SGHMCMC: A PYTHON PACKAGE FOR STOCHASTIC GRADIENT MARKOV CHAIN MONTE CARLO

Xiaoyu Jiang
Duke University
xj35@duke.edu

Luman Huang
Duke University
luman.huang@duke.edu

Abstract

In this paper, we present *sghmcmc*, a software package that implements the Stochastic gradient Markov chain Monte Carlo (SG-MCMC). Amid the rapid development of Bayesian statistics, it is ever more important to develop an efficient, accurate way of estimating posteriors. In the practical setting where models often involve thousands of variables, traditional Markov chain Monte Carlo (MCMC) fails to efficiently explore the parameter space with random walk updates; hence it is nature to use gradients, provided they exist, as a guidance of the updates. Hamiltonian Monte Carlo (HMC) is one variant that leverages the gradient to introduce a more efficient exploration of the posterior space and SG-MCMC is the stochastic version that only uses part of the data to approximate the gradient. This paper can be divided into three parts: a mathematical intuition behind HMC and SG-MCMC, a Python implementation of both MCMC variants mentioned above, and applications of SG-MCMC on both simulated and real world data sets.

Keywords: Bayesian, MCMC, SG-MCMC, Efficiency, Big Data

1 Introduction

With the recent development of Bayesian statistics and computational capacities, Markov Chain Monte Carlo (MCMC) becomes the defacto tool for Bayesian posterior inference. However, MCMCs' notoriously slow mixing rate in high-dimensional, complex models and poor scalability limit their use in real world applications. In the past decade, various modifications have been made to provide more efficient explorations of the posterior space, such as Hamiltonian Monte Carlo (HMC) and its manifold variants. HMC's efficiency comes by defining a *potential energy* function independent of the target posterior distribution and devising a continuous dynamics that explores the energy landscape, enabling proposals of distant states. Note that the gain in efficient exploration of the state spaces comes at the cost of updating gradients at each proposal step, which is huge for large date sets.

Recently, to address the hefty computational burden of calculating gradients, stochastic gradient variants of such methods are proposed and proved useful in large data sets. Our package *sghmcmc* implements one of the variant based on Chen et al.'s *Stochastic Gradient Hamiltonian Monte Carlo*. The goal of this package is to provide an easy to use interface for large scale Bayesian inference. We fully intend intend to use this package later in our own research and perhaps integrate it into various Bayesian machine learning pipelines.

2 Mathematical Formulation and Algorithms

2.1 Hamiltonian Monte Carlo

Suppose we sample from a posterior distribution of θ given a set of independent observations $x \in D$:

$$p(\theta|D) \propto p(\theta) \prod_{i=1}^{n} p(x_i|\theta) \tag{1}$$

We define potential energy function U to be:

$$U = -\sum_{i=1}^{n} \log p(x_i|\theta) - \log p(\theta) \tag{2}$$

Hamiltonian Monte Carlo (HMC) (Neal, 2010) provides a method for proposing θ in a Metropolis-Hastings (MH) framework that explores the state state space by introducing a set of auxiliary momentum variables r. Instead of sampling from $p(\theta|D)$, HMC samples from a joint distribution

defined by

$$\pi(\theta, r) \propto \exp(-U(\theta) - \frac{1}{2} r^T M^{-1} r) \quad (3)$$

We can simply discard r samples and the θ would have marginal distribution $p(\theta|D)$. Here, M is a mass matrix which defines a kinetic energy term when combined with r. Without more information regarding the target distribution, M is usually set to the identity matrix. Hamiltonian function is consequently defined by $H(\theta, r) = U(\theta) + \frac{1}{2} r^T M^{-1} r$. Intuitively, H measures the total energy in the system with position variables θ and momentum variables r.

Algorithm 1 Hamiltonian Monte Carlo

Input: Starting position $\theta^{(1)}$ and step size ϵ
for *t = 1,2, ...* **do**
  Resample $r^{(t)} \sim N(0, M)$
  $(\theta_0, r_0) = (\theta^{(t)}, r^{(t)}))$
  Simulate discretization of Hamiltonian dynamics in Eq.(4):
  $r_0 \leftarrow r_0 - \frac{\epsilon}{2} \nabla U(\theta_0)$
  for *i = 1 to m* **do**
    $\theta_i \leftarrow \theta_{i-1} + \epsilon M^{-1} r_{i-1}$
    $r_i = r_{i-1} - \epsilon \nabla U(\theta_i)$
  end
  $r_m \leftarrow r_m - \frac{\epsilon}{2} \nabla U(\theta_m)$
  $(\hat{\theta}, \hat{r}) = (\theta_m, r_m)$
  MH correction:
  $u \sim$ Uniform[0,1]
  $\rho = e^{H(\hat{\theta}, \hat{r}) - H(\theta^t, r^t)}$
  if $u < min(1, \rho)$ **then**
    $\theta^{t+1} = \hat{\theta}$
  end
end

HMC simulates Hamiltonian dynamics to propose samples

$$\begin{cases} d\theta = M^{-1} r dt \\ dr = -\nabla U(\theta) dt \end{cases} \quad (4)$$

Over any time, the Hamiltonian dynamics of Eq.(4) defines a mapping from the state t to the state t + s. It is trivial to check that this state preserves the total energy, H, so proposals are always accepted. In practice, simulating exactly from the continuous systems is impossible. Hence, some errors are introduced when we try to discretize the system and should be corrected by adding a MH step. As for dicretizing the system, one common approach is the "leapfrog" method, which is included in Alg 1.

A quick recap of the HMC algorithm. First, we drew a random momentum variable r which combined with θ define our energy level. Once the energy level is fixed, we change our θ and r according to the Hamiltonian dynamics defined in Eq. (4). Note that H is supposedly be fixed in this step. After exploring the energy landscape, we update with θ and a MH step to correct the error caused by discretization.

2.2 Naive Stochastic Gradient HMC

It's necessary to compute the gradient of energy function based on the entire dataset when implementing HMC. When facing massive and high-dimensional data, the gradient computation is hard to conduct. Stochastic gradient HMC is a way to take the advantage of HMC and solve the problem of computation complexity. A straightforward implementation of stochastic gradient HMC, also called Naive Stochastic Gradient HMC, is simply using $\nabla U(\hat{\theta})$ instead of $\nabla U(\theta)$. $\nabla U(\hat{\theta})$ is a noisy estimate of gradient based on a minibatch D randomly sampled from the entire dataset D.

$$\nabla U(\hat{\theta}) \approx \nabla U(\theta) + N(0, V(\theta)) \quad (5)$$

Here, V is the covariance of the stochastic gradient noise. This replacement modifies the equation of Hamiltonian dynamics simulation by affecting the update of momentum r. The updated simulation equations could be viewed as a discrete system of the following continuous differential equations:

$$\begin{cases} d\theta = M^{-1} r dt \\ dr = -\nabla U(\theta) dt + \text{N}(0,\ 2\text{B}\ (\theta) dt) \end{cases} \quad (6)$$

In the above equations, $B(\theta) = \frac{1}{2} \epsilon V(\theta)$. $N(0, 2B(\theta)dt)$ can be viewed as noise term.

One critical problem of naive stochastic gradient HMC is that the noise term affects the distribution of (θ, r) and leads it far away from target distribution $\pi(\theta, r)$. Therefore, the distribution of (θ, r) is no longer invariant under Eq. (6). Specifically, the H calculated based on minibatch data can be far away from the results of simulation based on entire dataset. We can solve this problem by inserting an MH step after short simulation runs. At this time, the acceptance rate could

be reasonable. However, each MH step requires costly computation based on the entire dataset, which increases the computation complexity and decreases Hamiltonian Monte Carlo's benefits on efficiency. Another method, which is the key point of this paper, is introducing friction term to HMC in order to alleviate the problems resulted from noise term. This method can help us achieve the desired invariant distribution $\pi(\theta, r)$ again while keeping the computation advantages of Hamiltonian Monte Carlo.

2.3 Stochastic Gradient HMC with Friction

As stated in Sec. 2.3, a friction term is included in the algorithm to alleviate the effect of noise and preserve the efficiency of Hamiltonian Monte Carlo at the same time. The update of momentum is modified as shown in Eq. (7).

$$\begin{cases} d\theta = M^{-1}rdt \\ dr = -\nabla U(\theta)dt - BM^{-1}rdt + \\ N(0, 2B(\theta)dt) \end{cases} \tag{7}$$

Intuitively, the friction term $BM^{-1}r$ helps decrease the energy $H(\theta, r)$, thus reducing the influence of the noise. This modified version of dynamics is known as *second-order Langevin dynamics* in physics. It has been shown that the distribution of (θ, r) computed by Eq. (7) is invariant as in the original Hamiltonian dynamics as Eq. (4). However, Eq. (7) can not be applied directly. In practice, we can't know the noise model B. We can only use the inaccurate estimation of B, $\hat{B}$, to update the Hamiltonian dynamics system in Eq.(7). An improved practical method is to introduce a user specified friction term C. The modified equations are shown in Eq.(8).

$$\begin{cases} d\theta = M^{-1}rdt \\ dr = -\nabla U(\theta)dt - CM^{-1}rdt + \\ N(0, 2(C - \hat{B})dt) + N(0, 2\hat{B}dt) \end{cases} \tag{8}$$

The corresponding Stochastic Gradient Hamiltonian Monte Carlo algorithm is shown in Alg.2. When applying SGHMC, users set the value of $\hat{B}$ and C by themselves.

3 Package Description

3.1 Algorithm Implementation in Python

We implement the SGHMC algorithm in python and wrap it up as a package "sghmcmc". Users can install the package simply using pip and import to their own python file. The package can implement SGHMC, naive SGHMC, and standard HMC.

Algorithm 2 Stochastic Gradient HMC

for *t = 1,2, ...* **do**
  optionally, resample momentum r as $r^{(t)} \sim N(0, M)$
  $(\theta_0, r_0) = (\theta^{(t)}, r^{(t)}))$
  Simulate dynamics in Eq.(8):
  for *i = 1 to m* **do**
    $\theta_i \leftarrow \theta_{i-1} + \epsilon_t M^{-1} r_{i-1}$
    $r_i \leftarrow r_{i-1} - \epsilon \nabla \hat{U}(\theta_i) - \epsilon_t C M^{-1} r_{i-1} + N(0, 2(C - \hat{B})\epsilon_t$
  end
  $(\hat{\theta}, \hat{r}) = (\theta_m, r_m)$
  MH correction:
  $u \sim$ Uniform[0,1]
  $\rho = e^{H(\hat{\theta}, \hat{r}) - H(\theta^t, r^t)}$
  if $u < min(1, \rho)$ **then**
    $\theta^{t+1} = \hat{\theta}$
  end
end

As stated before, SGHMC use minibatches to calculate gradient, so it has higher efficiency compared with HMC. HMC has a slightly better accuracy with a higher computation cost compare with SGHMC. They all perform as classes in the package. Users can choose the required algorithm according to their needs, and simply call the class and their embedded functions. Specifically, each algorithm can perform following critical functions and features:

- Users initiate the class and input the data (or not),loglikelihood, gradient of loglikelihood, initial guess of θ, and other possible user specified arguments.
- Function sampling: perform sampling process based on user inputs, including number of iterations, length of leapfrog step, step size, and size of minibatches (e.g. SGHMC: as shown in Alg.2).
- Users can get the samples by calling $class.res$, which returns a list of θ.
- The algorithms set default mass matrix as identity matrix, but user can also specify it.

Methods/Iteration	10	100	1000
`HMC (Parallelism)`	5.75	59.40	267.27
`HMC (Plain Python)`	8.19	79.72	391.43

Table 1: Parallelism Improvements (Processor Time)

3.2 Optimization

After implementing algorithms using plain Python straightforwardly, we optimize the code by JIT compilation and parallelism computation.

Numba package is introduced in this step. We use $@guvectorize$ to compile part of the velocity function, which needs to work on high-dimension multiplication. The improved velocity function cost 2.8% time of plain python version when dealing with 100-dimension data. However, it performs worse when dealing with low-dimension data (such as two dimensions). Therefore, it's better to apply this method when handling high-dimension problem.

Parallelism computation is achieved by using multiprocessing. We take advantage of CPU processing capability to deal with the calculation of $\nabla U(\theta)$ and $U(\theta)$. When we want to sample from a large data set, calculation of $\nabla U(\theta)$ and $U(\theta)$ needs to go through the dataset, which incurs multiple times computation on a function with different inputs. Parallelism computation can increase the efficiency and decrease the time cost. This method has significant effects on HMC algorithm. When calling the sampling function (sample θ) in a 10000-observation data set, parallelism computation can speed up 29.79% with 10 iterations, 25.5% with 100 iterations, and 31.72% with 1000 iterations (as shown in Table.1).

4 Examples

4.1 Application on Simulated Data sets

To explore the performance of our algorithm, we generated simple data sets and compared the outputs with data sets' true distributions. A standard normal distribution with 10 thousand data points was simulated. We used SGHMC to fit the data and sampled for mean value. We also consider HMC with M-H steps and naive SGHMC as sampling methods to make a comparison with SGHMC.

As shown in Fig.1, both HMC and SGHMC provides outputs close to the true distribution. Results of Naive SGHMC greatly diverges from the true distribution. These results are in line with the theory that both HMC and SGHMC (with friction) can maintain the distribution π as invariant distribution, while noise in naive SGHMC leads the output distribution far away from target distribution. Although it is insignificant, the result distribution provided by SGHMC has more bias compared with the result of HMC. However, SGHMC costs a lot less than HMC regarding to computation. In practice, when we deal with large data set or high dimension data, we usually want to decrease the accuracy in exchange for a higher computation efficiency.

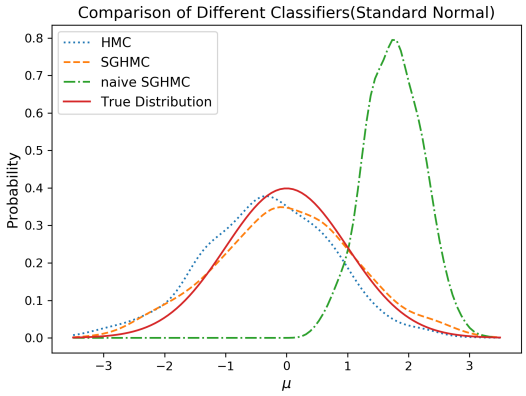

Figure 1: Empirical distributions associated with various sampling algorithms relative to the true target distribution $N(0, 1)$ (1000 iterations)

We also simulated a bivariate normal distribution and applied the above three algorithms to sample the mean value of the distribution (2-dimension problem). We plot the output μ samples in Fig.2. Similar with one-dimension case, SGHMC and HMC performs well, while naive SGHMC diverges significantly from the target distribution (as shown in Fig.2).

From the above cases, we can see that our SGHMC algorithm basically realizes its functions as high efficiency HMC with stable high accuracy and significant less computation cost.

4.2 Application on Real Data sets

We also applied our algorithms to a real-world problem. The data set we obtained contains the plasma glucose concentration of 532 females from a study on diabetes. We assume the plasma glucose concentration of female population is normally distributed. Our goal is to use SGHMC algorithm to find the parameters (mean) of the distribution.

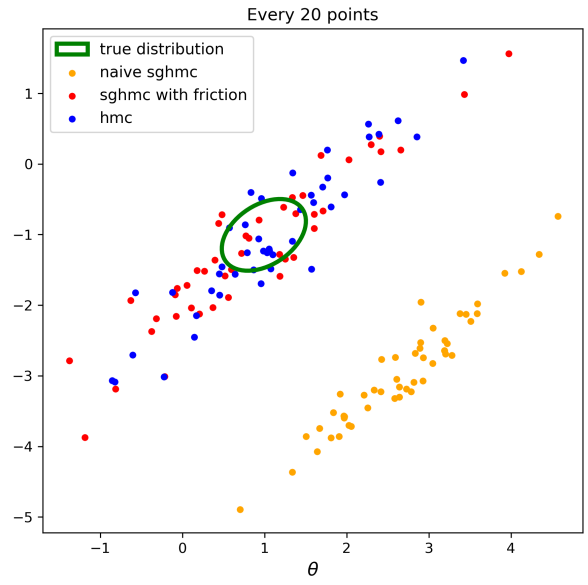

Figure 2: Empirical distributions associated with various sampling algorithms relative to the true target distribution $N(\mu, \sigma), (\mu = [1, -1], \sigma = \begin{bmatrix} 1 & 0.9 \\ 0.9 & 1 \end{bmatrix}$,sampling for μ) (1000 iterations)

We assume different prior distribution and examine the different outputs of SGHMC (as shown in Fig.3). From Fig.3, we can see that prior distribution has great influence on the simulation outputs. It takes effects when calculating $\nabla U(\theta)$.

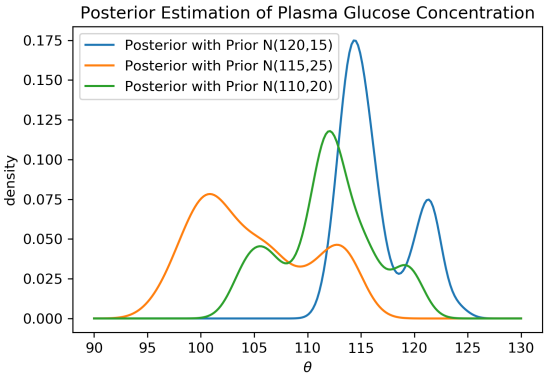

Figure 3: Posterior Distribution with Different Prior

5 Conclusion

SGHMC with friction provides a practical solution for the key challenge of HMC, inefficienct simulation when working on large-scale dataset. SGHMC uses the idea of minibatch to reduce the cost of gradient computation. In addition, the introduction of friction terms alleviates the negative influences of noise brought by minibatch. Our implementation and experiments validate the great performance of SGHMC. As a simulation method, SGHMC could be widely used to explore parameters' posterior distributions.

Some limitation still exists. We can still work on efficiency issues. When exploring target distributions, the directions that SGHMC updates θ are various and even opposite (U-Turn). The inconsistency in direction leads to inefficiency of algorithm. Therefore, further research can work on efficiency problem by modifying the algorithm to achieve No-U-Turn feature.

6 Reference

[1] T. Chen, E. Fox, and C. Guestrin. Stochastic Gradient Hamiltonian Monte Carlo. In $ICML$, 2014.