Stochastic Gradient Hamiltonian Monte Carlo

Jiawei Chen, Mingxuan Yang

Statistical Science

Duke University

Abstract

April 30, 2020

In the era of big data, the overwhelming amount of data input can easily render popular statistical methods useless. Hamiltonian Monte Carlo simulation algorithm (HMC) is one of such widely-applied MCMC simulation techniques, famous for its fast convergence speed and the ability to work with correlated samples. However, the algorithm requires to compute the gradient of the potential energy function, which can be computationally prohibitive due to immense datasets that are common nowadays. Thus, to address this problem, Stochastic Gradient Hamiltonian Monte Carlo (SGHMC) algorithm is proposed, which eases the computation burden by introducing the concept of gradient descent while maintaining the advantages of HMC in the meantime. In this paper, we implemented the SGHMC algorithm in Python, optimized code efficiency and applied it to a series of simulation tasks and real-world examples to demonstrate its edge in speed and accuracy.

Keywords— SGHMC, MCMC, Hamiltonian Dynamics, Simulation

1 Introduction

(The content of this paper is based on *Stochastic Gradient Hamiltonian Monte Carlo* by Chen et al [1].)

Simulation is an important tool to approximate and understand natural systems and processes, and is a widely applied technique in many fields, such as economics, ecology and statistics. Especially, it is an indispensible component in Bayesian inference, in order to understand the overly complicated posterior and predictive distributions.

Traditionally, a popular method to simulate from the target distribution $p(\theta)$ is to use Metropolis-Hastings algorithm. It is often implemented with a proposed normal random walk centered around the previously simulated data point, θ_0, to generate the next simulated candidate, θ'. Then a Metropolis-Hastings Correction step is carried out, i.e. accept the candidate with probability $\frac{p(\theta')}{p(\theta_0)}$, which ensures the simulated distribution converges to the target distributin.

However, as we can see from the sampling framework, the MH algorithm has the potential to result in high correlations between successive simulated data points. This means if we want to draw independent samples, we would have to discard a considerable amount of points and take only every n^{th} of them, which indicates a waste in computation power.

1.1 Hamiltonian Monte Carlo Simulation

An alternative, or improvement, to MH algorithm is to develop Hamiltonian dynamics $H(\theta, r) = U(\theta) + K(r)$, with a leapfrog finite difference method to discretize the system:

$$\begin{aligned} r_i(t+\frac{\delta}{2}) &= r_i(t) - \frac{\delta}{2} * \frac{\partial U(\theta)}{\partial \theta_i(t)} \\ \theta_i(t+\delta) &= \theta_i(t) + \delta * \frac{\partial K(r)}{\partial r_i(t+\frac{\delta}{2})} \\ r_i(t+\delta) &= r_i(t+\frac{\delta}{2}) - \frac{\delta}{2} * \frac{\partial U(\theta)}{\partial \theta_i(t+\delta)} \end{aligned}$$

where δ is the chosen step size. A detailed explanation on how HMC works is attached in **Appendix A**.

Unlike the traditional MH approach, the Hamiltonian dynamics perform pretty well in terms of getting rid of correlations between simulated samples. This is because we can draw a random set of momentum r in the beginning of each iteration, thus perturbing the dynamics to effectively explore the entire sample space of $p(\theta)$.

1.2 Stochastic Gradient HMC

Although the HMC method looks promising, what should be kept in mind is that we need to know

$$\frac{\partial U(\theta)}{\partial \theta_i} = \frac{\partial(-\log(p(\theta)))}{\partial \theta_i}$$

In a Bayesian paradigm, we usually aim to simulate from the posterior distribution, which can be written as the following form:

$$\begin{aligned} p(\theta \mid D) &\propto p(D \mid \theta) * p(\theta) \\ &\propto e^{\log(p(D|\theta))+\log(p(\theta))} \\ &\propto e^{\sum_{x\in D} \log(p(x|\theta))+\log p(\theta)} \\ &\propto e^{-U(\theta)}, \text{ where } \mathrm{U}(\theta) = -\sum_{\mathrm{x}\in\mathrm{D}} \mathrm{log}(\mathrm{p}(\mathrm{x} \mid \theta)) - \mathrm{log}\,\mathrm{p}(\theta)) \end{aligned}$$

As we can see, in order to know the gradient, we would have to perform a calculation using the entire dataset, which can be computationally prohibitive. This is the first problem we are going to settle with the proposed SGHMC algorithm. Besides, the process of carrying out the MH correction step also faces a tradeoff of efficiency and computation. This is the second problem that SGHMC can solve.

1.2.1 Naive Stochastic Gradient HMC

To address the first problem, we can consider using stochastic gradient HMC to reduce the burden of gradient calculation:

$$\nabla\tilde{U}(\theta) = -\frac{|D|}{|\tilde{D}|}\sum_{x\in\tilde{D}} \nabla\log(p(x \mid \theta)) - \nabla\log p(\theta))$$

where $\tilde{D}$ is a subset of our dataset.

With CLT, we can approximate the stochastic gradient with the help of a normal process: $\nabla\tilde{U}(\theta) \approx \nabla U(\theta) + \mathcal{N}(0, V(\theta))$ where $V(\theta)$ is the covariance of the stochastic gradient noise, depending on the current model parameters and sample size. Based on past experience, a minibatch size of only hundreads of data points suffices to make this CLT approximation accurate. Thus our Hamiltonian equation becomes:

$$\frac{\partial \theta}{\partial t} = \frac{\partial K(r)}{\partial r}$$
$$\frac{\partial r}{\partial t} \approx -\nabla \tilde{U}(\theta)$$

Sometimes it is also written as:

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

with $B(\theta) = \frac{1}{2}\epsilon V(\theta)$ being the diffusion matrix contributed by gradient noise.

However, when introducing a non-zero B, we can notice that the energy is no longer invariant in this Hamiltonian system, since we incorporate an additional noise into the system. This indicates that the simulated $p(\theta, r)$ can be very far from the target joint distribution. Therefore, we need to introduce a MH correction step sometime during the simulation, even if we do not consider the errors brought in by the finite difference method.

Then it comes to the tradeoff of computation and efficiency. If we introduce a correction step only after a short run of simulation, then with a large dataset it can be fairly computationally expensive; on the other hand, if we correct the procedure after a long run, then the acceptance ratio could be low, thus wasting the entire simulation result, which indicates inefficiency.

Therefore, we would like to think of some method to counter-act the noise introduced by stochastic gradient, and thus avoid such tradeoff and in the mean time address problem 2 as mentioned above.

1.2.2 Stochastic Gradient HMC

To reduce the influence of the introduced noise, we consider adding an extra ’friction term’ into our system.

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

To simplify the notation, we omit the dependence of B on θ for the remainder of the paper. The introduced friction term $BM^{-1}rdt$ helps to decrease the energy in the whole system and therefore reduce the influence of the noise. This is sometimes called *second-order Langevin dynamics*. With some proof, it can be shown that under the dynamics, our target joint distribution is the unique stationary distribution [1].

By far we assumed that the noise term relevant to B is known, but in practice it is not the case. Therefore, to avoid messing up with an inaccurate estimate on B, we could have an alternative version of the SGHMC as the following:

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

where C is specified by the users such that $C \succeq \hat{B}$

when we have perfect estimate $\hat{B} = B$ with this modification, the momentum update can be expressed as:

$$dr = -\nabla U(\theta)dt - CM^{-1}rdt + \mathcal{N}(0, 2Cdt)$$

As is shown, this is very close to the theoretical update in the previous section. Thus under this dynamics we can still have our target joint distribution as the unique stationary distribution.

When we do not have a perfect estimate, such as when $\hat{B} = 0$, we can show that as step $\delta \to 0$, $B = \frac{1}{2}\delta V$ tends to be 0. This means

$$\begin{aligned} dr &= -\nabla U(\theta)dt - CM^{-1}rdt + \mathcal{N}(0, 2(C - \hat{B})dt) + \mathcal{N}(0, 2B(\theta)dt) \\ &\to -\nabla U(\theta)dt - CM^{-1}rdt + \mathcal{N}(0, 2(C - \hat{B})dt) \\ &= -\nabla U(\theta)dt - CM^{-1}rdt + \mathcal{N}(0, 2Cdt) \quad \text{if } \hat{\text{B}} \text{ is set to 0} \end{aligned}$$

which is again dominated by C and thus can be specified by the users.

2 Algorithm and Optimization

2.1 Description of Algorithm

Essentially, the SGHMC algorithm is just an improvement to the HMC simulation method. The key difference is that, the SGHMC no longer employs leapfrog finite difference method in describing the Hamiltonian Dynamics of the system, because when learning with big data it can be computationally expensive to calculate the requested gradient on the whole dataset and to carry out MH correction step.

Instead, it adopts the concept of stochastic gradient and applies that to simulate the dynamics of the system, together with an introduced friction term to help counter-act the random noise resulted from stochastic gradient, in order to preserve the energy. With the introduction of friction terms and the adoption of stochastic gradients, it is expected to achieve faster speed in simulation, yet in the meantime retain the advantage of original HMC algorithm, which is to address the problem of correlation among simulated samples.

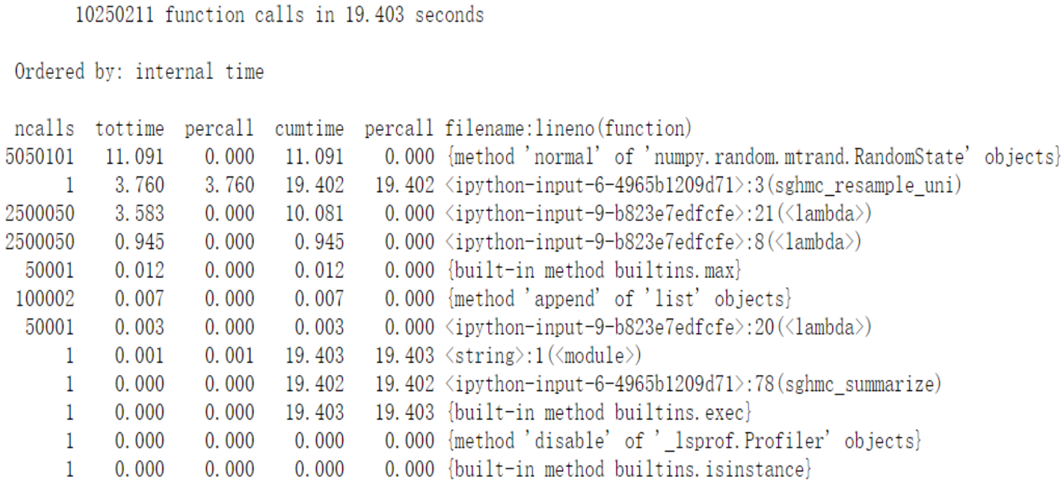
The SGHMC algorithm can be written as the following pseudocode:

```
for $\mathbf{t = 1, 2, ..., do}$
    optionally resample momentum :
    $r^{(t)} \sim \mathcal{N}(0, M)$
    $(\theta_0, r_0) = (\theta^{(t)}, r^{(t)})$
    simulate the new dynamics :
    for $\mathbf{i = 1, 2, ...m,\ do}$
        $\theta_i = \theta_{i-1} + \epsilon_t M^{-1} r_{i-1}$
        $r_i = r_{i-1} - \epsilon_t \nabla \tilde{U}(\theta_i) - \epsilon_t C M^{-1} r_{i-1} + \mathcal{N}(0, 2(C - \hat{B})\epsilon_t)$
    end for
    $(\theta^{(t+1)}, r^{(t+1)}) = (\theta_m, r_m)$
end for
```

2.2 Optimization of Algorithm

The basic set of codes were written using Python 3.8.0 in a fairly straightforward way.

In order to enhance its performance, we profiled the basic version by assigning a task to draw $50,000$ samples from a known target distribution with potential energy function as $U(\theta) = -2\theta^2 + \theta^4$. **Figure 1** below shows the profile result sorted by time.

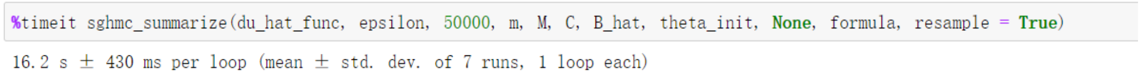


```
         10250211 function calls in 19.403 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  5050101   11.091    0.000   11.091    0.000 {method 'normal' of 'numpy.random.mtrand.RandomState' objects}
        1    3.760    3.760   19.402   19.402 <ipython-input-6-4965b1209d71>:3(sghmc_resample_uni)
  2500050    3.583    0.000   10.081    0.000 <ipython-input-9-b823e7edfcfe>:21(<lambda>)
  2500050    0.945    0.000    0.945    0.000 <ipython-input-9-b823e7edfcfe>:8(<lambda>)
    50001    0.012    0.000    0.012    0.000 {built-in method builtins.max}
   100002    0.007    0.000    0.007    0.000 {method 'append' of 'list' objects}
    50001    0.003    0.000    0.003    0.000 <ipython-input-9-b823e7edfcfe>:20(<lambda>)
        1    0.001    0.001   19.403   19.403 <string>:1(<module>)
        1    0.000    0.000   19.402   19.402 <ipython-input-6-4965b1209d71>:78(sghmc_summarize)
        1    0.000    0.000   19.403   19.403 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
```

Figure 1: Profile Result

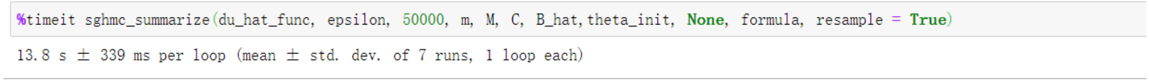
As we can see, the code is running rather slow without any optimization, taking a total amount of 19.4 seconds to finish the simulation. Also, it can be told that the most time-consuming part is the process of generating random normal varibles using ‘numpy‘ package. Besides, the ‘lambda‘ functions also seem to make the function clumsy, which are actually user-specified $\nabla\tilde{U}(\theta)$ functions. Thus, our target is to optimize this basic version focusing on this two main areas.

The first method we tried was altering the algorithm and adapting more efficient data structures. We re-organized the code to avoid redundant computations within for loops and converted list data structure into numpy array to increase speed. With this first method, we are able to increase the speed of main function *sghmc_summarise* from 16.2s to 13.8s, if using a timeit line magic. (**Figure 2** vs **Figure 3**)



```
%timeit sghmc_summarize(du_hat_func, epsilon, 50000, m, M, C, B_hat, theta_init, None, formula, resample = True)
16.2 s ± 430 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

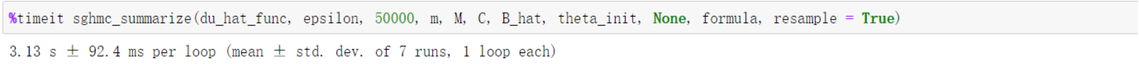
Figure 2: Basic Version



```
%timeit sghmc_summarize(du_hat_func, epsilon, 50000, m, M, C, B_hat,theta_init, None, formula, resample = True)
13.8 s ± 339 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Figure 3: First Attempt

Then, we also employed ‘numba‘ optimization method to compile the code and increase efficiency. The obstacle here was that ‘numba‘ only supported function input if it was already JIT-compiled. However, it does not make sense to ask users of our package to first run JIT compilation on their input function, *du_hat_func*. At last, we managed to achieve it by using a closure to compile the function in nopython mode inside the main *sghmc_summarize* function and the result was quite satisfactory. As can be seen from **Figure 4**, the time to draw 50, 000 samples drops to only 3.13s.



```
%timeit sghmc_summarize(du_hat_func, epsilon, 50000, m, M, C, B_hat, theta_init, None, formula, resample = True)
3.13 s ± 92.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Figure 4: JIT Compilation

During this process, we have also attempted ‘cython‘ to further optimize the code. Nevertheless, due to the unavoidable user-specified function input and heavy reliance on generating random numbers using ‘numpy‘ libaray, the effect was not very sound. Therefore, we decided to stick to JIT compilation. The optimized code can be found from the github link provided in **Appendix B**.

3 Code Testing

In order to make sure the code can run reasonably smoothly, `assert` command is widely used to check input types and values.

In this project, step size ϵ, initial value `theta_init` and `r_init` are `assert`ed to be positive numbers. Inputs `M`, `C` and `B_hat` are also guaranteed to be no less than 0 using `assert`. When `theta_init` is multi-dimensional, the shapes for these variables are checked to make sure that the corresponding matrix calculations are legal.

At last, in order to test whether the simulated distributions are reasonably close to the target distribution, we plot the empirical probability density function (pdf) of the simulated ones versus the true underlying pdfs. Two sets of plots, with one being a unimodal target distribution and the other being multi-modal, are displayed to demonstrate the accuracy of drawn samples. This external test python file can also be found in the github link in **Appendix B**.

4 Application and Comparitive Analysis

In this section, we implemented the SGHMC algorithm, together with a series of other similar algorithms such as a Stochastic Gradient Langevin dynamics algorithm (SGLD), a Stochastic Gradient Descent algorithm (SGD) with momentum and a standard HMC to see how well our implemented SGHMC performed and compared its result with other algorithms. Note that based on the target of comparitive analysis, we simply coded the other algorithms ourselves so that it would be more intuitive in terms of adjusting input data and tuning parameters.

4.1 Simulation

In this part, we choose to demonstrate the difference among various algorithms based on simulations with known underlying distributions.

4.1.1 Hamiltonian Dynamics Comparison

In the first part, our target distribution is characterized by a potential energy function $U(\theta) = \frac{1}{2}\theta^2$. We repeatedly draw points (θ, r) based on: 1. Naive Hamiltonian Dynamics without resampling r (Noisy Hamiltonian dynamics); 2. Naive Hamiltonian Dynamics resampling r every 50 steps(Noisy Hamiltonian dynamics); 3. SGHMC Dynamics (Noisy Hamiltonian dynamics with friction); 4. standard Hamilton Dynamics as a benchmark, indicating the true distribution.

We choose to draw 1000, 300, 300 and 1000 points from the four dynamics respectively, merely for a easily recognizable plot. The noisy gradient is chosen as $\nabla\tilde{U}(\theta) = \theta + \mathcal{N}(0, 4)$, with the step δ being 0.1. Then we plot them in a scatter plot to check the difference among the four dynamics in **Figure 5**.

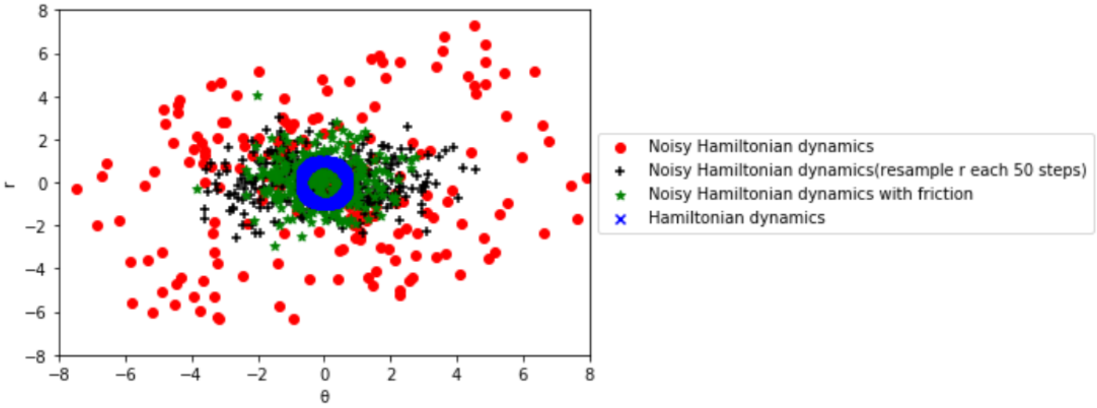


Figure 5: Hamiltonian Dynamics

As we can see, a pure Naive Hamiltonian Dynamics without resampling can lead to a very sparse set of points, while incorporating resampling momentum does help to control the divergence. If an extra friction is added, then the divergence can be further constrained and closer to the true distribution. This shows that with an appropriate friction term, our SGHMC algorithm can have a performance as good as HMC while having the advantage of fast implementing speed.

4.1.2 1-D Simulation Task

In this part, we are going to carry out a one-dimentional simulation task using the following algorithms: 1. SGHMC; 2. Naive SGHMC (with and without a MH correction step); 3. Standard HMC (with and without a MH correction step), and compare the draws with the true target distribution.

The selected target distribution is characterized by a potential energy function as

$U(\theta) = -2\theta^2 + \theta^4$. The noisy gradient is chosen as $\nabla\tilde{U}(\theta) = \nabla U(\theta) + \mathcal{N}(0, 4)$, with momentum being resampled every 50 steps for the HMC variants.

Figure 6 below shows the empirical distributions associated with the algorithms relative to the true underlying distribution. As we can see, all algorithms generate a reasonably close pdf to that of the true distribution except the Naive SGHMC without MH correction step. This result again confirms that our implementation of SGHMC is successful since it converges to the true distribution.

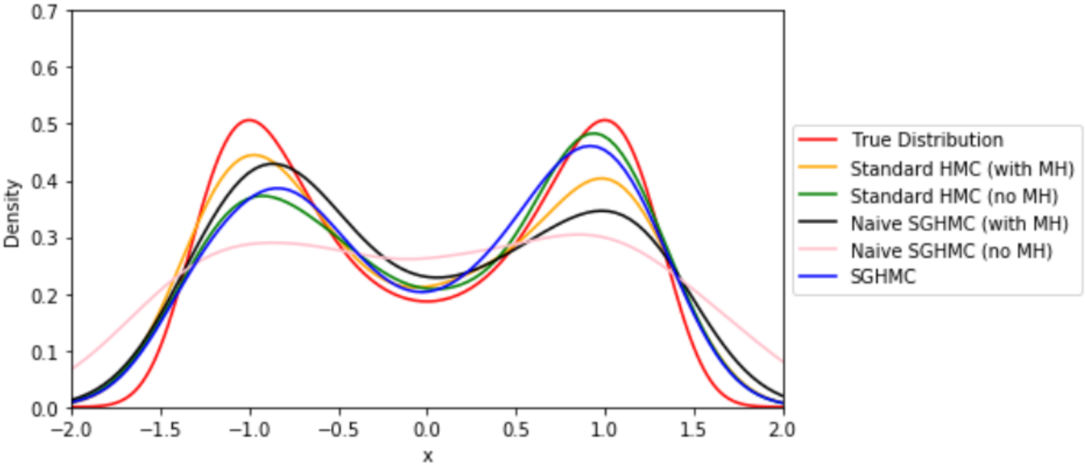


Figure 6: 1-D PDF

4.1.3 2-D Simulation Task

In this last part of simulation, we choose to draw samples from a two-dimensional distribution using only two algorithms: SGHMC and SGLD.

The target bivariate normal distribution has a potential energy function $U(\theta) = \frac{1}{2}\theta^T\Sigma^{-1}\theta$, and the noisy gradient is chosen to be $\Sigma^{-1}\theta + \mathcal{N}(0, I)$. The simulation result is in **Figure 7**.

As is shown, both algorithms generate points that cluster around the high-density area. However, SGLD seems to be less efficient in terms of exploring the entire parameter space and this shows that SGHMC seems to do better than SGLD in this case, though the two does perform similarly.

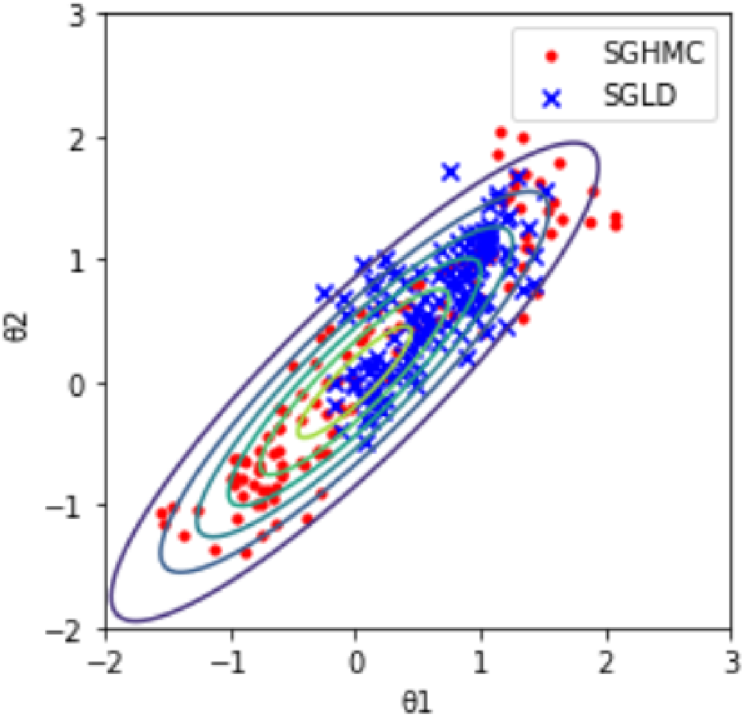


Figure 7: 2-D PDF

4.2 Application on Real World Datasets

In this section, we focus on comparing the following three algorithms: 1. a Stochastic Gradient Descent algorithm (SGD) with momentum; 2. a Stochastic Gradient Langevin dynamics algorithm (SGLD); 3. a Stochastic Gradient Hamiltonian Monte Carlo (SGHMC)

These three algorithms are applied on two real-world datasets, focusing on a logistic regression and a linear regression respectively, based on a Bayesian paradigm. This time, we mainly compare the accuracy of prediction on the test set to evaluate their performance.

4.2.1 Logistirc Regression based on Bayesian paradigm

In this example, we focus on a problem predicting whether a room is occupied based on environmental measures such as temperature, humidity, CO_2 and related measures [2]. The dataset is split into a training set of 5700 observations and 6 predictors, and a test set of 4149 observations.

In this problem, the prior distribution of β is chosen as

$$\beta \sim N(0, I_p)$$

The distribution of a single point y_i given x_i and β is

$$y_i \mid x_i, \beta \sim binary(\frac{e^{x_i^T \beta}}{1 + e^{x_i^T \beta}})$$

Therefore, the posterior distribution of β given a data set can be shown as

$$\begin{aligned} p(\beta \mid X, y) &\propto \prod_{i=1}^{n} p(y_i \mid X, \beta) p(\beta) \\ &\propto \prod_{i=1}^{n} (\frac{e^{x_i^T \beta}}{1 + e^{x_i^T \beta}})^{y_i} (\frac{1}{1 + e^{x_i^T \beta}})^{1 - y_i} e^{-\frac{1}{2}\beta^T I_p^{-1} \beta} \\ &= e^{\sum_{i=1}^{n}[y_i x_i^T \beta - log(1 + e^{x_i^T \beta})] - \frac{1}{2}\beta^T \beta} \\ &= e^{y^T X \beta - \sum_{i=1}^{n} log(1 + e^{x_i^T \beta}) - \frac{1}{2}\beta^T \beta} \end{aligned}$$

With this in mind, we can easily acquire the gradient by noticing

$$\nabla U(\beta) = -\nabla_\beta \left[y^T X \beta - \sum_{i=1}^{n} log(1 + e^{x_i^T \beta}) - \frac{1}{2}\beta^T \beta \right]$$

For stochastic gradient, we need to adjust it by using only a fraction of data points. And then we can draw β from its posterior distribution with the three different dynamics, and calculate test errors using β along its iteration.

The resulted test errors are shown below in **Figure 8**.

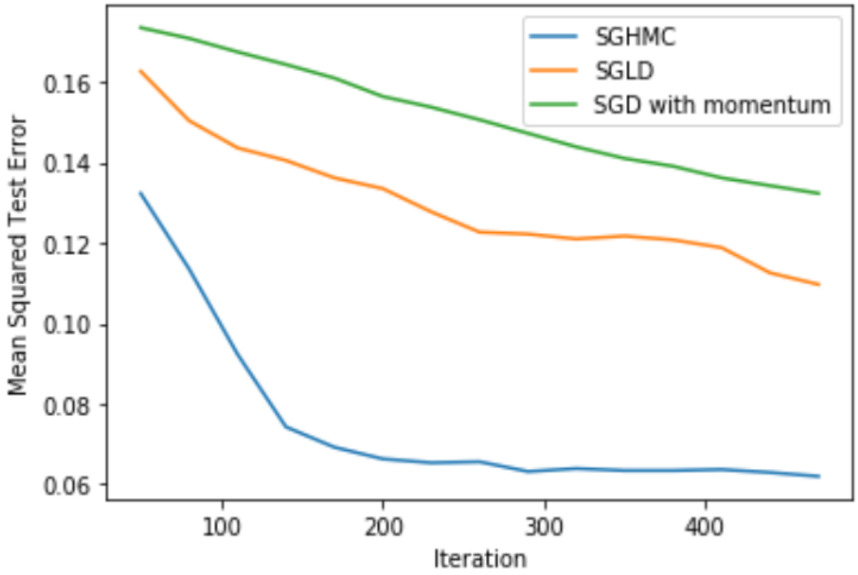


Figure 8: Logistic Regression

In the implementation, the step size is chosen as 0.001 for all of the algorithms for comparison. We can see that SGHMC performs the best in terms of the speed of convergence, which leads to a fast-decreasing test error rate.

4.2.2 Linear Regression based on Bayesian paradigm

In the second part, we are going to employ the simulation algorithms into a linear regression problem based on Bayesian paradigm.

In this example, the prior distribution of β is set as

$$\beta \sim N(0, I_p)$$

Since the data set has been scaled, the response variable y has mean 0 and variance 1. Thus the distribution of y given β is

$$y \sim N(X\beta, I_n)$$

Therefore, the posterior distribution of β given a data set can be shown as

$$\begin{aligned} p(\beta \mid X, y) &\propto p(y \mid \beta, X) p(\beta) \\ &\propto e^{-\frac{1}{2}(y-X\beta)^T I_n^{-1}(y-X\beta)} e^{-\frac{1}{2}\beta^T I_p^{-1}\beta} \\ &= e^{-\frac{1}{2}(y-X\beta)^T(y-X\beta)-\frac{1}{2}\beta^T\beta} \end{aligned}$$

With a step size chosen as 0.001 again, the resulted test error rate against the number of iterations are shown in **Figure 9**. Once again, we can see that with the same size of step, SGHMC converges with the fastest speed to the stationary distribution.

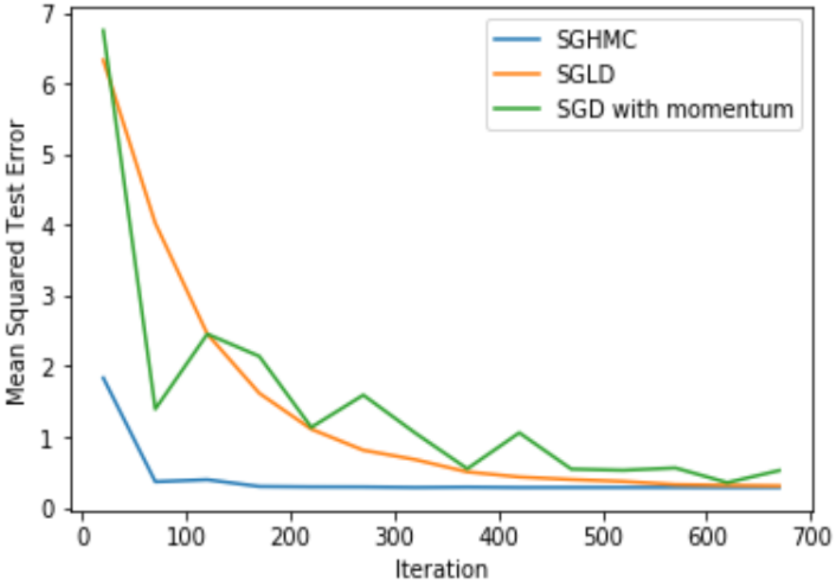


Figure 9: Linear Regression

5 Discussion

In this paper we derived and implemented the Stochastic Gradient Hamiltonian Monte Carlo algorithm. Several optimization methods were then employed to increase the efficiency of the code. Three simulation tasks and two real-world data analysis examples were carried out to present the performance of our optimized algorithm, and to compare the difference between SGHMC and a series of similar algorithms.

As we stated at the beginning, simulation is a widely-needed technique. However, in today's era of big data, it might be difficult to implement standard HMC simulation, which is a popular MCMC method. Thus, SGHMC, by introducing the concept of stochastic gradient and a counter-acting friction term, successfully decrease the computation expenses while maintaining a reasonable accuracy relative to the true distribution.

Supportive as we are, there are still some limitations of SGHMC. The most noticeable one is the difficulty of tuning parameters. Since extra noise and friction terms are introduced, the burden of parameter tuning can be very frustrating. One way to mitigate it is to parameterize the algorithm in a way that resembles Stochastic Gradient Descent algorithm so that experience of setting parameters for SGD could be borrowed [1]. Another limitation is that it is hard to obtain a satisfactory estimate of B, which is relevant to the noise term brought in by stochastic gradient. While relying on small steps can, to some extent, ease the problem, future research on how to acquire an estimate accurately and efficiently does need to be carried out to better address it.

References

[1] Chen Tianqi, Emily Fox, and Carlos Guestrin. Stochastic gradient hamiltonian monte carlo., 2014.

[2] Luis M. Candanedo and Veronique Feldheim. Accurate occupancy detection of an office room from light, temperature, humidity and co2 measurements using statistical learning models., 2016.

[3] M. Choi. Medical cost personal datasets., 2018.

[4] Max Welling and Yee W. Teh. Bayesian learning via stochastic gradient langevin dynamics., 2011.

Appendix

Appendix A: Basis of HMC

In physics, Hamiltonian dynamcis describes the motion of particles in a system based on its location (θ) and momentum (r) at some time t.

In terms of different locations, the particle would have potential energy, $U(\theta)$; in terms of its momentum, it would have different kinetic energy $K(r)$; the total energy in this system is thus called Hamiltonian, which can be mathematically stated as

$$H(\theta, r) = U(\theta) + K(r)$$

In a Hamiltonian system, this total energy is conserved.

To describe how potential energy and kinetic energy are converted into each other when the particle moves in the system in time, we have the Hamiltonian equations to descibe the dynamics:

$$\frac{\partial \theta_i}{\partial t} = \frac{\partial K(r)}{\partial r_i}$$
$$\frac{\partial r_i}{\partial t} = -\frac{\partial U(\theta)}{\partial \theta_i}$$

As is shown, these equations are derived on a continuous time scale. In order to simulate it on computer, some finite difference methods need to be carried out to discretize it. Leapfrog is a well-accepted method, since it's relatively easy to calculate and has a good performance.

With Leapfrog method and the above equations, we would then represent the Hamiltonian equations for a discrete time step δ:

$$r_i(t + \frac{\delta}{2}) = r_i(t) - \frac{\delta}{2} * \frac{\partial U(\theta)}{\partial \theta_i(t)}$$
$$\theta_i(t + \delta) = \theta_i(t) + \delta * \frac{\partial K(r)}{\partial r_i(t + \frac{\delta}{2})}$$
$$r_i(t + \delta) = r_i(t + \frac{\delta}{2}) - \frac{\delta}{2} * \frac{\partial U(\theta)}{\partial \theta_i(t + \delta)}$$

Now, in order to set up a connection between the Hamiltonian dynamics and our task which is to simulate from a distribution, consider the canoncial distribution for some energy function $E(\theta)$, which is $p(\theta) = \frac{1}{Z}e^{-E(\theta)}$, where Z is just a normalizing constant ensuring the pdf integrate into 1.

With this in mind, we would have the canoncial distributions for our Hamiltonian system as:

$$\begin{aligned} p(\theta, r) &\propto e^{-H(\theta,r)} \\ &\propto e^{-(U(\theta)+K(r))} \\ &\propto e^{-U(\theta)} * e^{-K(r)} \\ &\propto p(\theta)p(r) \end{aligned}$$

This pdf factors into the canoncial distributions of the location and momentum respectively. This means the two variables, θ and r, are independent, and therefore we can use the Hamiltonian dynamics to simulate the joint distribution of θ and r, and then consider only the part for θ to get its marginal distribution.

In order to exploit the Hamiltonian system, we can set the potential energy function $U(\theta) = -\log(p(\theta))$, the negative logarithmic form of our target distribution.

To simulate data points, we can select an initial random momentum r for a particle, numerically integrate using the Leapfrog method and then use the updated θ' as the new proposal. Notice that since our finite difference method is not perfect in maintaining energy conservation, like the traditional MH approach, we accept the proposal with probability

$$\frac{e^{-U(\theta')-K(r')}}{e^{-U(\theta)-K(r)}} = e^{U(\theta)-U(\theta')+K(r)-K(r')}$$

Appendix B: Github link for this paper

The source code of our package, together with the codes used to generate the plots in this paper and the test code can be found at either one of the following Github repos:

`https://github.com/Mingxuan-Yang/SGHMC`

`https://github.com/JC86JC/SGHMC`

Appendix C: installation guide

This package can be viewed on TestPyPi:

`https://test.pypi.org/project/sghmc-pkg-663-2.0.0/2.0.0`.

The preferred installation is through `pip`:

```
pip install -i https://test.pypi.org/simple/ sghmc-pkg-663-2.0.0==2.0.0
```