# Implementation of Stochastic Gradient Hamiltonian Monte Carlo

*Qianyin Lu, Zewen Zhang*

*27 April, 2020*

## Abstract

Hamiltonian Monte Carlo is a variant of Markov chain Monte Carlos (MCMC) that uses the gradient to scale more efficiently to higer dimensions. Stochastic Gradient HMC with a friction term introduced by the author provided an efficient way to estimate the gradient information without using the entire dataset and prevented the negative influence caused by the noise. This paper focuses on the explanation and implementation of SGHMC algorithm to logistic models with simulated and real data. We will use the simulated data to show the differences of Hamilton Dynamics Sampling under different scenarios. We will also demonstrate the optimization of this algorithm using python3. In addition, two other algorithms, Stochastic gradient Langevin dynamics (SGLD) and Stochastic Gradient Descent (SGD) will be introduced and we will compare the SGHMC algorithm with these two methods in terms of the prediction accuracy. Lastly, we will apply the real data into a logistic model and compare the method of Maximum Likelihood estimation and SGHMC.

## Background

Hamiltonian Monte Carlo (HMC) is a Markov chain Monte Carlos sampling method that takes use of derivatives of the density function to provide an efficient way that enables distant proposals. One advantage of such dynamics is that the target distribution is invariant under such system. It is also able to generate highly efficient samplings from the correlated distributions because of the use of momentum term. With the Metropolis-Hastings correction process to correct for its discretization of continuous system, the high acceptance probability is achieved. However, a limitation of such system is the requirement to calculate the gradient of the potential energy. It is extremely inefficient and even impossible to find out the gradient of the entire dataset with the incredibly huge data amount nowadays.

Thus, to discover a way that better implement the HMC, some algorithms were proposed. Most of these algorithms use mini batch to get a subset sampling and estimated the gradient with mini-batched dataset. An intuitive way was to update the momentum term with noise included: $\Delta r = -\epsilon\nabla\tilde{U}(\theta) = -\epsilon\nabla U(\theta) + N(0, \epsilon^2 V)$ (Chen, Fox&Guestrin, 2014). However, adding noise to the dynamics increases entrophy and thus $\pi(\theta, r)$ is no longer invariant. Thus, a MH correction step is needed.

As a result, the MH step introduces a difficult tradeoff to the HMC dynamics. On the one hand, it is extremely costly to run MH for all datapoints. On the other hand, lengthy simulations lead to low accceptance rate. Therefore, the author proposed a new method to offset the negative effect of using stochastic gradients by introducing a "friction" term to the momentum update (Chen, Fox&Guestrin, 2014).

## Description of Algorithm

To better understand the SGHMC algorithm, it is essential to first explore the basic Hamiltonian Monte Carlo method. The HMC borrows the idea from physics and introduces a potential energy function when sampling from the target distribution. Typically, the target distribution is set to be the posterior distribution of the interest parameter $\theta$ given a set of observations D. A common way to express the distribution with potential energy function U is shown below:

$$p(\theta|\mathcal{D}) \propto exp(-U(\theta))$$

where $\theta$ is the parameter that we are interested in, $\mathcal{D}$ is the data and U is the potential function defined by:

$$U = -\sum_{x \in \mathcal{D}} log(p(\theta|D) - logp(\theta)$$

Mentioned in the first part, HMC system has the benefits over other algorithms in terms of sampling from correlated distributions and defining distant proposals due to the "auxiliary" variable - the momentum term r (Chen, Fox&Guestrin, 2014). Hence, the mechanism becomes taking the marginal distribution of $\theta$ from the joint distribution of $\theta$ and r: $\pi(\theta, r) \propto exp(-U(\theta) - 0.5r^T M^{-1} r)$. Taking the derivative with respect to $\theta$ and r, we got our sampling process for the HMC method:

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

Due the effect of changing from continuous to discrete system, additional step of MH correction is needed. Then, pseudo code from figure 1 from author illustrates how the algorithm work.

**Input**: Starting position $\theta^{(1)}$ and step size $\epsilon$
**for** $t = 1, 2 \cdots$ **do**
  *Resample momentum* $r$
  $r^{(t)} \sim \mathcal{N}(0, M)$
  $(\theta_0, r_0) = (\theta^{(t)}, r^{(t)})$
  *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**
    $\theta_i \leftarrow \theta_{i-1} + \epsilon M^{-1} r_{i-1}$
    $r_i \leftarrow r_{i-1} - \epsilon \nabla U(\theta_i)$
  **end**
  $r_m \leftarrow r_m - \frac{\epsilon}{2} \nabla U(\theta_m)$
  $(\hat{\theta}, \hat{r}) = (\theta_m, r_m)$
  *Metropolis-Hastings correction*:
  $u \sim \text{Uniform}[0, 1]$
  $\rho = e^{H(\hat{\theta}, \hat{r}) - H(\theta^{(t)}, r^{(t)})}$
  **if** $u < \min(1, \rho)$, **then** $\theta^{(t+1)} = \hat{\theta}$
**end**

Figure 1: HMC Algorithm

However, due to the scalability of the data, instead of directly computing the gradient $\nabla U(\theta)$ using the entire dataset $\mathcal{D}$, we consider a noisy estimate based on a minibatch $\tilde{\mathcal{D}}$ sampled uniformly at random from the entire dataset $\mathcal{D}$ to compute:

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

SGHMC algorithm proposed by the author offers a brilliant way of replacing the gradient of U with stochastic gradients from mini-batched data. Not only included the noise term, the author also introduced a friction term $-BM^{-1}$, where $B = 0.5\epsilon V(\theta)$. Since the noise increases the total energy, the friction term here then decreases the potential energy. Therefore, similarly as HMC method, after taking the derivative with respect to $\theta$ and r, the updated sampling algorithm becomes:

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

Notice that this system also reflect the second-order Langevin dynamics in practical use and this is how we generate the SGLD system by ourselves later in the comparison part. As the friction term becomes larger,Langevin dynamics reduces to first-order. Then, according to the sampling mechanism above, we got the final algorithm for the SGHMC with a friction term as below:

**Algorithm 2:** Stochastic Gradient HMC

**for** $t = 1, 2 \cdots$ **do**
  *optionally, resample momentum* $r$ *as*
  $r^{(t)} \sim \mathcal{N}(0, M)$
  $(\theta_0, r_0) = (\theta^{(t)}, r^{(t)})$
  *simulate dynamics in Eq.(13):*
  **for** $i = 1$ ***to*** $m$ **do**
    $\theta_i \leftarrow \theta_{i-1} + \epsilon_t M^{-1} r_{i-1}$
    $r_i \leftarrow r_{i-1} - \epsilon_t \nabla \tilde{U}(\theta_i) - \epsilon_t C M^{-1} r_{i-1}$
      $+\mathcal{N}(0, 2(C - \hat{B})\epsilon_t)$
  **end**
  $(\theta^{(t+1)}, r^{(t+1)}) = (\theta_m, r_m)$, no M-H step
**end**

Figure 2: SGHMC Algorithm

## Optimization for Performance

Since there are only two loops in our algorithm function, and it does not require M-H steps, our function is already fast to run even under simple python. In additional to that, the vectorization is already applied in the original algorithm, there was no obvious difference when we applied vectorization to our algorithm.

Due to the fact that our function calls two sub-functions written by ourselves, as well as the reason that some build-in NumPy distribution functions will be need to be hand-written, we have some restrictions if we try to optimize it into C++, also it will consume much more time to complete the code compared to other optimization methods. Therefore, in our case we just optimize with our function with numba.jit.

There mainly two parts in our optimization process:

1. We took the process of choosing minibatch of the whole data out of the main function and wrote it into a function.
2. Jit compilation of the main function: sghmc sampling algorithm as main optimization strategy.

We compared three versions of our algorithm function to see the difference of runtime under simulated data, and the same gradient function as in the original algorithm in the paper. The simulated data has 20000 observations and 5 variables. We tested the algorithm with 30 iterations in the first loop and updated the momentum 200 times in the outside loop for each test.

case1: Regular python with batch function outside the algorithm function

7.43 s ± 101ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

case2: Jit compilation of algorithm function with batch function outside

5.35 s ± 341 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Even though we decided to write a simple function for minibatch for efficiency, we were still curious about the result if we do not put the minibatch function outside the main function:

case3: Jit compilation of algorithm function with batch function inside

9.37 s ± 891 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In the comparison between three cases, we failed to notice huge difference, which might be caused by the simple structure of the main function, leaving not much space for improvement, indicating that all versions of programs running at similar speed, and are all efficiently fast for our test. However, we can still notice that the numba improved function with batch function outside the main algorithm has the best performance.

## Applications to simulated data

In the simulated scenarios, we'd like to compare the behavior of HMC with performance of SGHMC as setting SGHMC as a relatively upgraded version of HMC using the same gradient. The stochastic gradient is noisy because we can only give an approximation of the gradient. And as demonstrated in the paper, a friction can be added to the momentum to minimize the effect of the injected noise on the Hamilton dynamics.

To construct a better and clear understanding of the difference in performance of Hamilton dynamics with/without friction and resampling, we followed the example and re-built the figure2 in the original paper. We simulated from various Hamiltonian dynamics over 15000 steps. We used the same $U(\theta) = \frac{1}{2}\theta^2$ and $\epsilon = 0.1$.

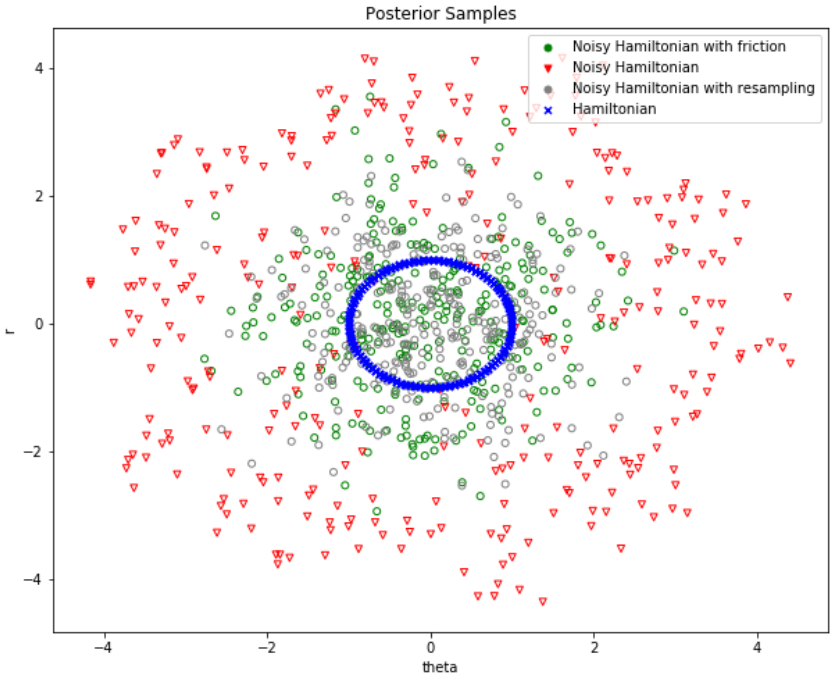


Figure 3: posterior samples

From the figure, we can notice that the Hamilton simulation is the closest to center (0), since the energy is most conserved in this case. Cases using Hamilton with noise has more discrete distribution, especially for

Noisy Hamilton without resampling and friction, which is far away from the center. Noisy Hamilton with friction helps mitigate discrete result by adding friction.

## Applications to real data sets

Before diving into the introduction of data, it is necessary to first illustrates how the SGHMC algorithm is applied to a logistic model. Logistic regression assigns the probability of success to a dichotomous response variable.

$$Pr(y_i = 1|\boldsymbol{x_i}, \boldsymbol{\theta}) = \frac{\exp\{\boldsymbol{x}_i^T\boldsymbol{\theta}\}}{1 + \exp\{\boldsymbol{x}_i^T\boldsymbol{\theta}\}}$$

where $\boldsymbol{x}_i$ is a vector of length $p$ covariates for data point $i$ and $\boldsymbol{\theta}$ is a vector of regression coefficients of length $p$. In a Bayesian framework, we would assign the a prior distribution on our unknown parameters $P(\boldsymbol{\theta}) \sim \mathcal{N}(0, V)$ where V is the variance matrix which is known.

The final potential energy function should be:

$$\nabla U(\theta) = log(p(\theta|D) + log(p(\theta))$$

$$log(p(\theta|D) = ylog(p) + (1-y)log(1-p) = ylog(p/(1-p)) + log(1-p)$$

$$\frac{\partial log(p(\theta|D)}{\partial \theta} = yX - X\frac{exp(X^T\theta)}{1 + exp(X^T\theta)}$$

$$\frac{\partial log(p(\theta))}{\partial \theta} = -V^{-1}\theta$$

Then, the estimated gradient using mini-batched data is:

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

Then, We applied the algorithm to a dataset that used to predict Heartdisease. This dataset is from an ongoing cardiovascular study in Framingham and is published on the Kaggle website. The dependent variable $y$ used in this data is whether the patient has 10-year risk of future coronary heart disease (CHD). This variable is denoted as TenYearCHD and documented as 1 if the patient have such risk and 0 otherwise. There are 15 independent variables available in total. However, we decided to use only six of them for the simplicity of calculation. We chose age, male, cigsPerDay, totChol, sysBP and glucose as predictors intuitively. The detailed description of these variables can be seen from the kaggle website.

Since TenYearCHD is a binary variable, it makes sense to build a logistic regression model. A logistic model can be written as following:

$$Y = logit(p) = log(\frac{p}{1-p}) = \beta X$$

$$p = \frac{exp(X^T\beta)}{1 + exp(X^T\beta)}$$

We immediately realized that the beta coefficients got from the MLE are actually the estimated unknown parameters $\theta$ that we achieved from the SGHMC algorithm. Thus,for the analysis part, we decided to compare the result of SGHMC with regular logistic model in terms of its prediction accuracy and the values of beta coefficients.

Before doing the actual analysis, we first did some data preparation. We realized there are quite a few missing values for some variables. Because we only have approximately 4000 datapoints and about 1/8 of these data have missing values, it is not wise to remove all these points. Thus, we imputed the missing data with a median of the non-missing points. We also separate the data into training(80%) and test(20%) parts.

Then, we compared the result of SGHMC with the logistic regression model provided by python package. Since there aren't many data points and we only ran 1000 iterations for SGHMC, we just used the alogrithm that wasn't optimized from our own package.

For prediction accuracy, we built a helper function called test error, which will also be used later in our comparison part. This package manually calculated the test error of SGHMC algorithm by counting how many predictions generated from SGHMC matched with the actual test result for each iteration. We then applied the training data to both algorithms and got the following result:

Logistic Regression Model: 0.827

SGHMC: 0.76

From this, we can see that logistic regression model with python's own package seems doing a better job than the SGHMC model. A possible reason could be that the iteration number of SGHMC is not large enough. However, if we ran too many iterations, a tradeoff of computational efficiency will occur because the logistic regression model package obviously ran much faster. Another reason could be that we kept all parameters in logistic regression package as default, so some parameters are optimized automatically to achieve the best result.

In addition, we also compared the estimation of the unknown parameters.In order to make the result unchanged with the influence of scale, we first normalized the predictors of the training and test data. From the following figure, it is easy to observe that the result from SGHMC and logistic regression model are very similar even with such small number of iterations. This means the samplings from the posterior distribution for the interest parameter from a SGHMC algorithm and a Maximum likelihood estimation are quite close to each other. Since the MLE a relatively authentic way to calculate the $\beta$s, we can conclude that the $\theta$s sampled using SGHMC algorithm are also very accurate.

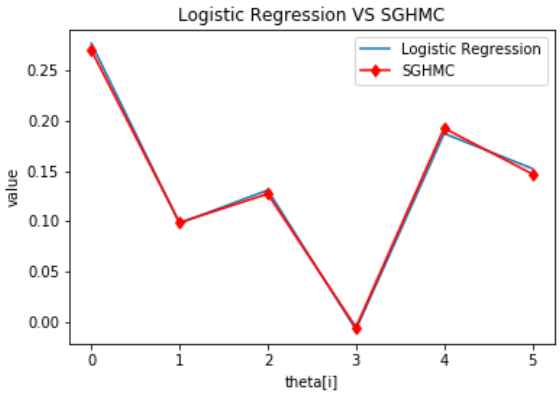


Figure 4: Logistic Regression VS SGHMC

## Comparative analysis with competing algorithms

For this part, we tried to replicate the result from author's figure 4. Instead of using the MNIST data, we simply generated a set of data by ourselves. Because the MNIST data also has its $Y$ as binary, we decided on a binomial distribution. We set a true theta value and use that to generate our dependent variable. Similarly, the predictors are normalized. Then, we chose three algorithms: SGLD, SGD and SGHMC. Since

the existing packages of these algorithms are either hard to find or having some complicated and additional optimizations included, we just generated a simplified version of them according to author's descriptions. Different from the author, we decided to test for the performance of 2000 iterations for each algorithm because we have fewer data points. We used the same way to calculate the test error from the last part. Due to the relative large data and iteration number, we used the optimized SGHMC algorithm in this part.

The simulation result is shown below. To see a zoom-in version of each algorithm's result, you can open the notebook from the comparison folder. Although at the very beginning of the iteration process, SGLD has a relative smaller test error, such difference is trivial and SGHMC becomes the bottom one soon at about 80th iteration. This could be caused by the fact that we did not include a burn-in process. In general, SGHMC always performs better than the other two algorithms, SGLD is the second best one and SGD is the last one. In addition, with the increasing number of iterations, the test error tend to converge. Such result is consistent with author's and we can imply that with the number of iterations increases, the test errors of all three algorithms are getting smaller and closer to each other.

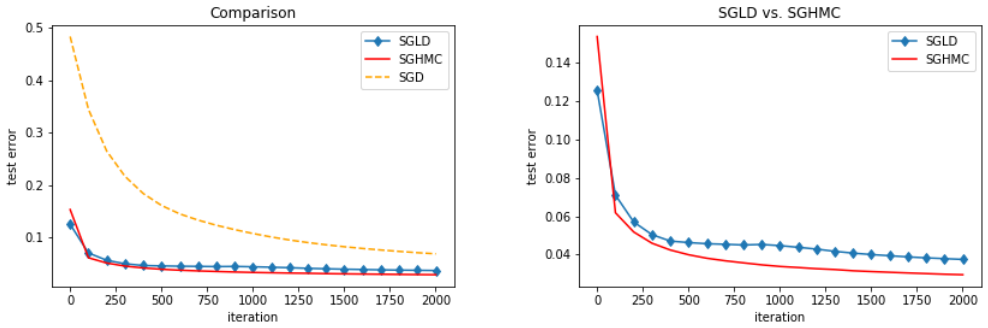


Figure 5: Comparison

## Discussion

In conclusion, the SGHMC algorithm has relatively lower prediction test error compared to SGLD and SGD. It also did a good job in sampling thetas from the posterior distribution even comparing to the "authentic" logistic regression model with python's own package. As the iteration number increases, the estimation becomes more accurate and the test errors become smaller and smaller. However, although we did not run an official %timeit to measure how long each algorithm ran, we still realized the obvious time difference between SGHMC with the other algorithms. To be more specific, the optimized version of SGHMC in python3 ran the slowest among all (SGD, SGLD and MLE). One possible reason for this scenario is that SGHMC has two iterated loops and python doesn't do well in running loops obviously. Since we had a inner and outer loop, it is hard to do any additional vectorization and the computation process is too complex. Compare to other single-loop alogrithms with m iterations, SGHMC needs to instead run $n^m$ times. Therefore, if we have enough time, changing the SGHMC algorithm to a C++ version will definitely makes it run much faster.

## References

Chen, T., Fox, E., and Guestrin, C. *Stochastic Gradient Hamiltonian Monte Carlo.* ArXiv e-prints. 1402.4102.

Neal, R.M. MCMC using Hamiltonian dynamics.*Handbook of Markov Chain Monte Carlo* , 54:113–162, 2010.

Dileep *Logistic regression To predict heart disease.* Kaggle, https://www.kaggle.com/dileep070/heart-disease-prediction-using-logistic-regression, 2019

## Code

All code is available at: https://github.com/QianyinLu/mypackage

**To install the package:**

!pip install –index-url https://test.pypi.org/simple/ presnie
from algorithm import sghmc
sghmc.sghmc() is our algorithm.
OR:
!pip install git+https://github.com/QianyinLu/mypackage
from algorithm import sghmc
sghmc.sghmc() is our algorithm.