|x)) + \text{ELBO}(q)$$

- The most commonly used families of distributions belong to the mean-field variational family of distributions
  - Beware, the mean-field family cannot capture correlations between marginal densities!

$$q(z) = \prod_{j=1}^{m} q_j(z_j)$$

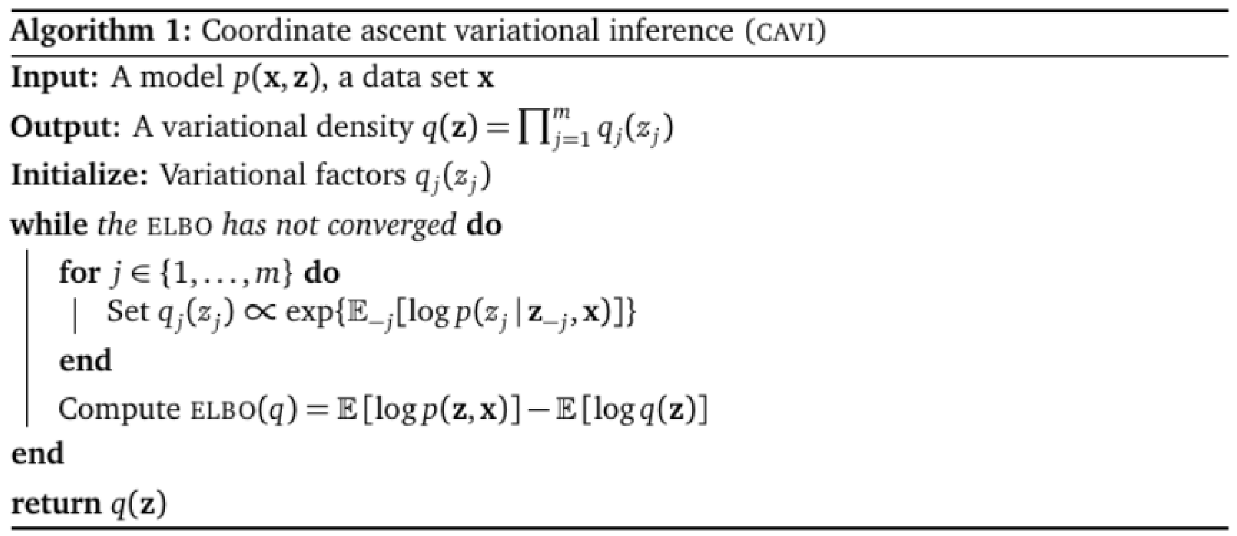
- There exists work though, which approximates more complex distributions [11] [12]

[11] Saul, L.K. and Jordan, M.I., 1996. Exploiting tractable substructures in intractable networks. In Advances in neural information processing systems (pp. 486-492).

[12] Barber, D. and Wiegerinck, W., 1999. Tractable variational structures for approximating graphical models. In Advances in Neural Information Processing Systems (pp. 183-189).

Variational Approaches to Inference

Variational Inference



Algorithm 1: Coordinate ascent variational inference (CAVI)

```
Input: A model $p(\mathbf{x}, \mathbf{z})$, a data set $\mathbf{x}$
Output: A variational density $q(\mathbf{z}) = \prod_{j=1}^{m} q_j(z_j)$
Initialize: Variational factors $q_j(z_j)$
while the ELBO has not converged do
    for $j \in \{1, \ldots, m\}$ do
        Set $q_j(z_j) \propto \exp\{\mathbb{E}_{-j}[\log p(z_j \mid \mathbf{z}_{-j}, \mathbf{x})]\}$
    end
    Compute ELBO$(q) = \mathbb{E}[\log p(\mathbf{z}, \mathbf{x})] - \mathbb{E}[\log q(\mathbf{z})]$
end
return $q(\mathbf{z})$
```

- But CAVI needs to iterate through the entire dataset for one iteration, hence motivating the use of stochastic approximations

Variational Approaches to Inference

Variational Inference

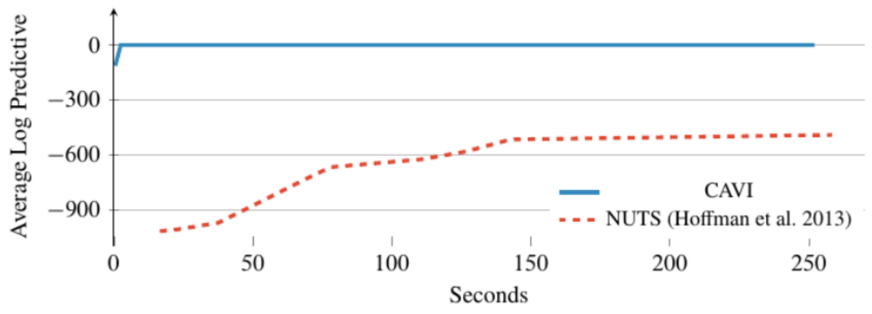


Figure: CAVI vs HMC using a NUTS sampler.

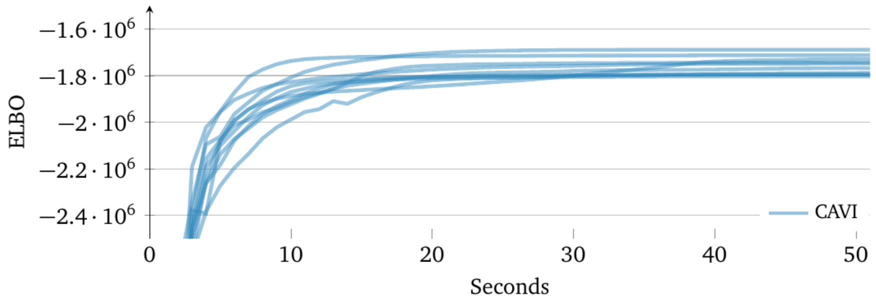


Figure: CAVI with different initializations.

Variational Approaches to Inference

Stochastic Variational Inference [13] [14]

- Stochastic variational inference (SVI) combines the natural gradients of Amari with the stochastic optimization of Robbins and Monroe
- The algorithm form of SVI can be sketched in the following way
  1. Subsample one or more data points from the data
  2. Analyze the subsample using the current variational parameters
  3. Implement a closed-form update of the variational parameters
  4. Repeat
- The used natural gradients alter the parameter space s.t. the same distance in different directions alters the symmetrized KL divergence by equal amounts
- Natural gradients are easier to compute than Euclidian gradients
- Suitable for extreme-scale applications, who don't need to fit in memory

[13] Hoffman, M.D., Blei, D.M., Wang, C. and Paisley, J., 2013. Stochastic variational inference. The Journal of Machine Learning Research, 14(1), pp.1303-1347.

[14] Robbins, H. and Monro, S., 1951. A stochastic approximation method. The annals of mathematical statistics, pp.400-407.

Variational Approaches to Inference

Stochastic Variational Inference

- Stochastic optimization then follows the cheaper natural gradients to find the maximum of the ELBO
- The sequence of step size for the optimization needs to obey the following condition meanwhile:

$$\sum_t \epsilon_t = \infty; \quad \sum_t \epsilon_t^2 < \infty$$

- Writing the gradient of the ELBO as an expectation one can then computer Monte-Carlo estimates of the gradient and hence compute a few gradients before starting the actual SVI algorithm
- Has to make a few constraining assumptions for closed-form optimization:
  - Each conditional is in an exponential family
  - The variational distribution has to be in the same exponential family
- Does not need to iterate over the complete dataset and only requires computation over local context for each iteration

Variational Approaches to Inference

Stochastic Variational Inference

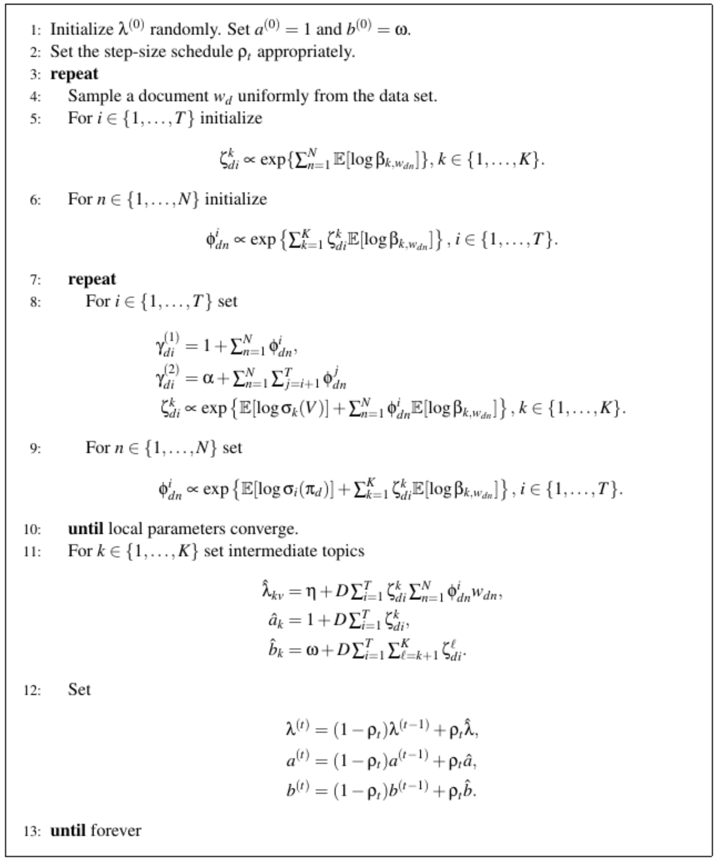
```
1: Initialize $\lambda^{(0)}$ randomly.
2: Set the step-size schedule $\rho_t$ appropriately.
3: **repeat**
4:     Sample a data point $x_i$ uniformly from the data set.
5:     Compute its local variational parameter,
           $$\phi = \mathbb{E}_{\lambda^{(t-1)}}[\eta_g(x_i^{(N)}, z_i^{(N)})].$$
6:     Compute intermediate global parameters as though $x_i$ is replicated $N$ times,
           $$\hat{\lambda} = \mathbb{E}_{\phi}[\eta_g(x_i^{(N)}, z_i^{(N)})].$$
7:     Update the current estimate of the global variational parameters,
           $$\lambda^{(t)} = (1-\rho_t)\lambda^{(t-1)} + \rho_t\hat{\lambda}.$$
8: **until** forever
```

Variational Approaches to Inference

Stochastic Variational Inference

More involved algorithm for a hierarchical dirichlet process topic model

- Draw an infinite number of topics, $\beta_k \sim \text{Dirichlet}(\eta)$ for $k \in 1, 2, 3, \ldots$
- Draw corpus breaking proportions, $\nu_k \sim \text{Beta}(1, \omega)$ for $k \in 1, 2, 3, \ldots$
- For each document d:
  - Draw document-level top indices
  - Draw document breaking proportions
  - For each word n:
    - ▷ Draw topic assignment
    - ▷ Draw word



```
Initialize $\lambda^{(0)}$ randomly. Set $a^{(0)} = 1$ and $b^{(0)} = \omega$.
Set the step-size schedule $\rho_t$ appropriately.
**repeat**
    Sample a document $w_d$ uniformly from the data set.
    For $i \in \{1, \ldots, T\}$ initialize
        $\zeta_{di}^k \propto \exp\{\sum_{n=1}^N \mathbb{E}[\log \beta_{k,w_{dn}}]\}, k \in \{1, \ldots, K\}.$
    For $n \in \{1, \ldots, N\}$ initialize
        $\phi_{dn}^i \propto \exp\left\{\sum_{k=1}^K \zeta_{di}^k \mathbb{E}[\log \beta_{k,w_{dn}}]\right\}, i \in \{1, \ldots, T\}.$
    **repeat**
        For $i \in \{1, \ldots, T\}$ set
            $\gamma_{di}^{(1)} = 1 + \sum_{n=1}^N \phi_{dn}^i,$
            $\gamma_{di}^{(2)} = \alpha + \sum_{n=1}^N \sum_{j=i+1}^T \phi_{dn}^j$
            $\zeta_{di}^k \propto \exp\left\{\mathbb{E}[\log \sigma_k(V)] + \sum_{n=1}^N \phi_{dn}^i \mathbb{E}[\log \beta_{k,w_{dn}}]\right\}, k \in \{1, \ldots, K\}.$
        For $n \in \{1, \ldots, N\}$ set
            $\phi_{dn}^i \propto \exp\left\{\mathbb{E}[\log \sigma_i(\pi_d)] + \sum_{k=1}^K \zeta_{di}^k \mathbb{E}[\log \beta_{k,w_{dn}}]\right\}, i \in \{1, \ldots, T\}.$
    **until** local parameters converge.
    For $k \in \{1, \ldots, K\}$ set intermediate topics
        $\hat{\lambda}_{kv} = \eta + D \sum_{i=1}^T \zeta_{di}^k \sum_{n=1}^N \phi_{dn}^i w_{dn},$
        $\hat{a}_k = 1 + D \sum_{i=1}^T \zeta_{di}^k,$
        $\hat{b}_k = \omega + D \sum_{i=1}^T \sum_{\ell=k+1}^K \zeta_{di}^\ell.$
    Set
        $\lambda^{(t)} = (1 - \rho_t)\lambda^{(t-1)} + \rho_t \hat{\lambda},$
        $a^{(t)} = (1 - \rho_t) a^{(t-1)} + \rho_t \hat{a},$
        $b^{(t)} = (1 - \rho_t) b^{(t-1)} + \rho_t \hat{b}.$
**until** forever
```

Variational Approaches to Inference

Stochastic Variational Inference

Recap:

- Core idea: Use stochastic optimization to optimize the variational objective, following the noisy estimates of the natural gradient
- Adaptive learning rates can further accelerate the stochastic inference routine
- Can be used to further scale up recent advances
- Can benefit from further advances in stochastic optimization

Variational Approaches to Inference

Black Box Variational Inference [15] [16]

- Expands on the idea of stochastic variational inference by using Monte-Carlo samples to estimate the gradients of the variational objective
  - Extending variational inference to a more general setting, where there exists no closed-form solution and computation of the expectation becomes computationally intractable
- Rough outline:
  1. Recast variational distribution in terms of a simple function f of the latent and observed variables
  2. Take the expectation w.r.t. this distribution and compute the gradients using Monte-Carlo estimates
  3. Use these noisy gradients in a stochastic optimization algorithm to optimize variational parameters
- Control of the variance of the Monte-Carlo estimator then becomes the main issue
  - Two approches to reduce the variance of the estimator are introduced

[15] Ranganath, R., Gerrish, S. and Blei, D., 2014, April. Black box variational inference. In Artificial Intelligence and Statistics (pp. 814-822).

[16] Chu, C., Minami, K. and Fukumizu, K., 2020. The equivalence between Stein variational gradient descent and black-box variational inference. arXiv preprint arXiv:2004.01822.

Variational Approaches to Inference

Black Box Variational Inference

- Seek an unbiased estimator of the gradient $\rightarrow$ rewrite gradient of ELBO as expectation w.r.t. the variational distribution

$$\nabla_\lambda \mathcal{L} = \mathbb{E}_q \left[\nabla_\lambda \log q(z|\lambda)(\log p(x,z) - \log q(z|\lambda)) \right]$$

- Using Monte-Carlo we can then compute noisy unbiased gradient with

$$\nabla_\lambda \mathcal{L} \approx \frac{1}{S} \sum_{s=1}^{S} \nabla_\lambda \log q(z_s|\lambda)(\log p(x,z_s) - \log q(z_s|\lambda))$$

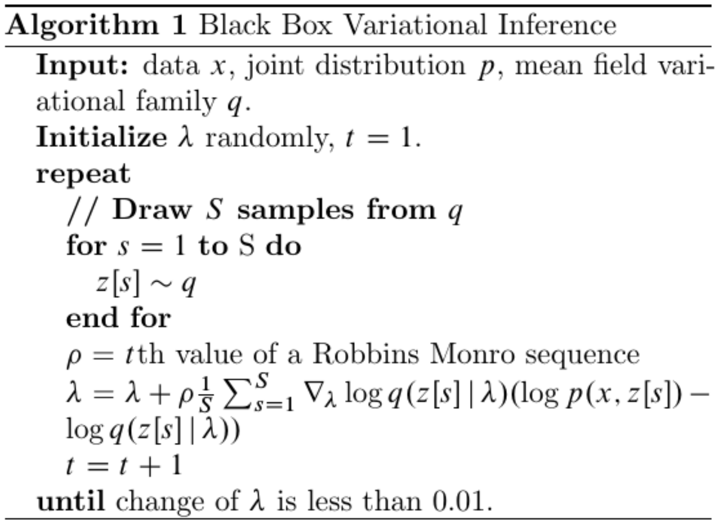
- While our estimator is unbiased we still have to reduce the variance of the estimator as efficiently as possible

  $\Rightarrow$ Rao-Blackwellization, i.e. replace a RV with its conditional expectation w.r.t. a subset of the Vs

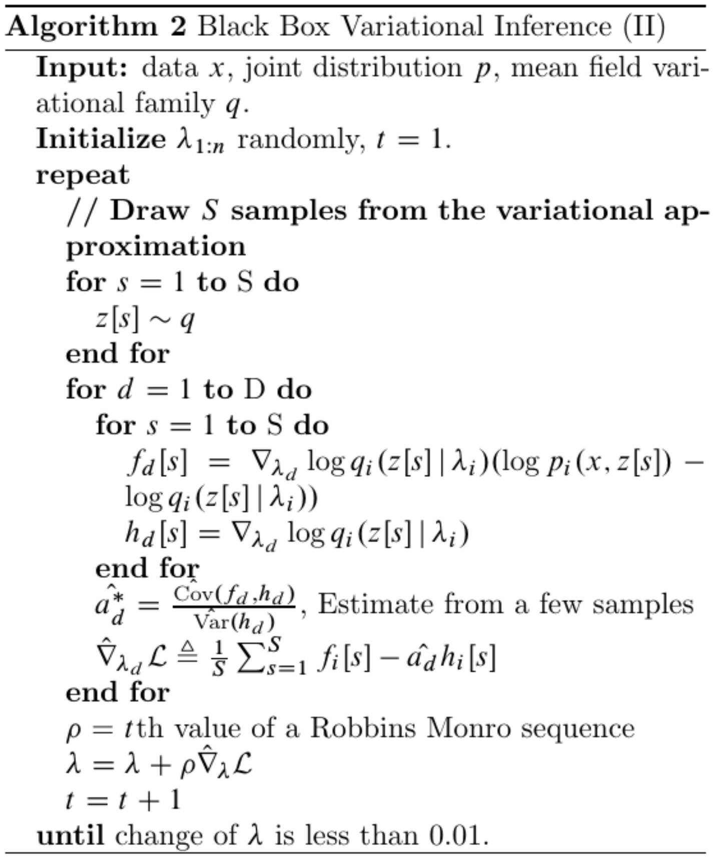
  $\Rightarrow$ Control-variates

Variational Approaches to Inference

Black Box Variational Inference



```
Algorithm 1 Black Box Variational Inference
  Input: data $x$, joint distribution $p$, mean field variational family $q$.
  Initialize $\lambda$ randomly, $t = 1$.
  repeat
    // Draw $S$ samples from $q$
    for $s = 1$ to S do
      $z[s] \sim q$
    end for
    $\rho$ = $t$th value of a Robbins Monro sequence
    $\lambda = \lambda + \rho \frac{1}{S} \sum_{s=1}^{S} \nabla_\lambda \log q(z[s] \mid \lambda)(\log p(x, z[s]) - \log q(z[s] \mid \lambda))$
    $t = t + 1$
  until change of $\lambda$ is less than 0.01.
```



```
Algorithm 2 Black Box Variational Inference (II)
  Input: data $x$, joint distribution $p$, mean field variational family $q$.
  Initialize $\lambda_{1:n}$ randomly, $t = 1$.
  repeat
    // Draw $S$ samples from the variational approximation
    for $s = 1$ to S do
      $z[s] \sim q$
    end for
    for $d = 1$ to D do
      for $s = 1$ to S do
        $f_d[s] = \nabla_{\lambda_d} \log q_i(z[s] \mid \lambda_i)(\log p_i(x, z[s]) - \log q_i(z[s] \mid \lambda_i))$
        $h_d[s] = \nabla_{\lambda_d} \log q_i(z[s] \mid \lambda_i)$
      end for
      $\hat{a}_d^* = \frac{\hat{\mathrm{Cov}}(f_d, h_d)}{\hat{\mathrm{Var}}(h_d)}$, Estimate from a few samples
      $\hat{\nabla}_{\lambda_d} \mathcal{L} \triangleq \frac{1}{S} \sum_{s=1}^{S} f_i[s] - \hat{a_d} h_i[s]$
    end for
    $\rho$ = $t$th value of a Robbins Monro sequence
    $\lambda = \lambda + \rho \hat{\nabla}_\lambda \mathcal{L}$
    $t = t + 1$
  until change of $\lambda$ is less than 0.01.
```

- Algorithm extends upon version 1 by using Rao-Blackwellization and control-variates

Variational Approaches to Inference

Black Box Variational Inference

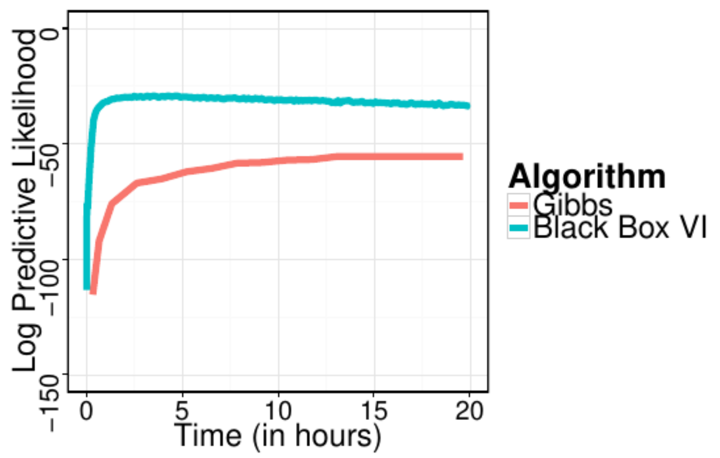


Figure: Comparison between Metropolis-Hastings within Gibbs and Black Box Variational Inference

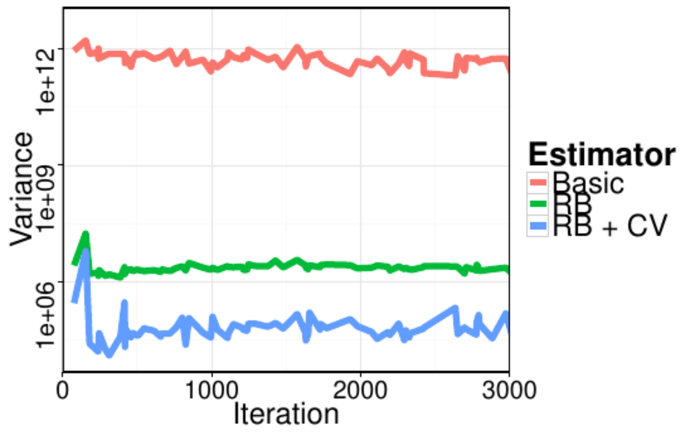


Figure: Variance comparison between the different algorithms

Variational Approaches to Inference

Automatic Differentiation Variational Inference [17] [18]

- Premise: Define your probabilistic model and your data, ADVI then performs inference in an automatic fashion
  - ADVI generates the variational algorithm for the defined model
- Steps of ADVI
  1. Transform the latent space s.t. all latent variables are defined on the same space
  2. Recast gradient of variational objective as expectation over q
  3. Reparameterize the gradient in terms of a Gaussian
  4. Use noisy gradients to optimize the variational distribution
- Key ingredients herein are automatic differentiation capabilities in the probabilistic programming system and a library of transformations for step 1.

[17] Kucukelbir, A., Tran, D., Ranganath, R., Gelman, A. and Blei, D.M., 2017. Automatic differentiation variational inference. The Journal of Machine Learning Research, 18(1), pp.430-474.

[18] Kucukelbir, A., Ranganath, R., Gelman, A. and Blei, D., 2015. Automatic variational inference in Stan. In Advances in neural information processing systems (pp. 568-576).

Variational Approaches to Inference

Automatic Differentiation Variational Inference

- Apply transformation s.t. the latent variables θ live in the real coordinate space $\mathbb{R}^k$

$$p(x,\zeta) = p(x, T^{-1}(\zeta)) |\det J_{T^{-1}}(\zeta)|$$

- The variational objective (ELBO) in the real coordinate space is then given by

$$\mathcal{L}(\phi) = \mathbb{E}_{q(\zeta;\phi)}\left[\log p(x, T^{-1}(\zeta)) + \log |\det J_{T^{-1}}(\zeta)|\right] + \mathbb{H}[q(\zeta;\phi)]$$

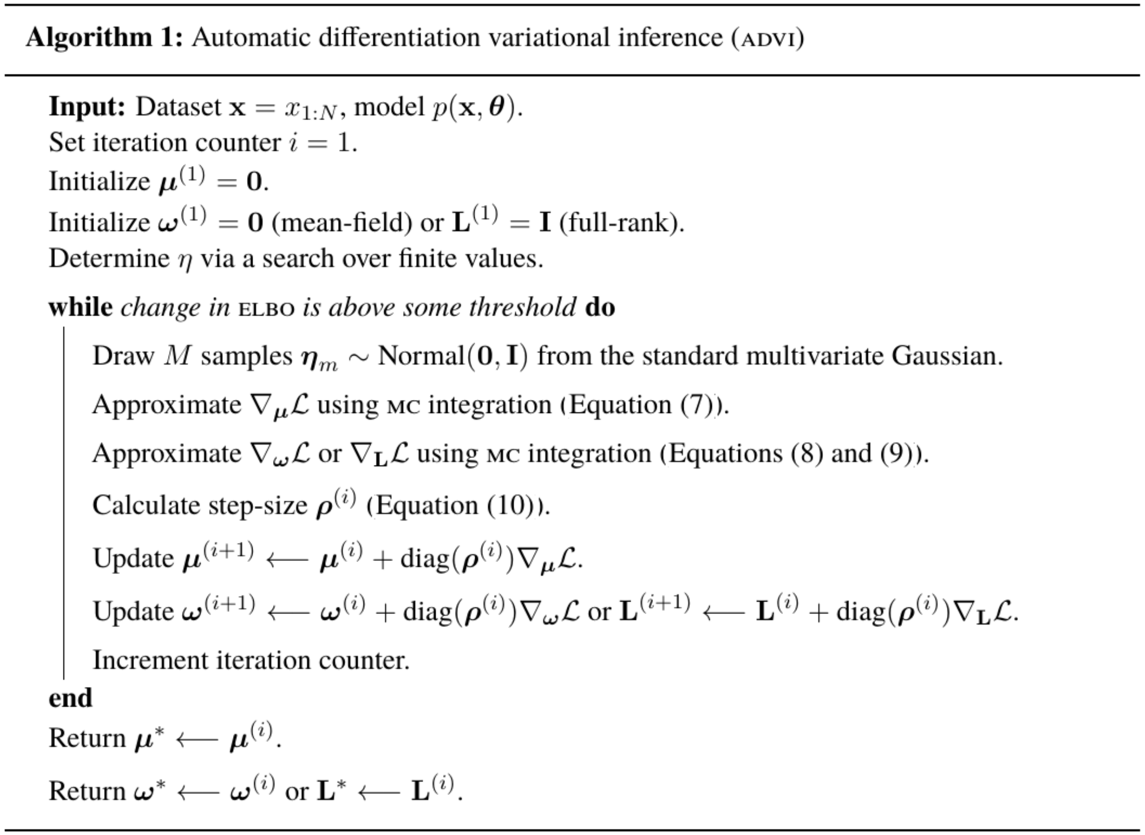
- Applying elliptical standardization to make the expectation tractable

$$\phi^\star = \arg\max_{\phi} \mathbb{E}_{N(\eta;0,I)}\left[\log p\left(x, T^{-1}(S_\phi^{-1}(\eta))\right) + \log |\det J_{T^{-1}}\left(S_\phi^{-1}(\eta)\right)|\right] + \mathbb{H}[q(\zeta;\phi)]$$

- Which can then be stochastically optimized

Variational Approaches to Inference

Automatic Differentiation Variational Inference



Algorithm 1: Automatic differentiation variational inference (ADVI)

```
Input: Dataset $\mathbf{x} = x_{1:N}$, model $p(\mathbf{x}, \boldsymbol{\theta})$.
Set iteration counter $i = 1$.
Initialize $\boldsymbol{\mu}^{(1)} = \mathbf{0}$.
Initialize $\boldsymbol{\omega}^{(1)} = \mathbf{0}$ (mean-field) or $\mathbf{L}^{(1)} = \mathbf{I}$ (full-rank).
Determine $\eta$ via a search over finite values.
while change in ELBO is above some threshold do
    Draw $M$ samples $\boldsymbol{\eta}_m \sim \text{Normal}(\mathbf{0}, \mathbf{I})$ from the standard multivariate Gaussian.
    Approximate $\nabla_{\boldsymbol{\mu}}\mathcal{L}$ using MC integration (Equation (7)).
    Approximate $\nabla_{\boldsymbol{\omega}}\mathcal{L}$ or $\nabla_{\mathbf{L}}\mathcal{L}$ using MC integration (Equations (8) and (9)).
    Calculate step-size $\boldsymbol{\rho}^{(i)}$ (Equation (10)).
    Update $\boldsymbol{\mu}^{(i+1)} \longleftarrow \boldsymbol{\mu}^{(i)} + \text{diag}(\boldsymbol{\rho}^{(i)})\nabla_{\boldsymbol{\mu}}\mathcal{L}$.
    Update $\boldsymbol{\omega}^{(i+1)} \longleftarrow \boldsymbol{\omega}^{(i)} + \text{diag}(\boldsymbol{\rho}^{(i)})\nabla_{\boldsymbol{\omega}}\mathcal{L}$ or $\mathbf{L}^{(i+1)} \longleftarrow \mathbf{L}^{(i)} + \text{diag}(\boldsymbol{\rho}^{(i)})\nabla_{\mathbf{L}}\mathcal{L}$.
    Increment iteration counter.
end
Return $\boldsymbol{\mu}^{*} \longleftarrow \boldsymbol{\mu}^{(i)}$.
Return $\boldsymbol{\omega}^{*} \longleftarrow \boldsymbol{\omega}^{(i)}$ or $\mathbf{L}^{*} \longleftarrow \mathbf{L}^{(i)}$.
```

Variational Approaches to Inference

Automatic Differentiation Variational Inference

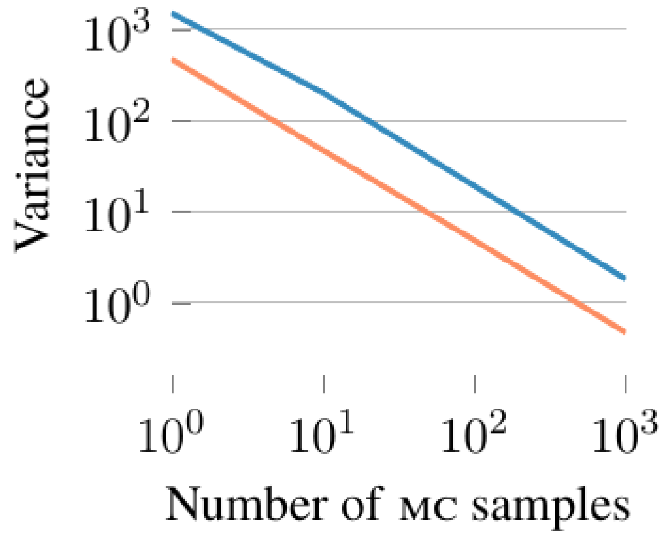


Figure: Univariate

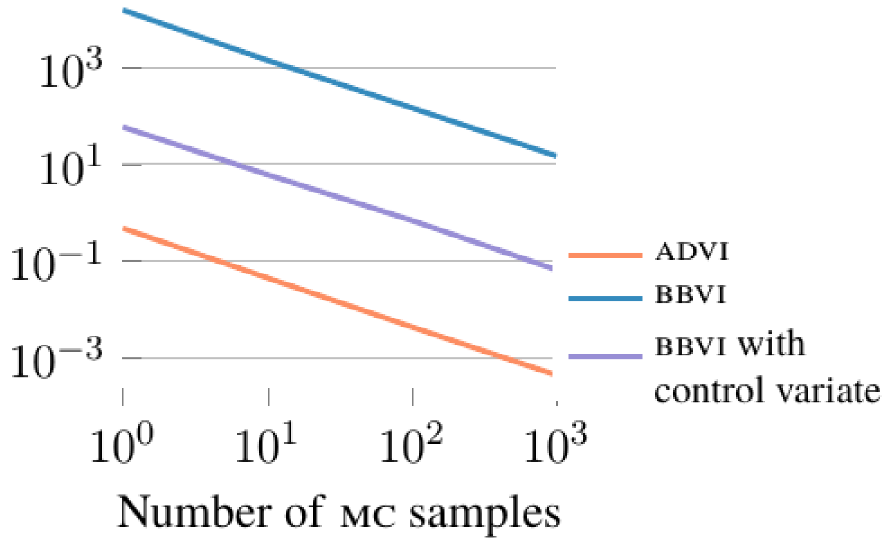


Figure: Multivariate

Comparison of gradient estimator variances

Variational Approaches to Inference

Automatic Differentiation Variational Inference

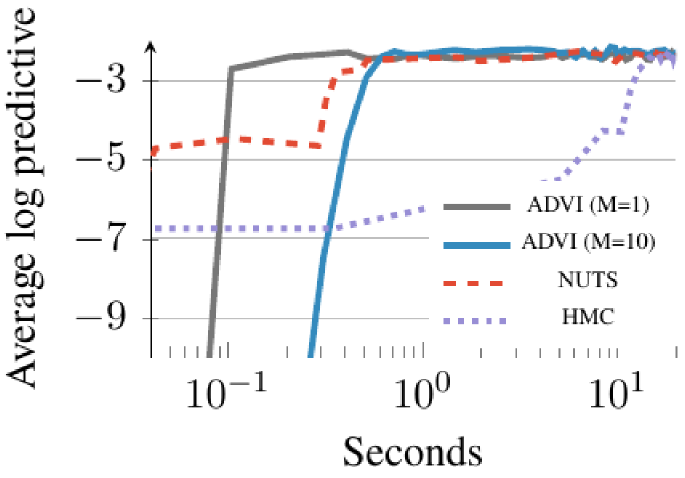


Figure: Linear regression

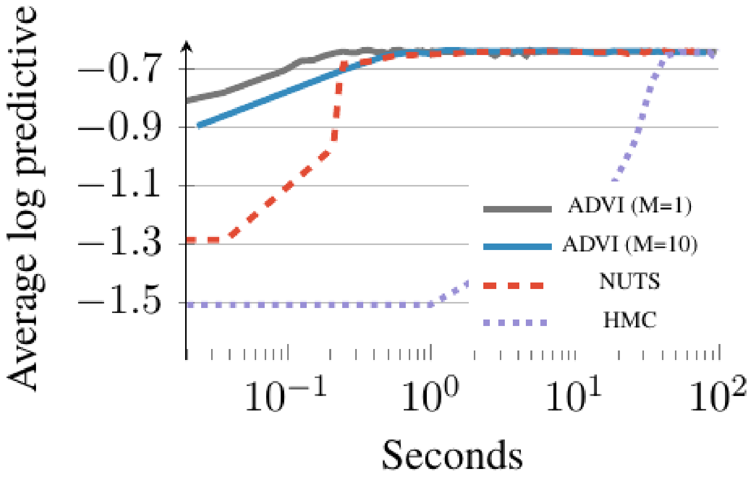


Figure: Hierarchical logistic regression

Held-out predictive accuracy results.

Outline

Stan [19]

Overview

- Stan is primarily aimed at statisticians and provides a full-fledged suite for them to express their statistical models and perform statistical inference
- Methods of inference provided:
  - Hamiltonian Monte-Carlo
  - no-U-turn sampler
  - Automatic Differentiation Variational Inference
- Defines its own separate DSL with interfaces for python, R, Matlab, Julia, State, Mathematica and the command-line
- Automatically differentiates the generative model using reverse-mode automatic differentiation
- Stan's core library in C++ with its interfaces to other languages makes it difficult to link to external simulators, the defined generatve model does furthermore get compiled, hence introducing a further layer of abstraction

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

Stan

Syntax [20]

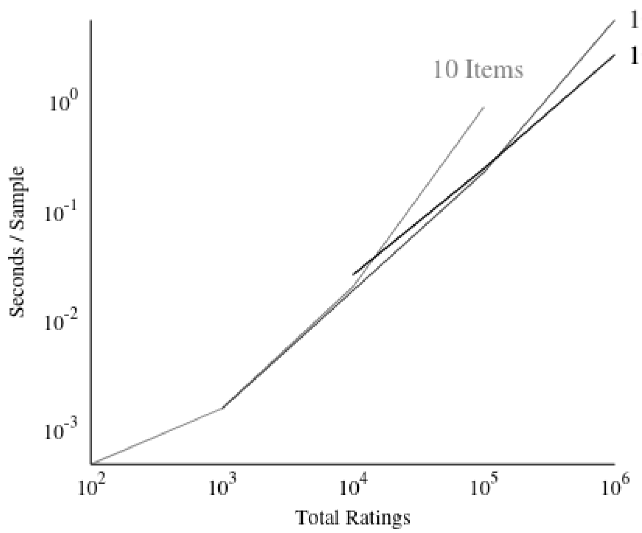
- Data block declares the data, which the program expects to receive
- Parameters block declared the unknown quantities, which are to be estimated
- Transformed parameters are functions of data and parameters
- Model block defines the computation of the log-posterior density

```
data {
  int N;
  vector[N] x;
  vector[N] y;
}
parameters {
  vector[2] log_a;
  ordered[2] log_b;
  real<lower=0> sigma;
}
transformed parameters {
  vector<lower=0>[2] a;
  vector<lower=0>[2] b;
  a <- exp(log_a);
  b <- exp(log_b);
}
model {
  vector[N] ypred;
  ypred <- a[1]*exp(-b[1]*x) + a[2]*exp(-b[2]*x);
  y ~ lognormal(log(ypred), sigma);
}
```

[20] Gelman, A., Lee, D. and Guo, J., 2015. Stan: A probabilistic programming language for Bayesian inference and optimization. Journal of Educational and Behavioral Statistics, 40(5), pp.530-543.

Stan

Application Performance



				Stan		JAGS	
# items	# raters	# groups	# data	time	memory	time	memory
20	2,000	100	40,000	:02m	16MB	:03m	220MB
40	8,000	200	320,000	:16m	92MB	:40m	1400MB
80	32,000	400	2,560,000	4h:10m	580MB	:??m	?MB

Venture [21] [22]

Overview

- Virtual machine for general-purpose probabilistic programming building on the ideas of Church, but enabling the user to specify custom inference strategies.
- Enables custom stochastic control flows through its stochastic procedure inferface
- Methods of inference provided include exact- and approximate inference:
  - Metropolis-Hastings
  - Hamiltonian Monte-Carlo
  - Gibbs sampling
  - Sequential Monte-Carlo
  - Variational inference
  - Inference programming is possible
- Evolved into a modern version, called *VentureScript*
- Able to link to external models, but not as easily as successor developments such as *Gen*

[21] Mansinghka, V., Selsam, D. and Perov, Y., 2014. Venture: a higher-order probabilistic programming platform with programmable inference. arXiv preprint arXiv:1404.0099.

[22] Goodman, N., Mansinghka, V., Roy, D.M., Bonawitz, K. and Tenenbaum, J.B., 2012. Church: a language for generative models. arXiv preprint arXiv:1206.3255.

Venture

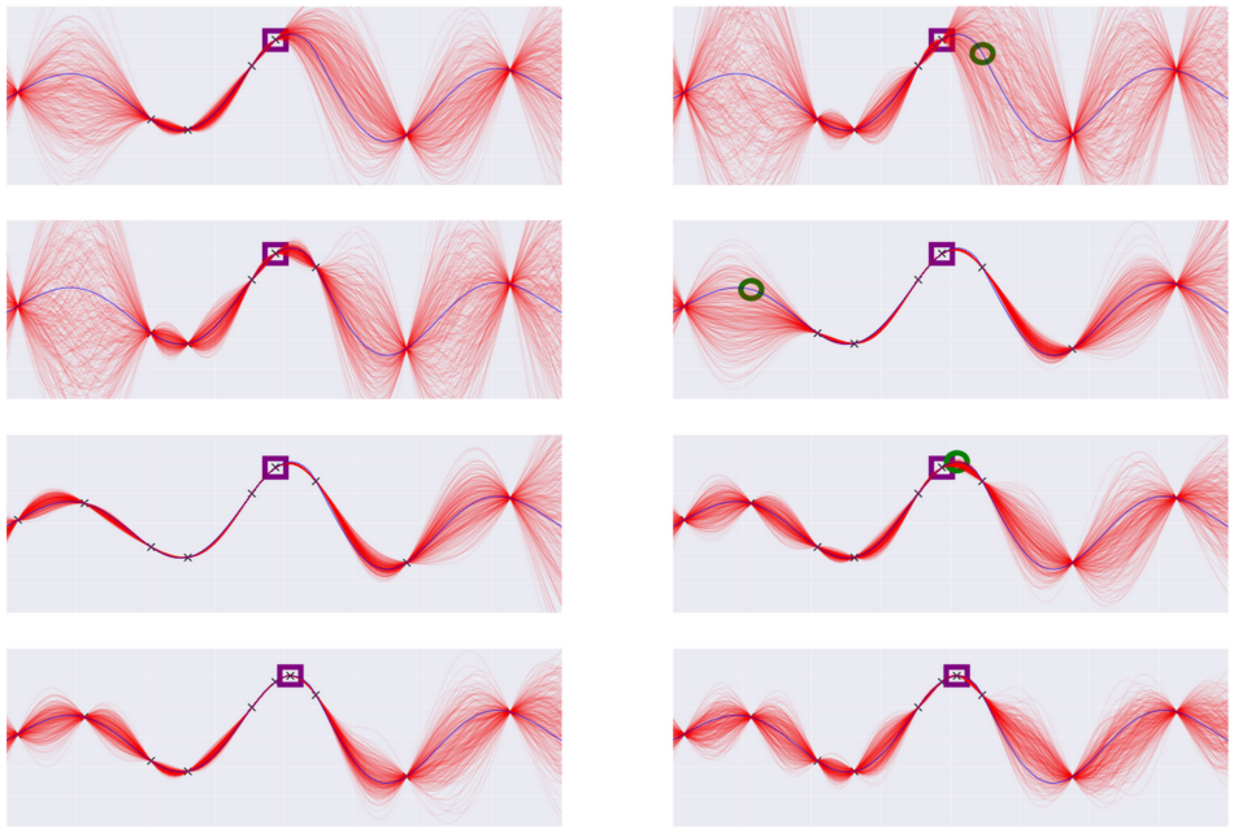
Syntax: Bayesian GP Optimization

- Repeatedly samples from the response surface created by the Gaussian process (GP) surrogate to reduce the number of function executions and discern the next point for function execution
- Then samples the function at said point to enrich our GP response surface
- Applies Metropolis-Hastings inference to the hyperparameters of the covariance function after each function execution

```
run(load_plugin("gpexample_plugin.py"));

// The target
assume V = make_audited_expensive_function("V");

// The GP memoizer
assume zero_func = make_const_func(0.0);
assume V_sf = tag(quote(hyper), 0, uniform_continuous(0, 10));
assume V_l = tag(quote(hyper), 1, uniform_continuous(0, 10));
assume V_se = make_squaredexp(V_sf, V_l);
assume V_package = gpmem(V, zero_func, V_se);
assume V_probe = first(V_package);
assume V_emu = second(V_package);

// A very naive estimate of the argmax of the given function
define mc_argmax = proc(func) {
  candidate_xs = mapv(proc(i) {uniform_continuous(-20, 20)},
                      arange(20));
  candidate_ys = mapv(func, candidate_xs);
  lookup(candidate_xs, argmax_of_array(candidate_ys))
};

// Shortcut to sample the emulator at a single point without packing
// and unpacking arrays
define V_emu_pointwise = proc(x) {
    run(sample lookup(V_emu(array(unquote(x))), 0))
};

// Main inference loop
infer repeat(15, do(pass,
    // Probe V at the point mc_argmax(V_emu_pointwise)
    predict(V_probe(unquote(mc_argmax(V_emu_pointwise)))),
    // Infer hyperparameters
    mh(quote(hyper), one, 50)));
```

Venture

Application Performance



PyMC3 [23]

Overview

- Very well-suited for the construction of graphical models, but is not Turing-complete as it is unable to model recursive distributions, and programs that can write programs
- Provided inference routines:
  - Hamiltonian Monte-Carlo
  - No-U-Turn Sampler
  - Sequential Monte-Carlo
  - Automatic Differentiation Variational Inference
  - Operator Variational Inference...
- Provides first-class support for the incorporation of Gaussian processes for the construction of Bayesian nonparametric models

[23] Salvatier, J., Wiecki, T.V. and Fonnesbeck, C., 2016. Probabilistic programming in Python using PyMC3. PeerJ Computer Science, 2, p.e55.

PyMC3

Syntax

- Constructing a toy neural network with 2 hidden layers of 5 neurons each
- Manual construction here, but PyMC3 is able to use Keras's API to Theano
- Mini-batches then accelerate convergence and allow the model to scale

```
def construct_nn(ann_input, ann_output):
    n_hidden = 5

    # Initialize random weights between each layer
    init_1 = np.random.randn(X.shape[1], n_hidden).astype(floatX)
    init_2 = np.random.randn(n_hidden, n_hidden).astype(floatX)
    init_out = np.random.randn(n_hidden).astype(floatX)

    with pm.Model() as neural_network:
        ann_input = pm.Data('ann_input', X_train)
        ann_output = pm.Data('ann_output', Y_train)
        weights_in_1 = pm.Normal('w_in_1', 0, sigma=1,
                                 shape=(X.shape[1], n_hidden),
                                 testval=init_1)
        weights_1_2 = pm.Normal('w_1_2', 0, sigma=1,
                                shape=(n_hidden, n_hidden),
                                testval=init_2)
        weights_2_out = pm.Normal('w_2_out', 0, sigma=1,
                                  shape=(n_hidden,),
                                  testval=init_out)
        act_1 = pm.math.tanh(pm.math.dot(ann_input,
                                         weights_in_1))
        act_2 = pm.math.tanh(pm.math.dot(act_1,
                                         weights_1_2))
        act_out = pm.math.sigmoid(pm.math.dot(act_2,
                                              weights_2_out))
        # Binary classification -> Bernoulli likelihood
        out = pm.Bernoulli('out',
                           act_out,
                           observed=ann_output,
                           total_size=Y_train.shape[0]
                          )
    return neural_network

neural_network = construct_nn(X_train, Y_train)

with neural_network:
    inference = pm.ADVI()
    approx = pm.fit(n=30000, method=inference)

minibatch_x = pm.Minibatch(X_train, batch_size=50)
minibatch_y = pm.Minibatch(Y_train, batch_size=50)
neural_network_minibatch = construct_nn(minibatch_x, minibatch_y)
with neural_network_minibatch:
    approx = pm.fit(40000, method=pm.ADVI())
```

PyMC3

Application Performance [24]

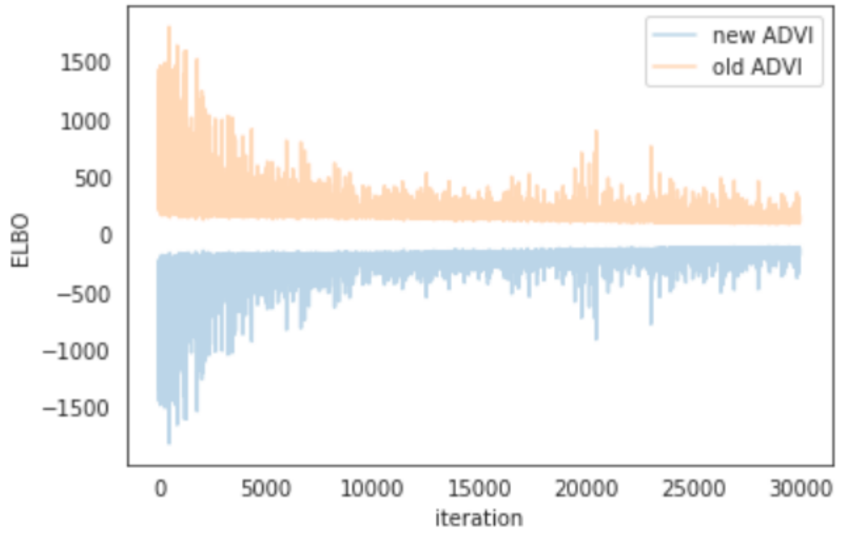


Figure: ADVI, 2073.86it/s

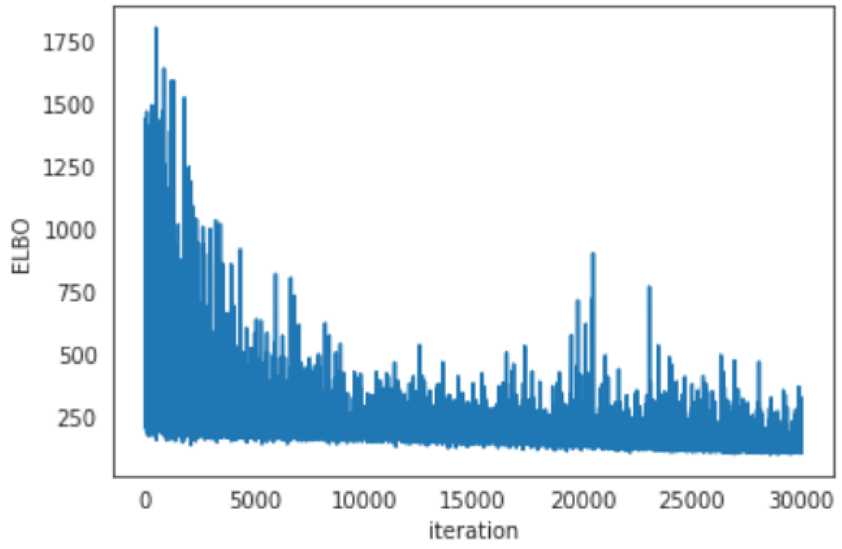


Figure: ADVI with mini-batch, 3586.77it/s

[24] Source: PyM3 Notebook on Variational Inference with Bayesian Neural Networks

TensorFlow Probability [25] [26]

Overview

- Probabilistic reasoning and statistical analysis library built on top of TensorFlow with a full integration with deep models defined in TensorFlow, automatic differentiation support and scalability on accelerators
- Provided inference routines:
  - Hamiltonian Monte-Carlo
  - Langevin Monte-Carlo
  - no-U-turn sampler
  - Variational inference
- Provides probably the most performant Markov Chain Monte-Carlo implementations to data, including multi-chain parallelism

[25] Dillon, J.V., Langmore, I., Tran, D., Brevdo, E., Vasudevan, S., Moore, D., Patton, B., Alemi, A., Hoffman, M. and Saurous, R.A., 2017. Tensorflow distributions. arXiv preprint arXiv:1711.10604.

[26] Lao, J., Suter, C., Langmore, I., Chimisov, C., Saxena, A., Sountsov, P., Moore, D., Saurous, R.A., Hoffman, M.D. and Dillon, J.V., 2020. tfp. mcmc: Modern Markov Chain Monte Carlo Tools Built for Modern Hardware. arXiv preprint arXiv:2002.01184.

TensorFlow Probability

Syntax

```
import convnet, pixelcnnpp

def make_encoder(x, z_size=8):
  net = convnet(x, z_size*2)
  return make_arflow(
    tfd.MultivariateNormalDiag(
      loc=net[..., :z_size],
      scale_diag=net[..., z_size:])),
    invert=True)

def make_decoder(z, x_shape=(28, 28, 1)):
  def _logit_func(features):
    # implement single autoregressive step,
    # combining observed features with
    # conditioning information in z.
    cond = tf.layers.dense(z,
      tf.reduce_prod(x_shape))
    cond = tf.reshape(cond, features.shape)
    logits = pixelcnnpp(
      tf.concat((features, cond), -1))
    return logits
  logit_template = tf.make_template(
    "pixelcnn++", _logit_func)
  make_dist = lambda x: tfd.Independent(
    tfd.Bernoulli(logit_template(x)))
  return tfd.Autoregressive(
    make_dist, tf.reduce_prod(x_shape))
```

```
def make_prior(z_size=8, dtype=tf.float32):
  return make_arflow(
    tfd.MultivariateNormalDiag(
      loc=tf.zeros([z_size], dtype)))

def make_arflow(z_dist, n_flows=4,
   hidden_size=(640,)*3, invert=False):
  maybe_invert = tfb.Invert if invert else tfb.
     Identity
  chain = list(itertools.chain.from_iterable([
    maybe_invert(tfb.MaskedAutoregressiveFlow(
      shift_and_log_scale_fn=tfb.\
      masked_autoregressive_default_template(
          hidden_size))),
    tfb.Permute(np.random.permutation(n_z)),
  ] for _ in range(n_flows)))
  return tfd.TransformedDistribution(
    distribution=z_dist,
    bijector=tfb.Chain(chain[:-1]))
```

Figure: SOTA with a PixelCNN++ decoder and autoregressive flows for encoder and prior.

TensorFlow Probability

Application Performance

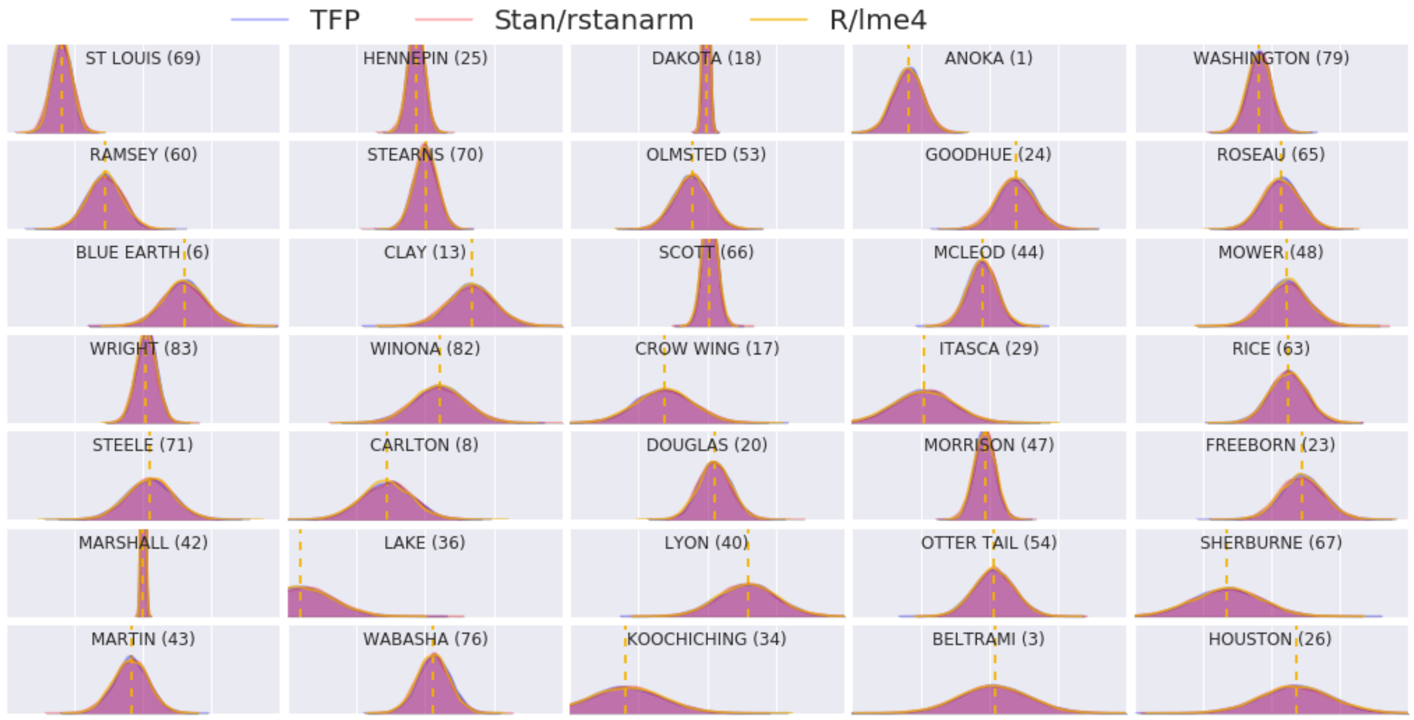


Figure: Linear Mixed-Effect Regression in TensorFlow Probability, R, and Stan [27]

[27] Source: TensorFlow Probability Tutorial

Pyro [29] & NumPyro [30]

Overview

- Pyro & NumPyro are both geared towards the definition of probabilistic programins in conjunction with state-of-the-art deep learning for large-data and high-dimensional models
- Methods of inference:
  - Stochastic Variational Inference
  - Importance Sampling
  - Sequential Monte-Carlo
  - Hamiltonian Monte-Carlo...
- Saw a further iteration in NumPyro, which uses JAX [28] as its backend to address accelerators
- Can link to simulator codes, but requires e.g. PPX bindings

[28] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M.J., Leary, C., Maclaurin, D. and Wanderman-Milne, S., 2020. JAX: composable transformations of Python+ NumPy programs, 2018. URL http://github. com/google/jax, p.18.

[29] Bingham, E., Chen, J.P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P. and Goodman, N.D., 2019. Pyro: Deep universal probabilistic programming. The Journal of Machine Learning Research, 20(1), pp.973-978.

[30] Phan, D., Pradhan, N. and Jankowiak, M., 2019. Composable effects for flexible and accelerated probabilistic programming in NumPyro. arXiv preprint arXiv:1912.11554.

Pyro

Syntax

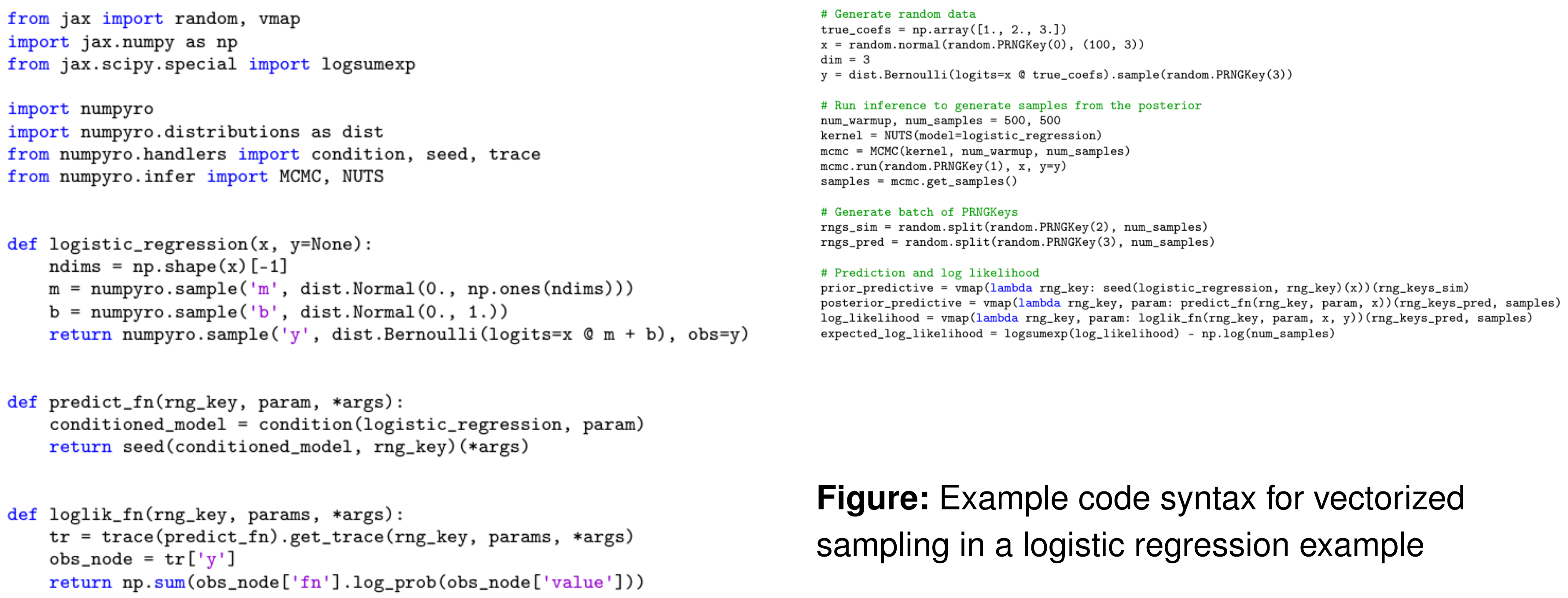
```
def model():
  loc, scale = torch.zeros(20), torch.ones(20)
  z = pyro.sample("z", Normal(loc, scale))
  w, b = pyro.param("weight"), pyro.param("bias")
  ps = torch.sigmoid(torch.mm(z, w) + b)
  return pyro.sample("x", Bernoulli(ps))


def guide(x):
  pyro.module("encoder", nn_encoder)
  loc, scale = nn_encoder(x)
  return pyro.sample("z", Normal(loc, scale))
```

```
def conditioned_model(x):
  return pyro.condition(model, data={"x": x})()

optimizer = pyro.optim.Adam({"lr": 0.001})
loss = pyro.infer.Trace_ELBO()

svi = pyro.infer.SVI(model=conditioned_model,
                     guide=guide,
                     optim=optimizer,
                     loss=loss)

losses = []
for batch in batches:
    losses.append(svi.step(batch))
```

Figure: Pyro example with generative model, approximate posterior, constraint specification, and stochastic variational inference

NumPyro

Syntax



```python
from jax import random, vmap
import jax.numpy as np
from jax.scipy.special import logsumexp

import numpyro
import numpyro.distributions as dist
from numpyro.handlers import condition, seed, trace
from numpyro.infer import MCMC, NUTS


def logistic_regression(x, y=None):
    ndims = np.shape(x)[-1]
    m = numpyro.sample('m', dist.Normal(0., np.ones(ndims)))
    b = numpyro.sample('b', dist.Normal(0., 1.))
    return numpyro.sample('y', dist.Bernoulli(logits=x @ m + b), obs=y)


def predict_fn(rng_key, param, *args):
    conditioned_model = condition(logistic_regression, param)
    return seed(conditioned_model, rng_key)(*args)


def loglik_fn(rng_key, params, *args):
    tr = trace(predict_fn).get_trace(rng_key, params, *args)
    obs_node = tr['y']
    return np.sum(obs_node['fn'].log_prob(obs_node['value']))
```

```python
# Generate random data
true_coefs = np.array([1., 2., 3.])
x = random.normal(random.PRNGKey(0), (100, 3))
dim = 3
y = dist.Bernoulli(logits=x @ true_coefs).sample(random.PRNGKey(3))

# Run inference to generate samples from the posterior
num_warmup, num_samples = 500, 500
kernel = NUTS(model=logistic_regression)
mcmc = MCMC(kernel, num_warmup, num_samples)
mcmc.run(random.PRNGKey(1), x, y=y)
samples = mcmc.get_samples()

# Generate batch of PRNGKeys
rngs_sim = random.split(random.PRNGKey(2), num_samples)
rngs_pred = random.split(random.PRNGKey(3), num_samples)

# Prediction and log likelihood
prior_predictive = vmap(lambda rng_key: seed(logistic_regression, rng_key)(x))(rng_keys_sim)
posterior_predictive = vmap(lambda rng_key, param: predict_fn(rng_key, param, x))(rng_keys_pred, samples)
log_likelihood = vmap(lambda rng_key, param: loglik_fn(rng_key, param, x, y))(rng_keys_pred, samples)
expected_log_likelihood = logsumexp(log_likelihood) - np.log(num_samples)
```

Figure: Example code syntax for vectorized sampling in a logistic regression example

Pyro & NumPyro

Application Performance

Framework	HMM[6]	COVTYPE
Stan (64-bit CPU)	0.53	135.94
Pyro (32-bit CPU)	30.51	32.76
Pyro (GPU)	-	3.36
NumPyro (32-bit CPU)	**0.09**	30.11
NumPyro (64-bit CPU)	0.15	71.18
NumPyro (GPU)	-	**1.46**

Figure: Time (ms) per leapfrog step in different frameworks

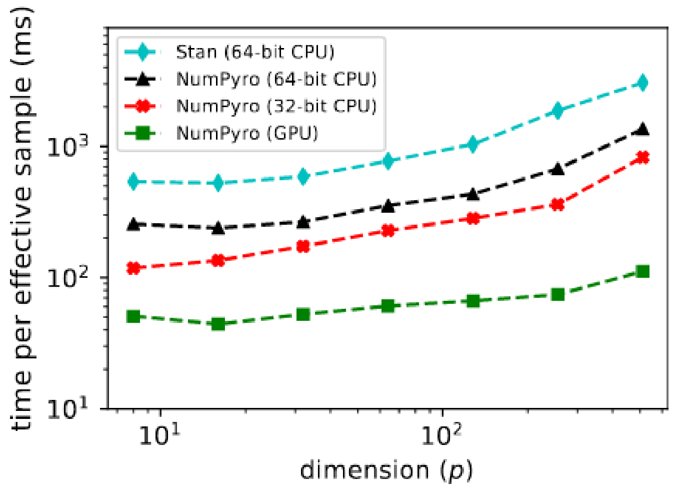


Figure: Time (ms) per effective sample for a sparse kernel interaction model as the dimensionality of the dataset (p) is varied.

Edward2 [31] [32]

Overview

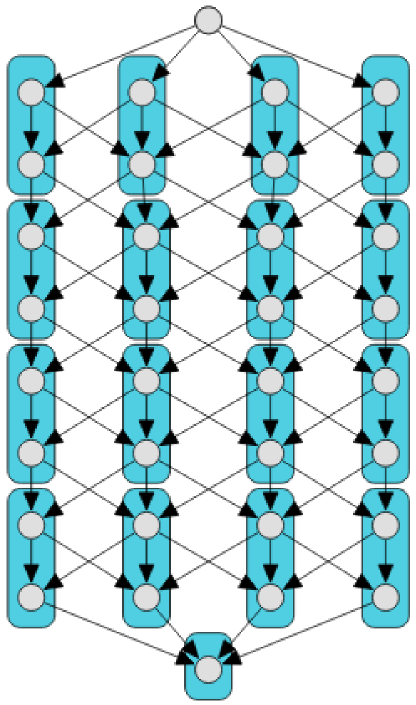
- A low-level approach to the embedding of probabilistic programming in the deep learning ecosystem, which hence runs directly on accelerator hardware and paves the way for larger scale models.
- Especially well-suited for the representation of uncertainty in neural network
- The same inference libraries as TensorFlow probability
- Especially well-suited for structured, hierarchical problems
- Highly efficient Monte-Carlo routines - use the same backend as TensorFlow probability
- Is able to utilize recent advances in XLA, such as sharding for large-scale models
- Removes many of the higher-level abstractions other languages benefit from

[31] Tran, D., Hoffman, M.W., Moore, D., Suter, C., Vasudevan, S. and Radul, A., 2018. Simple, distributed, and accelerated probabilistic programming. In Advances in Neural Information Processing Systems (pp. 7598-7609).

[32] Tran, D., Dusenberry, M., van der Wilk, M. and Hafner, D., 2019. Bayesian layers: A module for neural network uncertainty. In Advances in Neural Information Processing Systems (pp. 14660-14672).

Edward2

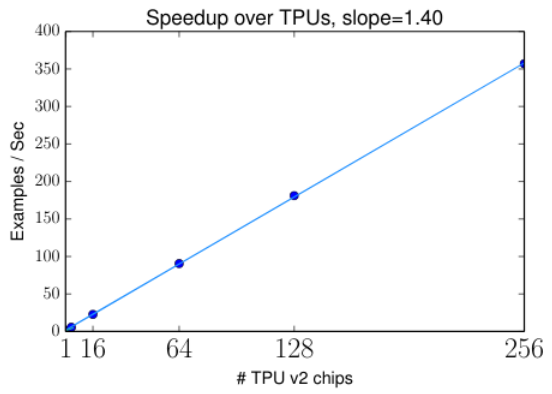
Syntax: Distributed Autoregressive Flow



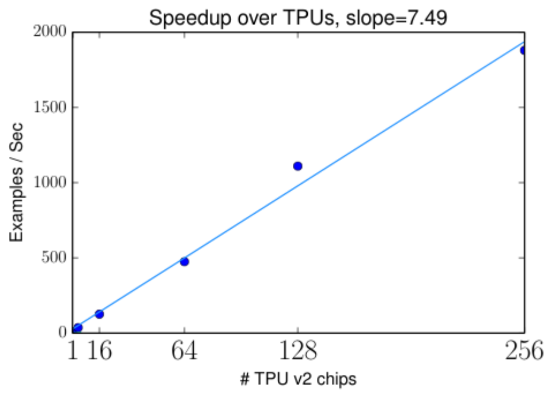
```
import SplitAutoregressiveFlow, masked_network
tfb = tf.contrib.distributions.bijectors

class DistributedAutoregressiveFlow(tfb.Bijector):
  def __init__(flow_size=[4]*8):
    self.flows = []
    for num_splits in flow_size:
      flow = SplitAutoregressiveFlow(masked_network, num_splits)
      self.flows.append(flow)
    self.flows.append(SplitAutoregressiveFlow(masked_network, 1))
    super(DistributedAutoregressiveFlow, self).__init__()

  def _forward(self, x):
    for l, flow in enumerate(self.flows):
      with tf.device(tf.contrib.tpu.core(l//2)):
        x = flow.forward(x)
    return x

  def _inverse_and_log_det_jacobian(self, y):
    ldj = 0.
    for l, flow in enumerate(self.flows[::-1]):
      with tf.device(tf.contrib.tpu.core(l//2)):
        y, new_ldj = flow.inverse_and_log_det_jacobian(y)
        ldj += new_ldj
    return y, ldj
```

Edward2

Application Performance



System	Runtime (ms)
Stan (CPU)	201.0
PyMC3 (CPU)	74.8
Handwritten TF (CPU)	66.2
Edward2 (CPU)	68.4
Handwritten TF (1 GPU)	**9.5**
Edward2 (1 GPU)	**9.7**
Edward2 (8 GPU)	**2.3**



(top-left): Vector-Quantized VAE on 64x64 ImageNet

(bottom-left): Image Transformer on 256x256 CelebA-HQ

(top-right): Time per leapfrog step for No-U-Turn Sample in Bayesian logistic regression

Gen [33] [34]

Overview

- Introduces multiple further abstractions, which differ from the other probabilistic programming frameworks as it relies on the abstraction of generative functions and a directly expandable infernece library
  - Especially geared towards computer vision and robotics
- Provides a dynamic, as well as a static DSL
- Provided inference routines:
  - Hamiltonian Monte-Carlo
  - Importance Sampling
  - Sequential Monte-Carlo
  - Black-Box Variational Inference...
- Able to link to simulators and amenable to metaprogramming

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

[34] Cusumano-Towner, M., Lew, A.K. and Mansinghka, V.K., 2020. Automating Involutive MCMC using Probabilistic and Differentiable Programming. arXiv preprint arXiv:2007.09871.

Gen

Programmable Inference

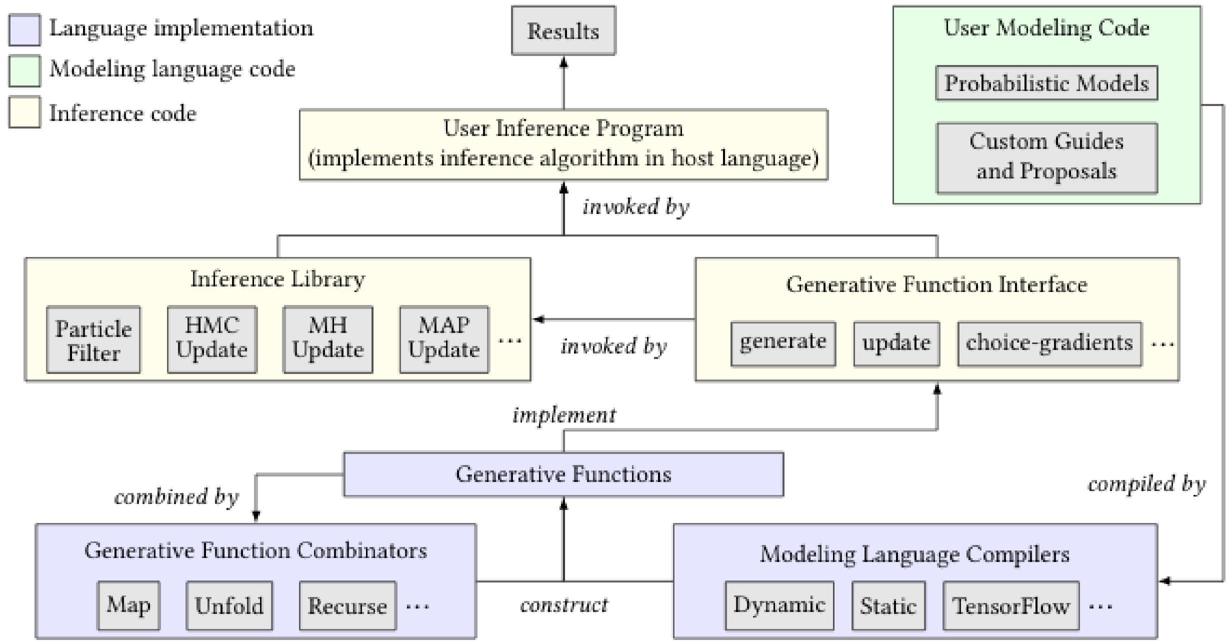


Figure: Gen's layout, which introduces further abstractions to go beyond current probabilistic programming systems.

Gen

Syntax: Body Pose Inference

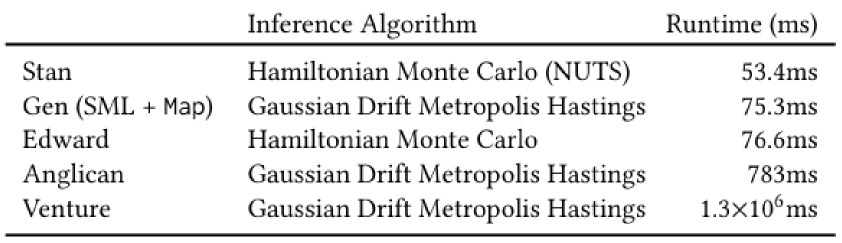
```
@gen function body_pose_prior()
  rot = @trace(uniform(0, 1), :rotation),
  elbow_r_x = @trace(uniform(0, 1), :elbow_right_x)
  elbow_r_y = @trace(uniform(0, 1), :elbow_right_y)
  elbow_r_z = @trace(uniform(0, 1), :elbow_right_z)
  ...
  pose = BodyPose(rot, elbow_r_x, elbow_r_y, elbow_r_z, ...)
  return pose
end

@gen function model()
  pose = @trace(body_pose_prior(), :pose)
  image = render_depth_image(pose)
  blurred = gaussian_blur(image)
  @trace(independent_pixel_noise(blurred, 0.1), :image)
end


tf = pyimport("tensorflow")
image_flat = tf.placeholder(tf.float64)
image = tf.reshape(image_flat, [-1, 128, 128, 1])
W_conv1 = tf.Variable(initial_weight([5, 5, 1, 32]))
b_conv1 = tf.Variable(initial_bias([32]))
h_conv1 = tf.nn.relu(tf.add(conv2d(image, W_conv1), b_conv1))
h_pool1 = max_pool_2x2(h_conv1)
..
W_fc2 = tf.Variable(initial_weight([1024, 32]))
b_fc2 = tf.Variable(initial_bias([32]))
output = tf.add(tf.matmul(h_fc1, W_fc2), b_fc2)
neural_net = TFFunction([W_conv1, b_conv1, ..], [image_flat], output)
```

```
@gen function predict_body_pose((grad)(nn_output::Vector{Float64}))
  @trace(beta(exp(nn_output[1]), exp(nn_output[2])), :rotation)
  @trace(beta(exp(nn_output[3]), exp(nn_output[4])), :elbow_right_x)
  @trace(beta(exp(nn_output[5]), exp(nn_output[6])), :elbow_right_y)
  @trace(beta(exp(nn_output[7]), exp(nn_output[8])), :elbow_right_z)
..
end

@gen function proposal(image::Matrix{Float64})
  nn_input = reshape(image, 1, 128 * 128)
  nn_output = @trace(neural_net(nn_input), :network)
  @trace(predict_body_pose(nn_output[1,:]), :pose)
end


function inference_program(image::Matrix{Float64})
  observations = choicemap()
  observations[:image] = image
  (trace,) = importance_resampling(model, (), observations,
                proposal, (image,), 10) # use 10 particles
  return trace
end
```

Figure: Code and evaluation for body pose inference.

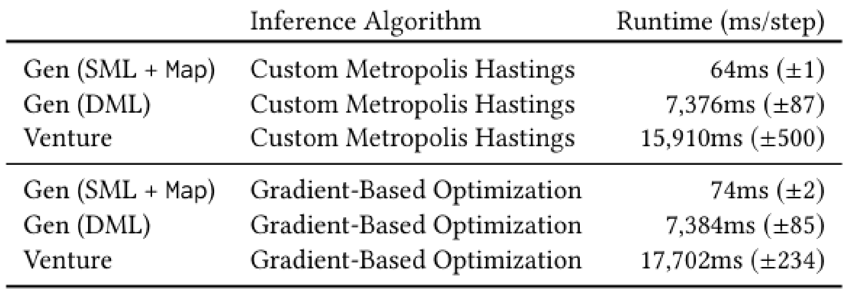
Gen

Application Performance



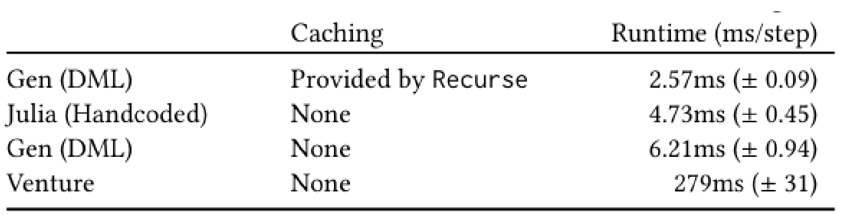
	Inference Algorithm	Runtime (ms)
Stan	Hamiltonian Monte Carlo (NUTS)	53.4ms
Gen (SML + Map)	Gaussian Drift Metropolis Hastings	75.3ms
Edward	Hamiltonian Monte Carlo	76.6ms
Anglican	Gaussian Drift Metropolis Hastings	783ms
Venture	Gaussian Drift Metropolis Hastings	1.3×10^6ms

Figure: Comparison of inference in collapsed model.



	Inference Algorithm	Runtime (ms/step)
Gen (SML + Map)	Custom Metropolis Hastings	64ms (±1)
Gen (DML)	Custom Metropolis Hastings	7,376ms (±87)
Venture	Custom Metropolis Hastings	15,910ms (±500)
Gen (SML + Map)	Gradient-Based Optimization	74ms (±2)
Gen (DML)	Gradient-Based Optimization	7,384ms (±85)
Venture	Gradient-Based Optimization	17,702ms (±234)

Figure: Comparison of inference in uncollapsed model. Including the visibly much fast static DSL of Gen in comparison with the dynamic DSL.



	Caching	Runtime (ms/step)
Gen (DML)	Provided by Recurse	2.57ms (± 0.09)
Julia (Handcoded)	None	4.73ms (± 0.45)
Gen (DML)	None	6.21ms (± 0.94)
Venture	None	279ms (± 31)

Figure: Comparison on gaussian process structure learning.

PyProb [35]

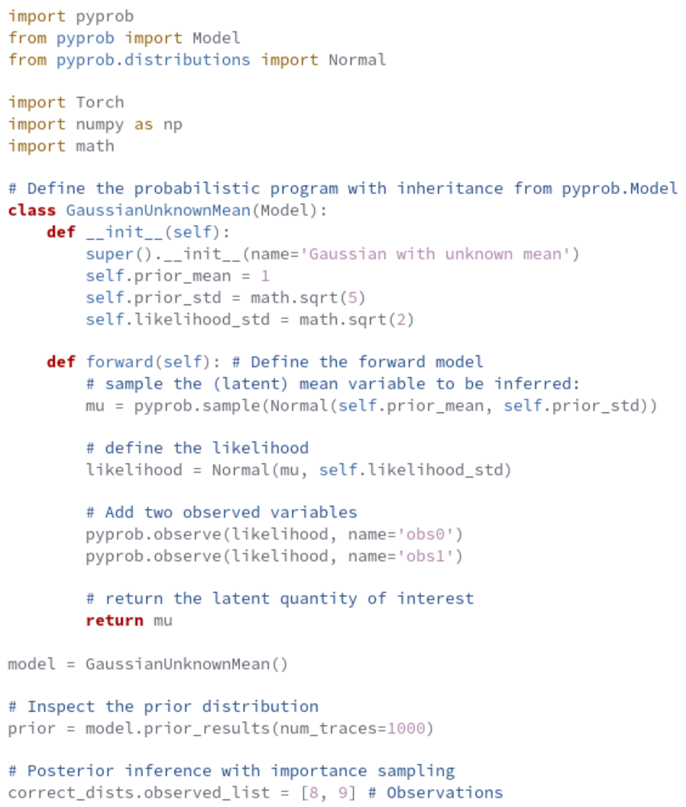
Overview

- Custom-made for concurrent workflows with HPC simulations through the PPX protocol based on flatbuffers, and tested on supercomputers
- Provided inference routines:
  - Markov Chain Monte-Carlo
  - Importance sampling
  - Importance sampling with inference compilation
- Made for distributed execution across large machines with distributed PyTorch providing the MPI-fuelled backend
- Intel-optimized PyTorch backend version
  - Possibly not the greatest performance on the heterogeneous machines of the future $\longrightarrow$ Is XLA possibly the safer bet for such a codebase?

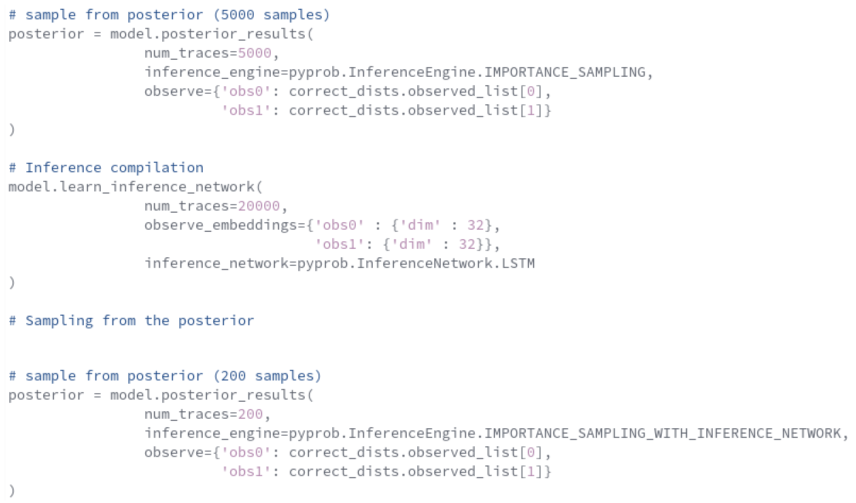
[35] Baydin, A.G., Shao, L., Bhimji, W., Heinrich, L., Naderiparizi, S., Munk, A., Liu, J., Gram-Hansen, B., Louppe, G., Meadows, L. and Torr, P., 2019. Efficient probabilistic inference in the quest for physics beyond the standard model. In Advances in neural information processing systems (pp. 5459-5472).

PyProb

Syntax



```python
import pyprob
from pyprob import Model
from pyprob.distributions import Normal

import Torch
import numpy as np
import math

# Define the probabilistic program with inheritance from pyprob.Model
class GaussianUnknownMean(Model):
    def __init__(self):
        super().__init__(name='Gaussian with unknown mean')
        self.prior_mean = 1
        self.prior_std = math.sqrt(5)
        self.likelihood_std = math.sqrt(2)

    def forward(self): # Define the forward model
        # sample the (latent) mean variable to be inferred:
        mu = pyprob.sample(Normal(self.prior_mean, self.prior_std))

        # define the likelihood
        likelihood = Normal(mu, self.likelihood_std)

        # Add two observed variables
        pyprob.observe(likelihood, name='obs0')
        pyprob.observe(likelihood, name='obs1')

        # return the latent quantity of interest
        return mu

model = GaussianUnknownMean()

# Inspect the prior distribution
prior = model.prior_results(num_traces=1000)

# Posterior inference with importance sampling
correct_dists.observed_list = [8, 9] # Observations
```

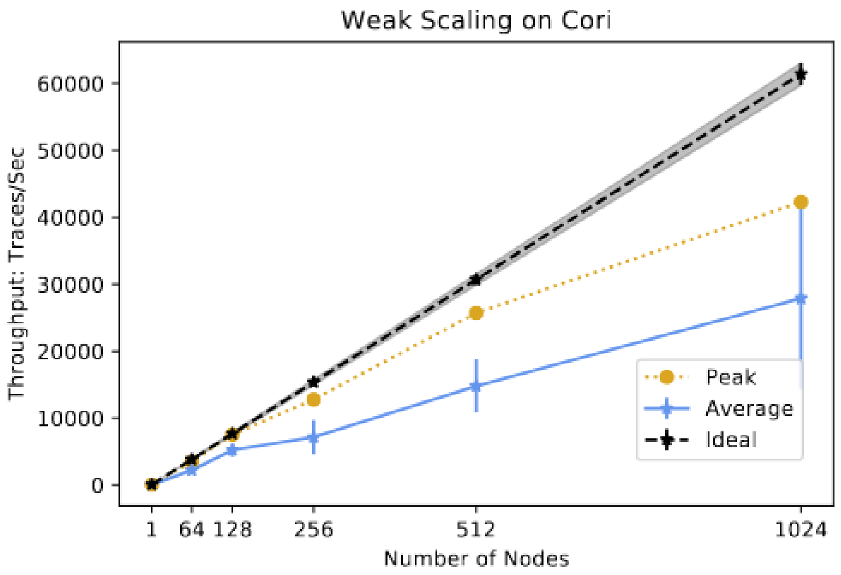


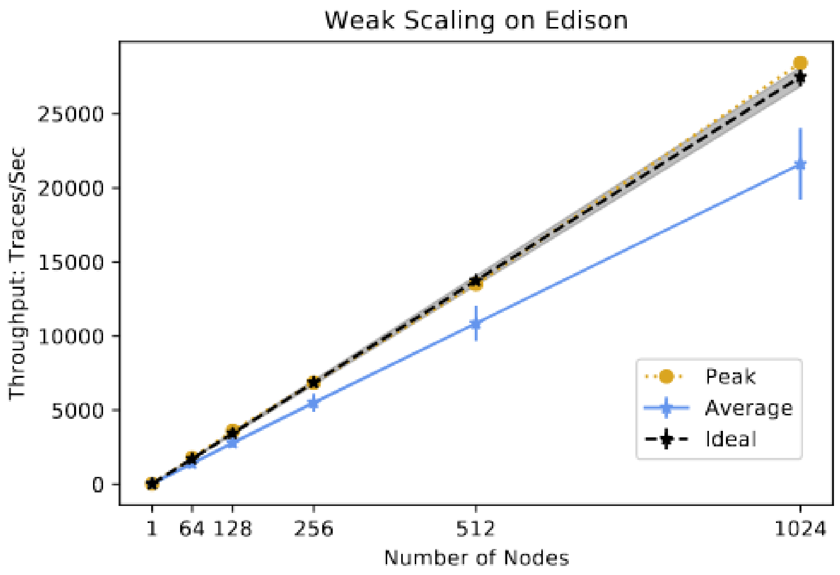
```python
# sample from posterior (5000 samples)
posterior = model.posterior_results(
                num_traces=5000,
                inference_engine=pyprob.InferenceEngine.IMPORTANCE_SAMPLING,
                observe={'obs0': correct_dists.observed_list[0],
                         'obs1': correct_dists.observed_list[1]}
)

# Inference compilation
model.learn_inference_network(
                num_traces=20000,
                observe_embeddings={'obs0' : {'dim' : 32},
                                    'obs1': {'dim' : 32}},
                inference_network=pyprob.InferenceNetwork.LSTM
)

# Sampling from the posterior


# sample from posterior (200 samples)
posterior = model.posterior_results(
                num_traces=200,
                inference_engine=pyprob.InferenceEngine.IMPORTANCE_SAMPLING_WITH_INFERENCE_NETWORK,
                observe={'obs0': correct_dists.observed_list[0],
                         'obs1': correct_dists.observed_list[1]}
)
```

Figure: Probabilistic inference on a Gaussian with unknown mean

PyProb

Application Performance





Turing [36]

Overview

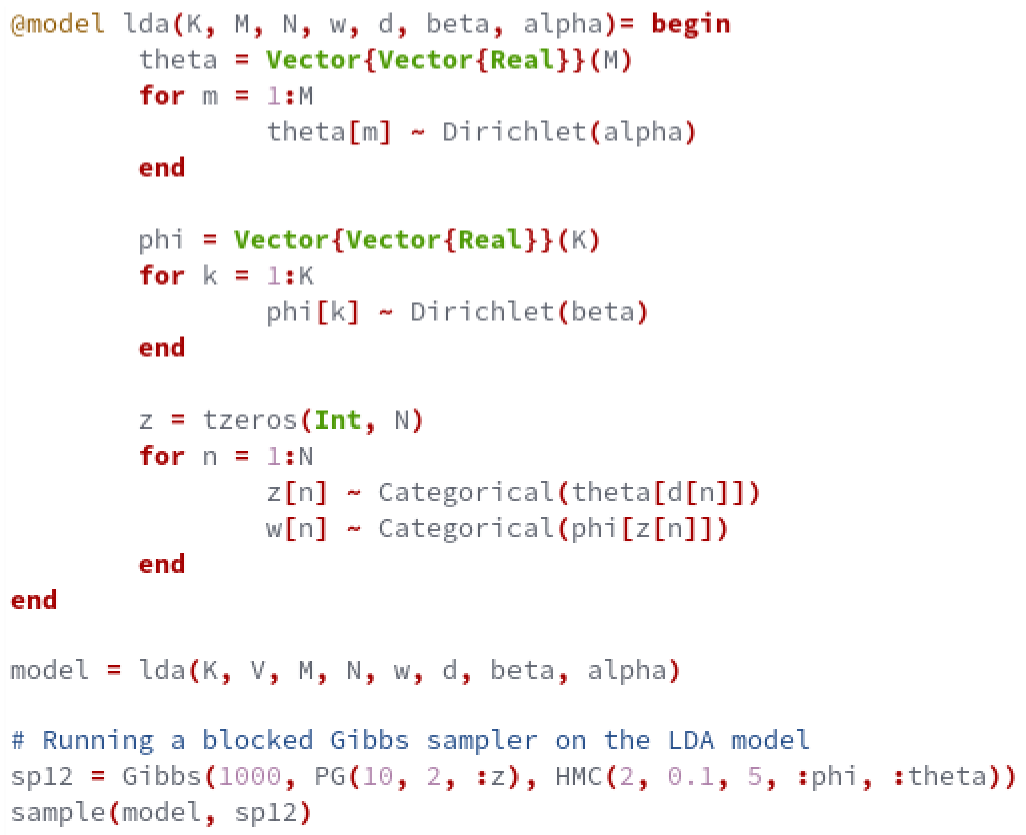
- Turing is a high-level probabilistic programming language, which provides extremely solid inference routines to the researcher
- Seamless compatibility with the entire Julia scientific machine learning stack enables many interesting cases
  - More on this later!
- Provided inference algorithms:
  - Hamiltonian Monte-Carlo
  - No-U-Turn Sampler
  - Automatic Differentiation Variational Inference
  - Normalizing Flows
- Different inference subroutines can be composed, hence allowing for individual inference subroutines

[36] Ge, H., Xu, K. and Ghahramani, Z., 2018, March. Turing: A Language for Flexible Probabilistic Inference. In International Conference on Artificial Intelligence and Statistics (pp. 1682-1690).

Turing

Syntax

- The *model* macro identifies our function as a probabilistic model to Turing, i.e. registers it for eventuel forward- and reverse-mode gradients
- θ, ϕ & z denote model parameters
- K, M, N, d, β, & α denote hyperparameters
- w denotes the observed data



```
@model lda(K, M, N, w, d, beta, alpha)= begin
        theta = Vector{Vector{Real}}(M)
        for m = 1:M
                theta[m] ~ Dirichlet(alpha)
        end

        phi = Vector{Vector{Real}}(K)
        for k = 1:K
                phi[k] ~ Dirichlet(beta)
        end

        z = tzeros(Int, N)
        for n = 1:N
                z[n] ~ Categorical(theta[d[n]])
                w[n] ~ Categorical(phi[z[n]])
        end
end

model = lda(K, V, M, N, w, d, beta, alpha)

# Running a blocked Gibbs sampler on the LDA model
spl2 = Gibbs(1000, PG(10, 2, :z), HMC(2, 0.1, 5, :phi, :theta))
sample(model, spl2)
```

Application performance

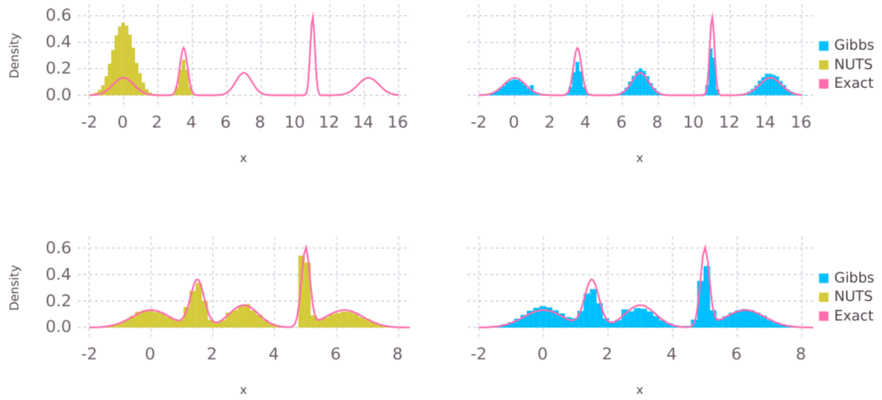


Figure: Performance on a Gaussian mixture model

Model	Dimensionality	Time (R) Turing (s)	Time (R) Stan (s)	Ratio (R)	Ratio (F)
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

Figure: Runtime comparison for Turing vs Stan for HMC.

Probabilistic Programming Frameworks

Summary

- There are many vectors we ought to consider when deciding upon a probabilistic programming framework
    - Can I represent my problem in the respectve DSL, or do I require a full-fledged language?
        - Python- or Julia-based probabilistic programming system
    - Do I require support for meta-programming?
        - Julia- or Lisp-based probabilistic programming system
    - How scalable and accelerator-portable is my framework supposed to be?
        - Probabilistic programming system with an XLA-backend
    - Do I want to interface with simulators?
        - Gen, or PyProb
    - Do I want to have inference programming capability?
        - Gen, or Venture
    - How high- or low-level do I want to program?

Outline

Introduction to Turing

- We will do our first steps in a probabilistic programming framework with Turing covering
  - The modelling syntax
  - Sampling
  - Accessing the trace
  - Automatic differentiation
  - Sampler visualization
- All content can be accessed in the Jupyter notebook **IntrotoTuring.ipynb**