Implementation of Stochastic Gradient Hamiltonian Monte Carlo

Gongjinghao Cheng, Zhenyu Tian
Duke Univeristy

1 Abstract

The purpose of this project is to implement the Stochastic Gradient Hamiltonian Monte Carlo(SGHMC) algorithm introduced by Tianqi Chen, Emily B. Fox and Carlos Guestrin. SGHMC algorithm takes advantage from minibatch and speed up the Hamiltonina Monte Carlo(HMC) algorithm. In this report, we will first illustrate the backgrounds and details of the algorithm, implement and optimize it in code, and then apply it on simulated data and real data. Comparisons between different version of HMC samplers will be made in terms of accuracy and running time.

2 Background

Hamiltonian Monte Carlo is one of the most popular Markov Chain Monte Carlo (MCMC) method in recent years. This method resembles energy system by imitating potential energy by the target distribution as well as kinetic energy by 'momentum' auxiliary variables.

Whereas HMC will explore the state space quickly, there is one limitation of HMC: Gradient of the potential energy function is essential for HMC algorithm and it ultilize the whole data set, which in modern days are in millions or even billions. The huge computational cost encourages ideas of using minibatches instead of the whole data set. It is evident that a naive algorithm that simply replaces the whole data set by minibatches is not consistent. Accordingly, we need to add a MH (Metropolis-Hasting) correction for keeping consistency. However, the MH correction also needs considerable amount of computation power.

In the paper by Tianqi Chen, Emily B. Fox and Carlos Guestrin, a new algorithm proposal is made by using stochastic estimation of gradient term with a friction term to offset the noises induced from the estimation process.

3 Algorithm

The basic Hamiltonian Monte Carlo method has the following settings. Suppose we have data $x_i \sim p(x|\theta),\ i = 1, ..., n$. We want to estimate θ by sampling from the following

posterior distribution:

$$p(\theta|x) \propto exp(-U(\theta)),$$
$$U(\theta) = -\sum_{x \in D} logp(x|\theta) - logp(\theta).$$

U is called potential energy function. The HMC algorithm generates the samples from the following joint distribution:

$$\pi(\theta, r) \propto exp(-H(\theta)),$$
$$H(\theta, r) = U(\theta) + \frac{1}{2} r^T M^{-1} r.$$

$H(\theta, r)$ is the summation of potential energy term and kinetic energy term and is known as the Hamiltonian function. By comparing this distribution with the posterior distribution, it is clear that the posterior distribution could be obtained by discarding the kinetic energy term. The reason that we introduce this auxiliary kinetic variable is that Hamiltonian can be simulated by the following dynamics:

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

The resulting HMC sampling algorithm is as below:

Algorithm 1 Hamiltonian Monte Carlo Algorithm (Chen et al.,2014)

Input: Starting position $\theta^{(1)}$ and step size ϵ
for t = *1,2,...* **do**
  Resample momentum r
  $r^{(t)} \sim N(0, M)$
  $(\theta_0, r_0) = (\theta^{(t)}, r^{(t)})$
  Simulate discretization of Hamiltonian dynamics:
  $r_0 \leftarrow r_0 - \frac{\epsilon}{2}\nabla(\theta_0)$
  for $i = 1$ *to* m **do**
    $\theta_i \leftarrow \theta_{i-1} + \epsilon M^{-1}$
    $r_i \leftarrow r_{i-1} - \epsilon \nabla U(\theta_i)$
  end
  $r_m \leftarrow r_m - \frac{\epsilon}{2}\nabla U(\theta_m)$
  $(\hat{\theta}, \hat{r}) = (\theta_m, r_m)$
  Metropolis-Hastings 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

Note that this method requires the users to users to calculate the gradient $\nabla U(\theta)$, which could be very complicate for large datasets. To avoid this computation, SGHMC takes a minibatch of the data, and estimate its gradient by adding friction and noise terms, resulting in the following dynamics:

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

Here, B reresents the noise model. The authors mentioned that B is usually not known in practice. Instead, we do have an estimation $\hat{B}$. Therefore, they introduced a user specified friction term $C \succeq \hat{B}$ and considered the following dynamics:

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

The resulting algorithm is as following:

Algorithm 2 Stochastic Gradient Hamiltonian Monte Carlo Algorithm (Chen et al.,2014)

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:
  for $i = 1$ *to* m **do**
    $\theta_i \leftarrow \theta_{i-1} + \epsilon_t M^{-1} r_{i-1}$
    $r_i \leftarrow r_{i-1} - \epsilon_t \nabla \tilde{U}(\theta_i) - \epsilon_t C M^{-1} r_{i-1} + N(0, 2(C - \hat{B})\epsilon_t$
  end
  $(\theta^{(t+1)}, r^{(t+1)}) = (\theta_m, r_m)$
end

4 Description of Modules

The functions we create and archive in package 'sghmc' are SGHMC and SGHMC_parallel, with inputs:

1. theta0: The initial values chosen by user
2. X: Data
3. gradU: The function compute gradient of $U(\theta)$ based on data and current theta
4. eps: Learning rate
5. sample_size: Number of samples generated from posterior distribution
6. B: Noise model chosen by user
7. C: Upper bound of B s.t. (C-B) is positive definite
8. batch_size: Number of data in each batch
9. burnin: Number of samples dropped initially for the HMC samples
10. n: Number of cores used, only applicable for SGHMC_parallel
11. M: The mass matrix, set to be identity in default

Both functions will provide θ's sampled from their posterior distribution.

5 Algorithm Optimization

We started by using the most basic code to implement the algorithm, then we profile the code to check the slowest part of the algorithm. It turns out that the multivariate normal sampling function from numpy costs the majority of the time. Then we use statistical knowledge to improve the efficiency of sampling from multivariate normal distribution, and then profile the code again. The result shows that function computing gradient of U takes significant amount of time. Hence, we decide to use numba to reduce its time consuming feature. Lastly, we used paralell process and further reduce the running time.

We applied the function from each optimizing step to $U(\theta)$ below (first example in 'Application to Simulated Data' section) in order to illustrate the increasing efficiency. We used a sample size of 10000 and a batch size of 500, kept other parameters the same, and run on the same simulated data.

Table 1: Running Time of Different Versions of SGHMC

Function	basic code	vectorized code	numba code	parallel code
Time	$13.6s(\pm 81.8ms)$	$3.03s(\pm 52ms)$	$2.33s(\pm 28ms)$	$1.2s(\pm 73.3ms)$

According to the table, it is evident that our numba version of function is about 6 times faster than the basic code in this case; the parallel code will be twice faster than numba code with 4 cores.

6 Application to Simulated Data

6.1 Example From Paper

This is the example from paper. The basic setting is as following:
1.$U(\theta) = -2\theta^2 + \theta^4$
2. $\nabla\tilde{U}(\theta) = \nabla U(\theta) + N(0, 4)$

Figure 1: Sampled θ Distribution vs True distribution

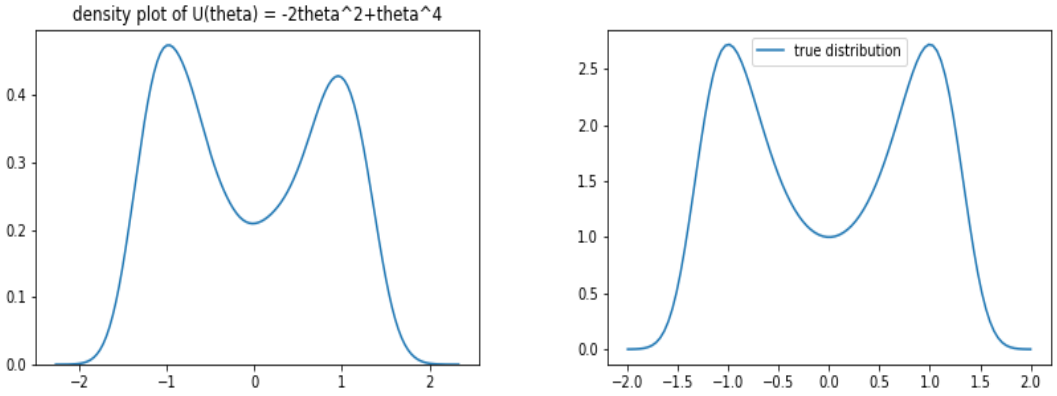


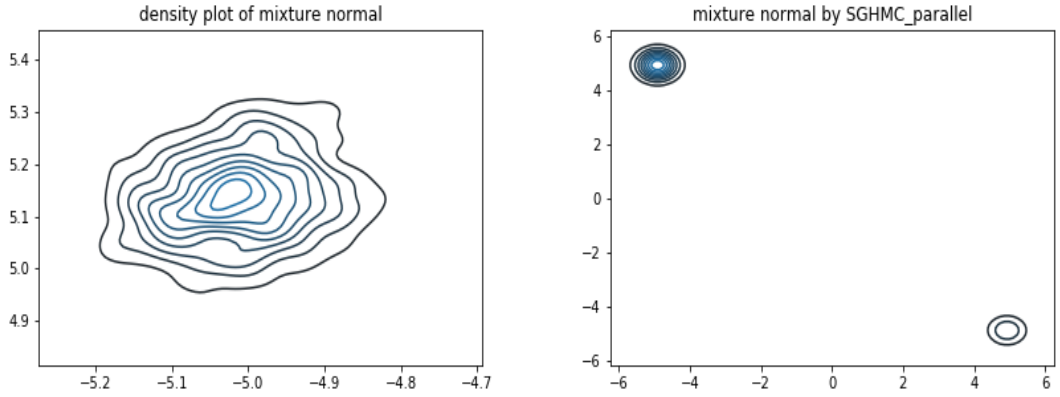
Figure 1 shows the simulated density (left) and the true density (right) of θ. Since the two densities have very similar shape, our sghmc module seems to work well in this case.

6.2 Example of Mixture Normal

We simulate 200 samples from distribution with density $p(x) = 0.5N(x|\mu_1, 1)+0.5N(x|\mu_1, 1)$, where $\mu = (\mu_1, \mu_2) = (5, -5)$.

Suppose we do not know true μ and want to estimate them. We will use our sghmc method to approximate μ.

Figure 2: SGHMC VS. SGHMC_parallel



In Figure 2, the left plot is produced by the result from SGHMC and the right plot is produced from SGHMC_parallel. The figure shows that the result of SGHMC_parallel will converge to two modes while the result of SGHMC only converge to one mode. It seems single chain algorithm will converge to one of the mode quickly, while multi chain algorithm allows the parameters to converge to different modes on different chains.

6.3 Example of Linear Model

We want to use sghmc to estimate the parameters of linear model. We perform the algorithm on simulated data with known true parameters, and compare the resulting estimates to the true paramaters.

Suppose we want to simulate 100 observations from a linear model with a 1-dim response variable and 3 predictors, the model could be expressed as below:

$$y|x, \alpha, \beta, \sigma^2 \sim N(\alpha + x^T\beta, \sigma^2)$$

where y is the response variables, x are the vector of 3 predictors with shape (3,1), α are the intercept, β are the vector with shape (3,1) of coefficeints for 3 predictor, σ^2 is the variance of noise.

To sample y from above model, we first sample x from $MVN(0, I_3)$. We set true $\alpha = 1$, true $\beta = (2, 3, 4)^T$ and true $\sigma^2 = 1$. Next, we sample noise from $N(0, \sigma^2)$. Finally we compute $y = \alpha + x^T\beta + noise$.

Repeat the above procedure 100 times to generate 100 samples of x, y. Note that we may vectorize y and matrixrize x so that we may compute y collectively.

Now apply our sghmc module for sampling from $p(\alpha, \beta, \sigma^2|Y, X)$. Note that we need to log transform σ^2 since we need the hmc algorithm to explore the whole state space freely. The true $log(\sigma^2) = log(1) = 0$

The results are shown in Table 2.

Table 2: Linear Model Test Result on Simulated data

parameter	α	β_1	β_2	β_3	$log(\sigma^2)$
posterior mean	0.985	1.875	2.855	3.922	-0.097

The posterior are close to the true values. The rmse based on the posterior estimation of parameters is 9.09.

Figure 3: Estimated Y VS. True Y

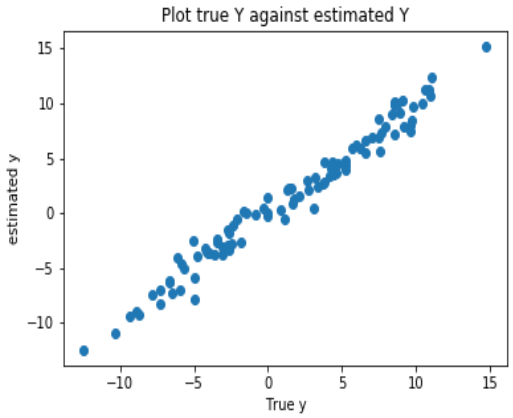


Figure 3 plots estimated y against true y. It is clear that the in general, true y and estimated y are close since they are along the 45 degree line, which means that they are near each other.

7 Application to Real Data

7.1 Example 1: linear model on azdiabetes data set

This data set is available on http://www2.stat.duke.edu/ pdh10/FCBS/Exercises/. It Contains the information gthered from diabete tests of 532 women living near Phoenix,

Arizona. There are 7 variables in this data set. For simplicity and purpose of illustration, we will only use the first 5 variables:

1. npreg: Number of pregancy
2. glu: glucose level
3. bp: Blood pressure
4. skin: Skin thickness
5. bmi: Body mass index

In this example, we will try to regress glu on other variables (npreg, bp skin, bmi, ped age) with our modules.

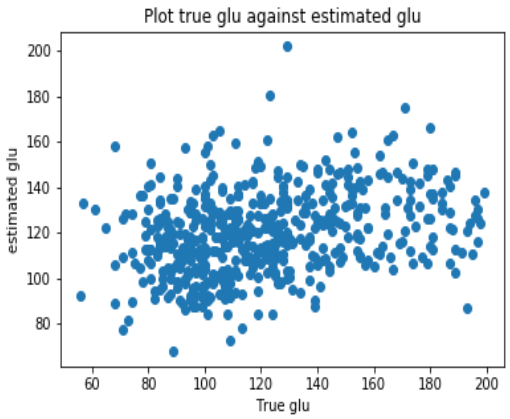
The estimated coefficient are shown in Table 3.

Table 3: Linear Model Test Result on Diabete Data Set

parameter	α	β_1	β_2	β_3	β_3	$log(\sigma^2)$
posterior mean	2.985	0.967	0.848	0.153	1.469	6.899

With the estimated coefficients, we estimate y and plot true y against estimated y:

Figure 4: Estimated Glucose Level VS. True Glucose Level



From Figure 4, we could see that the estimation is not very accurate, which is result from the non-linearity of the data. Therefore, We choose to test on another real data that possesses linearity.

7.2 Example 2: linear model on salary data

This data set is available on https://www.kaggle.com/vihansp/salary-data. The data set is about salary rate. It contains 30 observations and two variables, the yearly salary and years of experience.

We fit a simple regression model by regressing yearly salary on years of experience. We first transform yearly salary by shrinking it 10000 times in order to regularize. Then we apply our module to estimate the coefficients

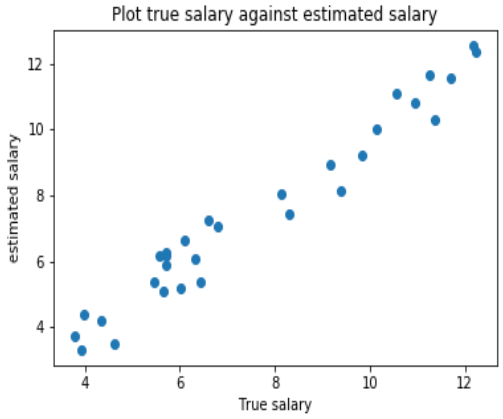
The estimated coefficient are shown in Table 4.

Table 4: Linear Model Test Result on Salary Data

parameter	α	β_1	$log(\sigma^2)$
posterior mean	1.878	1.040	-0.569

Figure 5 shows the relationship between true salary and estimated salary. Based on this plot, we clearly see that the points are near the 45 degree line, which means that our module is doing well in estimating the parameters.

Figure 5: Estimated Salary VS. True Salary



8 Comparative analysis with competing algorithms

We compared the performance of SGHMC_parallel function with the samplers from Pyhmc and PyStan packages in terms of the accuracy and time. To test the performance, we implemented the samplers on the first simulate example with the same settings mentioned in Section 5, and the results are showing below.

Figure 6: Comparison of the θ's Sampled by Different Samplers

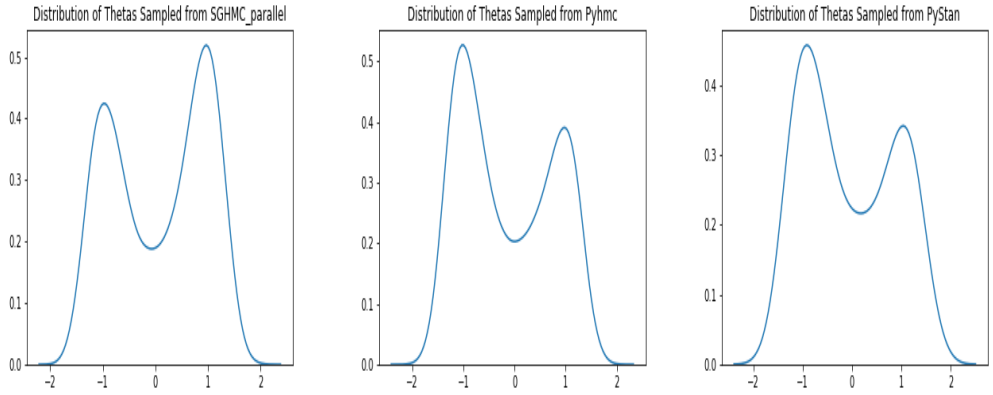


Figure 6 shows that the shape of all sampled θ distribution have similar shapes, which means all three samplers are doing great in terms of accuracy.

Table 5: Running Time of Different Samplers

Sampler	SGHMC_parallel	Pyhmc	PyStan
time	1.2s $\pm 73.3ms$	350ms $\pm 12.9ms$	1min 9s $\pm 522ms$

Table 5 shows that our function is significantly faster than the sampler from PyStan but slower than Pyhmc. It seems like our code could be further optimized by using other methods such as cythonize.

9 Conclusion

SGHMC uses the fundamental structure of HMC, but takes advantage from mini batch and avoids time consuming process computing the gradient based on full data. The friction offsets the noise generated by the stochastic estimates. According to our test results, sghmc is a valid mcmc sampler that generates qualified posterior samples.

However, there are some limitations of the algorithm. First of all, sghmc is not efficient in exploring the whole state space. As the example of mixture normal displays, the SGHMC function are not exploring the whole space. The solution to this is to apply several hmc chains, as the results from SGHMC_parallel has managed to overcome this limit. On the other hand, in order to perform this algorithm, we need to approximate large amount of statistics, such as C, B, M. Precise estimations of these values sometimes are the prerequisite of a good results. Besides, as a stochastic algorithm, we need more iteration steps for convergence. For further improvement, we could cythonize the code into

C++ language and further improve the running speed of the algorithm. Also, our current version of parallel function requires the users to set up iPython clusters by themselves. If we have more time, we will modify this function to a more user-friendly version.

10 Github Url and Installation Instruction

To reproduce the results in this report, please check the code available at the following Github Repository: https://github.com/kingcheng12/Stochastic-Gradient-HMC-with-Friction.

To Install our package, please run the following command:

pip install –index-url https://test.pypi.org/simple/ SGHMC-GOZH

More details can be found in the README file in the repository.

11 Contribution

Zhenyu Tian: SGHMC module and optimization, testing performance, report write-up.

Gongjinghao Cheng: repository maintenance and organization, packaging, linear model application, report write-up.

References

[1] Chen Tianqi, Fox B Emily, Guestrin Carlos. *Stochastic Gradient Hamiltonian Monte Carlo*. University of Washington, Seattle, WA. 2014.

[2] "HmcSampler." MATLAB & Simulink, www.mathworks.com/help/stats/bayesian-linear-regression-using-hamiltonian-monte-carlo.html.

[3] Hoff, Peter D. *A First Course in Bayesian Statistical Methods*. Springer Publishing Company, Incorporated. 2009