Implementation of Stochastic Gradient Hamiltonian Monte Carlo in Python

Weiting Miao[*], Chen An[†], Ryan Fang[‡]

April 27, 2021

Abstract

Stochastic Gradient Hamiltonian Monte Carlo (SGHMC) is a sampling method based on Hamiltonian Monte Carlo (HMC), a method that improves efficiency of general Markov chain Monte Carlo (MCMC) methods by setting up the sampling process in a Hamiltonian dynamics system. SGHMC improves HMC by simplifying the calculation of gradients. Applying stochastic gradient directly to HMC is infeasible since the noise introduced by the stochastic gradient will disturb the Hamiltonian dynamics system which results in the divergence in the sampling process. SGHMC ameliorates this problem by using a second degree Langevin dynamics with a friction term so that the desired target distribution remains stationary. SGHMC also has a small computation complexity. We implement SGHMC in Python as well as several competing algorithms, HMC, naive SGHMC, SGD, and SGLD, to show how good the performance of SGHMC is. We provide optimization of SGHMC using vectorization, cython, nJIT using numba, and a combination of these techniques. The optimization results in 7 times faster SGHMC algorithm with gradient provided and 10% increase in efficiency of SGHMC algorithm without gradient algorithm provided. Finally, we provide application of SGHMC and comparative analysis among SGHMC, HMC, and naive SGHMC using simulated experiments (including Hamiltonian Dynamics, 1-D simulation, 2-D simulation, and Bayesian Linear Regression) and real world datasets (including the MNIST dataset for recognizing digits and the Buenos Aires Airbnb prices dataset).

Keywords: Hamiltonian Monte Carlo, Stochastic Gradient Hamiltonian Monte Carlo, Markov Chain Monte Carlo, Bayesian Learning, Posterior Distribution

[*]Department of Economics, Duke University. Email: weiting.miao@duke.edu.
[†]Department of Mathematics, Duke University. Email: chen.an@duke.edu.
[‡]Department of Statistics, Duke University. Email: ryan.fang@duke.edu.

1 Introduction

In this section, we briefly state the ideas of Stochastic Gradient Hamiltonian Monte Carlo (SGHMC) sampling method. For more details, including the formulas, pseudocode, and mathematical foundation of the method and all the relative algorithms, see Section 2.

We follow Chen et al. (2014) and implement SGHMC sampling method in the paper. SGHMC is based on the Hamiltonian Monte Carlo (HMC) sampling method, developed in Duane et al. (1987), so we first illustrate HMC. To sample from a posterior distribution, we associate the distribution function to a potential energy function U. The idea of HMC is to make use of the fact that the total energy is preserved in a Hamiltonian dynamics. Inspired by this fact, we can generate a sequence of values that approximate the posterior distribution.

When the dataset is large, it is unfeasible to implement HMC directly, as we have to compute the gradient of U in the process, which involves heavy computational complexity. Therefore, it is natural to consider a stochastic gradient version of the algorithm. It is unfortunate, however, that the naive stochastic gradient version of HMC has the deficiency that a noise term is introduced in the discrete process, changing the total energy in the Hamiltonian dynamics. Thus, a Metropolis-Hastings (MH) correction step is necessary. A disadvantage for using the MH step is again pointing to the heavy computational complexity.

SGHMC is a sampling method that addresses the issue of the naive stochastic gradient version of HMC. The key point is that a friction term is introduced during the sampling process to control the effect from the noise term in the naive stochastic gradient version of HMC. Therefore, the MH step is not necessary in SGHMC. As a result, the accuracy of approximation of the posterior distribution is high and the computational complexity is low. These are the major advantages for SGHMC. Moreover, it turns out that SGHMC generates well-approximated posterior distributions even if data are highly correlated. Thanks to these advantages, the algorithm can be widely applied to any cases that require an MCMC method. It can be used in various fields, including applied math, economics, biostatistics, and other areas, to tackle with complicated models like Bayesian learning models. In Chen et al. (2014), the authors provided applications on a few simulated datasets and real world datasets. The result was that SGHMC gives rise to smaller test errors and smaller RMSEs. In our paper, we provide application of SGHMC on both simulated experiments (including Hamiltonian Dynamics, 1-D simulation, 2-D simulation, and Bayesian Linear Regression) and real world datasets (including the MNIST dataset for recognizing digits and the Buenos Aires Airbnb prices dataset). We also see good performances using SGHMC.

Despite the advantages, SGHMC may be time-consuming during the process of computing certain parameters. Also, if the parameters are chosen inappropriately, the result will not be satisfactory. Still, SGHMC is a highly competitive algorithm when one tries to use an MCMC method based on its advantages discussed above. When one needs to sample from a posterior distribution in research or in real world application, SGHMC can be directly applied once we obtain data and set up prior distribution and likelihood function.

2 Background and Algorithm

The standard HMC algorithm requires accurate gradient estimates which rely on the entire dataset when the desired distribution is a posterior distribution. To reduce the computational burden, stochastic gradient HMC uses noisy gradient estimates based on a randomly sampled minibatch instead. However, the naive stochastic gradient HMC approach would introduce noise in the momentum update and make Hamiltonian function be time variant, which leads to lower acceptance rates and thereby requires more iterations. To alleviate the issue, Chen et al. (2014) propose an algorithm of stochastic gradient HMC with friction (SGHMC). The main idea of SGHMC is to add a "friction" term to the momentum update to reduce the influence of the noise. They prove that distribution evolution under this dynamic system is stationary with a certain friction form. The problem is we do not know the noise model in practice. Their solution is to introduce a user specified friction term that is greater than the noise model in the sense of positive semi-definiteness.

Intuitively, we could image that the Hamiltonian Monte Carlo algorithm is like sampling the distance between the earth and the sun while the earth rotates around the sun on a fixed plane. The sun and earth forms a Hamiltonian dynamics system in the sense that the total energy of the system, which is the sum of the potential energy between the sun and the earth and the kinetic energy of earth if we consider the sun as static in the universe, remains constant. But if we apply stochastic gradient algorithm directly on the HMC algorithm, it creates a stochastic noise just like there is wind in the universe continuously blowing the earth, which is rotating around the sun on a two-dimensional plane. The noise gradients introduced to the system strictly increases the total entropy of the system by some reasonable assumptions, resulting in the divergence of the sampling process of the target parameter. The SGHMC algorithm proposed by Chen et al solves this problem by adding a friction term. More specifically, they introduce the friction term using the second order Langevin dynamics. It is like the plane in which the earth rotates around the sun has frictions that counter-balance the effect of wind on increasing the system entropy. Thus, in the long run, the distribution of the target parameter remains stationary, which is the reason that SGHMC is more robust and efficient than the naive stochastic gradient HMC.

Also, following Chen et al. (2014), we can image the original Hamiltonian dynamics system as a hockey puck sliding over a frictionless ice surface of varying heights. The height of the current puck position determines the potential energy, and the momentum of the puck captures the kinetic energy. The total energy in this conservative system remains constant over time. When it comes to naive stochastic gradient HMC formulation, the puck is on the same ice surface but with random wind blowing, which makes the system dissipative. In the case of SGHMC, the hockey puck is on a surface with friction. Although the wind is still blowing, the fraction from the surface prevents the puck moving around. This is also referred to as second-order Langevin dynamics.

In the following subsections, we will introduce the details, including the mathematical basis and pseudocode, of each algorithm.

2.1 Standard HMC

HMC is a sampling method in a Metropolis-Hastings (MH) framework that can make use of gradient information to efficiently explores the state spaces. Suppose we want to sample from the distribution of θ with the density function $p(\theta) \propto \exp(-U(\theta))$. HMC uses a set of auxiliary variables r to represent momentum of the particles. The total energy system with position θ and momentum r is defined by

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

where M is a mass matrix and often set to the identity matrix. To sample from $p(\theta)$, HMC considers generating samples from the joint distribution of (θ, r), and we can simply take the marginal distribution of θ as the desired distribution. The joint distribution of (θ, r) is defined by

$$\pi(\theta, r) \propto \exp(-H(\theta, r)). \tag{2}$$

Following the Hamiltonian equations, we have a system of dynamic equations

$$\begin{cases} d\theta = M^{-1} r dt \\ dr = -\nabla U(\theta) dt. \end{cases} \tag{3}$$

In a continuous time setting, the dynamics preserve the total energy. However, we usually work with discretized system in practice where the time-invariant property might not hold. An MH correction step is necessary in this case. With the common discretized approach "leapfrog", the pseudocode of the HMC algorithm is outlined in Algorithm 1.

In a Bayesian learning framework, the posterior distribution is given by

$$\begin{aligned} p(\theta \mid \mathcal{D}) &\propto p(\mathcal{D} \mid \theta) p(\theta) \\ &= \exp(\log p(\mathcal{D} \mid \theta) + \log p(\theta)) \\ &= \exp(\sum_{x \in \mathcal{D}} \log p(x \mid \theta) + \log p(\theta)) \end{aligned}$$

Thus, the potential energy function of the posterior distribution is

$$U(\theta) = -\sum_{x \in \mathcal{D}} \log p(x \mid \theta) - \log p(\theta), \tag{4}$$

where D is the data set, $p(\theta)$ is the prior density, and $p(x\theta)$ is the likelihood of x.

2.2 Naive SGHMC

Since it is very computationally expensive to directly calculate the gradient $\nabla U(\theta)$ for large data sets, SGHMC addresses the problem by using a minibatch of the data to derive the

Algorithm 1: Hamiltonian Monte Carlo

Result: A sample from the desired distribution
Given the starting position $\theta^{(1)}$ and step size ϵ;
for $t = 1, 2 \ldots$ **do**
  Optionally resample momentum r:
  $r^{(t)} \sim \mathcal{N}(0, M)$
  $(\theta_0, r_0) = (\theta^{(t)}, r^{(t)})$
  Simulate discretization of Hamiltonian dynamics:
  $r_0 \leftarrow r_0 - \frac{\epsilon}{2}\nabla U(\theta_0)$ **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:
  $u \sim \text{Uniform}[0, 1]$
  $\rho = \exp(H(\hat{\theta}, \hat{r}) - H(\theta^{(t)}, r^{(t)}))$
  if $u < \min(1, \rho)$ **then**
    $\theta^{(t+1)} = \hat{\theta}$
  end
end

gradient estimates:

$$\nabla \tilde{U}(\theta) = -\frac{|\mathcal{D}|}{|\tilde{\mathcal{D}}|} \sum_{x \in \tilde{\mathcal{D}}} \nabla \log p(x \mid \theta) - \nabla \log p(\theta), \tilde{\mathcal{D}} \subset \mathcal{D}. \tag{5}$$

If observations are independent, the noisy gradient is approximated as

$$\nabla \tilde{U}(\theta) \approx \nabla U(\theta) + \mathcal{N}(0, V(\theta)), \tag{6}$$

where V is the covariance matrix of the stochastic gradient noise.In practice, we can use empirical Fisher information as its estimator(Ahn et al., 2012).

The naive implementation of SGHMC is simply to replace the accurate gradient by this noisy gradient. The corresponding dynamic system becomes

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

where $B(\theta) = \frac{\epsilon}{2} V(\theta)$.

The main differences between the algorithm of naive SGHMC and HMC are:

1. Naive SGHMC uses noisy gradient estimates based on a subset of the data set;

2. In the updating process of momentum variables r, there is an additional noisy term with variance $2B(\theta)dt$.

With the gradient noise, the system is no longer time invariant. Users of naive SGHMC will face the tradeoff of computation complexity and efficiency. On the one hand, introducing an MH correction step after a short simulation runs is computationally costly because these MH steps rely on the entire dataset. On the other hand, if the MH correction is after a long simulation runs, the acceptance ratio could be very low.

2.3 SGHMC

To alleviate the poorly behaved properties caused by the noisy term, we add an fraction term into the original dynamic system, the corresponding updating process becomes:

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

We omit the dependence of B on θ for notation convenience in the remainder of the paper. The introduced friction term $BM^{-1}rdt$ helps reducing the influence of the noise. Chen et al. (2014) shows in their Theorem 3.2 that the target joint distribution under this dynamic system is the unique stationary distribution.

In practice, we do not have a perfect estimate of B. In this case, a user specified friction term $C \succeq \hat{B}$ is beneficial. The dynamics can be summarized by:

$$\begin{cases} d\theta = M^{-1}rdt \\ dr = -\nabla U(\theta)dt - CM^{-1}rdt + \mathcal{N}(0, 2(C - \hat{B})dt) + \mathcal{N}(0, 2Bdt). \end{cases} \tag{9}$$

The corresponding pseudocode is as follows:

Algorithm 2: Stochastic Gradient Hamiltonian Monte Carlo

Result: A sample from the desired distribution
Given the starting position $\theta^{(1)}$ and step size ϵ;
for $t = 1, 2 \ldots$ **do**
  Optionally resample momentum r:
  $r^{(t)} \sim \mathcal{N}(0, M)$
  $(\theta_0, r_0) = (\theta^{(t)}, r^{(t)})$
  Simulate dynamics described above:
  for $i = 1$ *to* m **do**
    $\theta_i \leftarrow \theta_{i-1} + \epsilon M^{-1} r_{i-1}$
    $r_i \leftarrow r_{i-1} - \epsilon_t \nabla \tilde{U}(\theta_i) - \epsilon_t C M^{-1} r_{i-1} + \mathcal{N}\left(0, 2(C - \hat{B})\epsilon_t\right)$;
  end
  $(\theta^{(t+1)}, r^{(t+1)}) = (\theta_m, r_m)$, no M-H step
end

According to our experiments, the choice of C doesn't affect performance much as long as it is greater than $\hat{B}$ in the sense of positive definite. In terms of the choice of $\hat{B}$, the simplest choice is $\hat{B} = 0$ and this is true when ϵ converges to 0.

3 Optimization

We write two python implementations of Stochastic Gradient Hamiltonian Monte Carlo algorithm, named $sghmc\_with\_grad$ and $sghmc\_with\_data$. $sghmc\_with\_grad$ method is used when the gradient function is given to the algorithm, while the $sghmc\_with\_data$ is called when users don't know the gradient function and only could provide data, likelihood function, and a prior to the algorithm.

To optimize both python implementations of the SGHMC algorithm, we first reduce the unnecessary iterations in the for-loop and set up the pseudo parameters for simulation tests. To test the $sghmc\_with\_grad$ function, the true target is one-dimensional and its distribution is given by $U(\theta) = -2\theta^2 + \theta^4$. The mass matrix is set to be one dimensional identity matrix. The learning step size is 0.1 to make sure that the algorithm won't diverge, and the number of total draws is 10000 with 50 steps between two consecutive draws. Finally, the empirical Fisher information matrix, $\hat{V}$ is set to be 4 and the friction term is 3 so that $C \geq \frac{1}{2}\epsilon\hat{V}$.

We use a randomly simulated dataset with 5 features and 10000 observations and employ the function to estimate the 5-dimension Bayesian linear regression parameters to test the speed of $sghmc\_with\_data$. The data is generated from a true linear regression model specified by $y = X^T\theta$ with a standard normal noise. The true θ is a five-dimension vector consisting of integers from 0 to 4. The prior on θ is a standard normal distribution. The mass matrix is again set to identity matrix. The learning step size is 0.0001, and the number of total draws is 10000 with 50 steps between two consecutive draws. Instead of constructing a set of minibatches and then randomly sample one minibatch, we build the minibatch by uniformly sampling observations from the original dataset directly. The minibatch size is 1000. And the matrix for friction term is set to be $13 * I_5$, and the empirical Fisher information matrix, $\hat{V}$ is set to be 0.

Then, we identify the bottlenecks in our Python functions through profiling in the testing environment specified by the parameters shown above for the two algorithm implementations. The profile result is shown in Figure 1. For $sghmc\_with\_grad$ function, the result shows that $np.dot$ and the user specified gradient function take the most amount of time and could be optimized. For $sghmc\_with\_data$ function, it seems that the two gradients of data likelihood functions and prior cost most amount of the time. In the meantime, we also found that our algorithm could not be optimized by parallelization since the next iteration depends on the last one.

To optimize the gradient function, we have tried to use $numba$ nJIT and vectorization to decorate the original input function. Also we have tried to use $numba$ nJIT to optimizing the updating process of parameters of interest and momentum variables, which is one critical step running in for-loops. We also have tried $cython$ optimization of the matrix multiplication, and the $njit$ optimization for the entire parameter update process. Some significant optimization results are shown in the Table 1. Note that since the testing

```
*** Profile stats marshalled to file 'sghmc_with_grad.prof'.
Mon Apr 26 13:20:10 2021    sghmc_with_grad.prof

         6999330 function calls in 8.666 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    3.656    3.656    8.666    8.666 <ipython-input-2-2dedc8ff593a>:2(sghmc_with_grad)
   499950    1.763    0.000    1.763    0.000 <ipython-input-10-aa50056b54bc>:3(<lambda>)
  1999801    1.632    0.000    1.632    0.000 {built-in method numpy.core._multiarray_umath.implement_array_function}
  1999800    0.776    0.000    2.588    0.000 <__array_function__ internals>:2(dot)
   499951    0.659    0.000    0.659    0.000 {method 'randn' of 'numpy.random.mtrand.RandomState' objects}
  1999800    0.180    0.000    0.180    0.000 /usr/local/lib/python3.7/site-packages/numpy/core/multiarray.py:716(do
t)
        1    0.000    0.000    8.666    8.666 {built-in method builtins.exec}
```

(A) SGHMC algorithm with gradient.

```
*** Profile stats marshalled to file 'sghmc3.prof'.
Mon Apr 26 14:15:08 2021    sghmc3.prof

         1500001 function calls (1499996 primitive calls) in 14.105 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   499950    7.379    0.000    7.379    0.000 <ipython-input-114-b807f05dee81>:3(logp_data_grad)
        1    5.635    5.635   14.105   14.105 <ipython-input-115-99b375c79739>:1(sghmc_with_data)
   499950    0.849    0.000    0.849    0.000 {method 'randn' of 'numpy.random.mtrand.RandomState' objects}
   499950    0.237    0.000    0.237    0.000 <ipython-input-114-b807f05dee81>:1(<lambda>)
        1    0.002    0.002    0.002    0.002 {method 'multivariate_normal' of 'numpy.random.mtrand.RandomState' obje
cts}
        1    0.002    0.002    0.002    0.002 {method 'choice' of 'numpy.random.mtrand.RandomState' objects}
        3    0.000    0.000    0.000    0.000 {built-in method numpy.zeros}
```

(B) SGHMC algorithm with data.

FIGURE 1: Profiling results for SGHMC algorithm with gradient and data.

environment is different between the $sghmc\_with\_grad$ and the $sghmc\_with\_data$, there is no significance comparing the running time between those two algorithms.

TABLE 1: Optimization results for both algorithms.

Optimization Method	SGHMC with Gradient	SGHMC with data
Base	7.35s ± 40.8 ms	12.8s ± 258ms
Cython	7.22 s ± 58.3ms	NA
Numba Vectorization	6.15 s ± 52.7ms	NA
Numba nJIT	1.54s ± 95.9ms	11.5s ± 129ms
Numba Vecotorization + nJIT	908ms ± 10.3ms	NA

In Table 1, *numba* vectorization means the vectorization optimization of the gradient functions in each algorithm implementation. *numba* nJIT is the optimization for the entire parameter updating process using nJIT. $Cython$ indicates the optimization for the matrix multiplication and for-loops in the parameter updating process. From the results in Table 1, we could see that $cython$ optimization on matrix multiplication and for-loop results in similar running time with the base function of $sghmc\_with\_grad$. The $njit$ and

vectorization using *numba* on matrix multiplication, for-loop and gradient functions have significantly increased the speed of the algorithm, reducing the running time from about 7.35 to 908 ms, which is 7 times faster than the base function. As for the $sghmc\_with\_data$, the *cython* and *numba* vectorization optimization is not stable, so that there are no values reported. The *numba* nJIT optimization on the parameter updating process results in about 10% increase in speed, which is some improvement but not a great improvement in speed.

4 Applications and Comparative Analysis

4.1 Simulated Experiments

We replicate the simulated examples in the original paper, and also implement the algorithm into a linear regression problem where the data is made up by ourselves so that we can test if the estimates are close enough to the true parameters.[1]

4.1.1 Hamiltonian Dynamics

We consider a potential energy function $U(\theta) = \frac{1}{2}\theta^2$. We apply the algorithms of standard HMC, naive SGHMC without resampling, naive SGHMC with resampling, and SGHMC with friction to sample the momentum term r and the parameter of interest θ. Also, we follow the paper and set the noisy gradient as $\theta + \mathcal{N}(0, 4)$. Figure 2 shows the dynamics of each algorithm. From the plot, we see that noisy Hamiltonian dynamics lead to diverging trajectories. Both resampling r and introducing a friction term help control divergence.

4.1.2 1-D Simulation

In this part, we consider a target distribution with $U(\theta) = -2\theta^2 + \theta^4$. We implement 5 algorithms to draw the density plot: standard HMC (with and without M-H step), naive SGHMC (with and without M-H step), and SGHMC with friction. Figure 3 represents the comparison between the true distribution and empirical distribution associated with various sampling algorithms. All empirical distributions generated by various algorithms except naive SGHMC without M-H correction are close to the true distribution. Comparing with naive SGHMC with M-H correction, the proposed SGHMC performs equally well and is with higher efficiency.

[1]Details about the simulation can be seen in https://github.com/QuietMorning/SGHMC

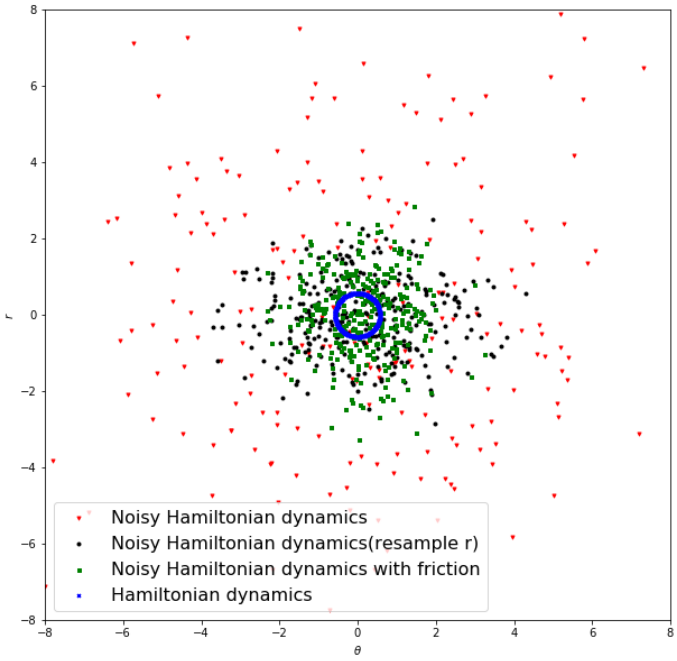

FIGURE 2: Hamiltonian Dynamics Trajectories

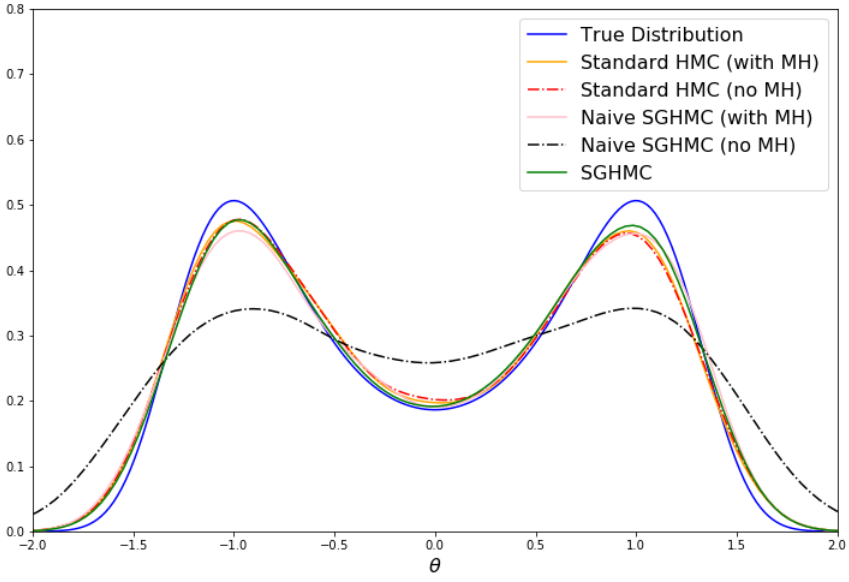

FIGURE 3: Density Plot

4.1.3 2-D Simulation

In this subsection, we contrast sampling of a bivariate Gaussian with correlation using HMC, naive SGHMC, and SGHMC with friction. The potential energy function is $U(\theta) = \frac{1}{2}\theta^T\Sigma^{-1}\theta$ with $\Sigma_{11} = \Sigma_{22} = 1$ and correlation $\rho = \Sigma_{12} = 0.9$. Figure 4 visualizes the first 50 samples of standard HMC, naive SGHMC, and the proposed SGHMC.Both HMC and SGHMC have a good coverage to the whole distribution, while naive SGHMC has more points located in places with low probability.

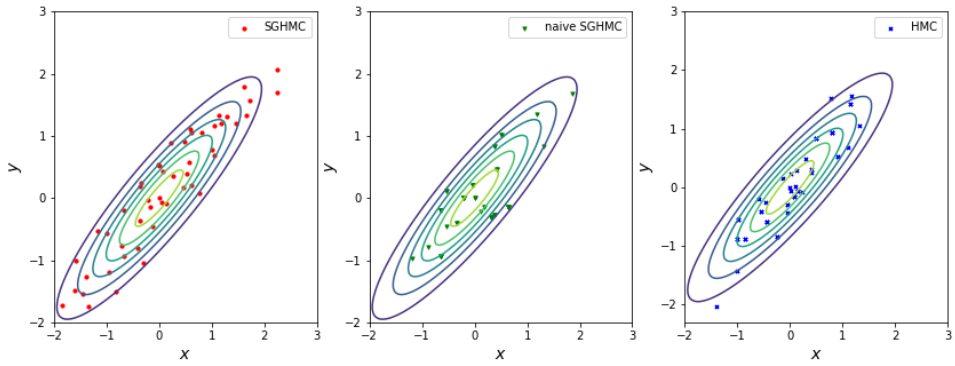

FIGURE 4: First 50 Samples

4.1.4 Bayesian Linear Regression

In the final simulation example, we use a randomly simulated dataset with 5 features and 10000 observations and employ the algorithm to estimate the 5-dimension Bayesian linear regression parameters. The true model is $y = X\theta + u$ where u is an error term with $u \sim \mathcal{N}(0, 1)$, and the true parameters are set as $\theta = [0, 1, 2, 3, 4]^T$. The prior distribution of θ is set as $\theta \sim \mathcal{N}(0, I)$. The posterior distribution of θ given the data is:

$$\begin{aligned} p(\theta \mid X, y) &\propto p(y \mid X, \theta)p(\theta) \\ &\propto \exp\left(-\frac{1}{2}(y - X\theta)^T(y - X\theta) - \frac{1}{2}\theta^T\theta\right). \end{aligned}$$

The corresponding gradient of the log posterior function with respect to θ is:

$$\nabla \log p(\theta \mid X, y) \propto -\theta + X^T y - X^T X\theta.$$

We set the size of minibatch as 1000 and step size as 0.0001. The mean values of estimates from the proposed SGHMC algorithm fit well to the true values. Also, all the variances are close to 0, indicating all the estimates are significant. As a comparison, we also use stardard HMC to sample from the posterior distribution but replace the accurate gradient estimates based on the entire dataset with the noisy gradient estimates based on a minibatch such that these two algorithms have comparable running time. Although the means of the

estimates from HMC are close to the true parameters, the variances are much larger than the variances of samples from SGHMC.

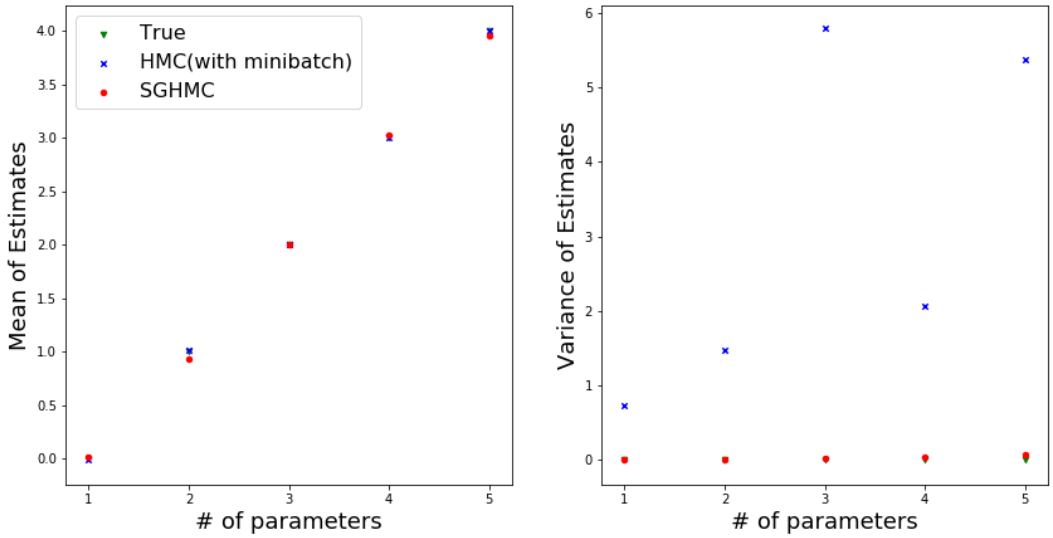

FIGURE 5: Bayesian Linear Regression Results

4.2 Real World Applications

4.2.1 Bayesian Neural Networks for Classification

In this part, we test the SGHMC algorithm using the MNIST data in Chen et al. (2014). The dataset contains 60000 training data and 10000 testing data. Each data corresponds to an image showing a handwritten digit, and the label is the true digit with a value in $[0, 9]$. For each data, we have 784 predictors as the data consist of images that can be described as a 28×28 matrix. The goal is to predict true digits from the image (i.e., the predictors).

We apply the code associated with Chen et al. (2014) while making necessary edits. To utilize the SGHMC algorithm, we create a Bayesian neural network and update the weights using the algorithm. To show the strength of the SGHMC algorithm, we also utilize SGD and SGLD (see Welling and Teh (2011)). From the plot, we can see that the SGHMC algorithm leads to a smaller error compared with SGD and SGLD. The better performance of SGHMC are shown in the following ways. First, it achieves the lowest test error. Second, the number of steps needed for the convergence is only about 50, much smaller than other algorithms. Third, once the convergence is attained, the test error keeps as a steady level.

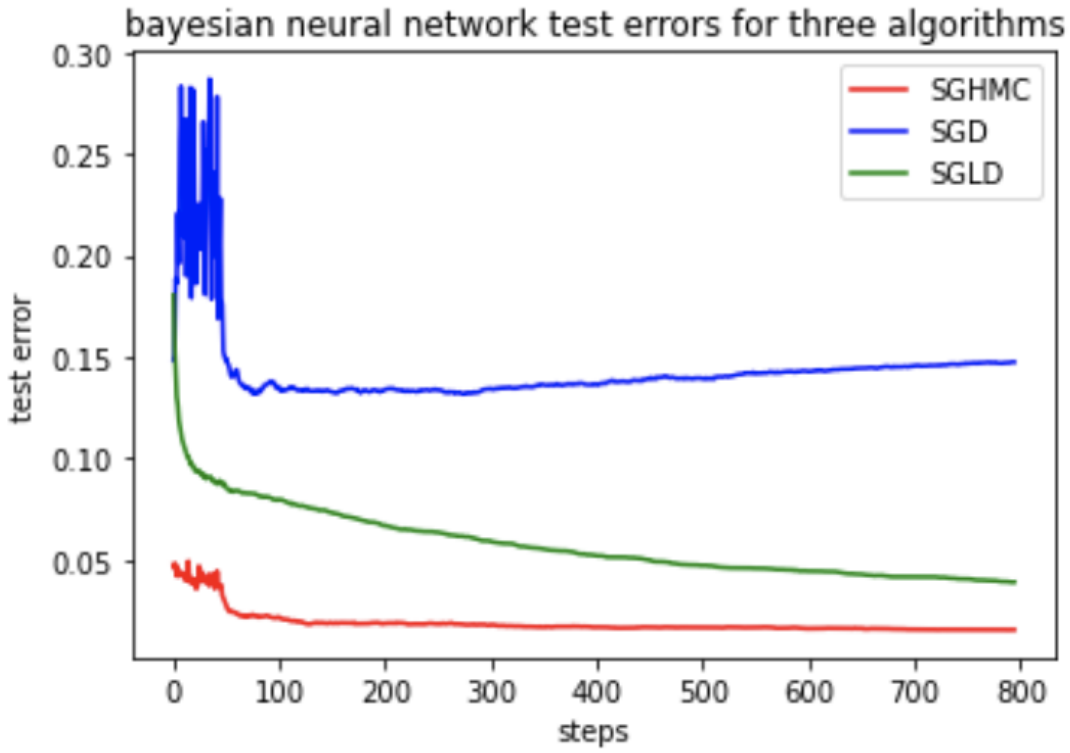

FIGURE 6: Application of SGHMC with Bayesian Neural Network

4.2.2 Prediction of the Airbnb listing prices in Buenos Aires

In this part, we test the SGHMC algorithm using the Buenos Aires Airbnb prices data[2]. The goal is to predict the price class for each Airbnb listing place in Buenos Aires. There are four classes from 1 to 4, where 1 means the cheapest and 4 means the most expensive, and the measure of the prediction is accuracy. Thus, random guess gives us 25% accuracy. The dataset contains 9681 places, and we choose 80% of them ($n = 7744$ places) as our training data and the rest are used as our test data. After data preprocessing, we obtain 73 predictors, such as the number of bedrooms, is the host a superhost, and the cleaning fee.

Our model is as follows. We transfer the expression of the labels using the one-vs-others method, i.e., to a matrix of the form $n \times 4$, where for each observation, if the label is 1, the transferred label is $(1, 0, 0, 0)$; if the label is 2, the transferred label is $(0, 1, 0, 0)$; and so on. Then for each column of the transferred label, we fit a linear regression model assuming a uniform prior and a $y_i \overset{\text{iid}}{\sim} N(x_i^T\theta, 1)$ model, for each $i \in \{1, \ldots, n\}$. Note that we also tried logistic regression here, but the accuracy is smaller compared to the linear regression model. Using the SGHMC algorithm on θ, we obtain four predicted labels for each place, since we do linear regression for each of the four columns in the transferred label. Finally, for each place, we simply take the argument that corresponds to the maximum of these four

[2]https://www.kaggle.com/c/duke-cs671-fall20-airbnb-pricing-data

values. That argmax value will be our prediction class of the Airbnb price class for the place.

Since we use the model

$$y_i \overset{\text{iid}}{\sim} N(x_i^T\theta, 1) \text{ for each } i \in \{1, \ldots, n\},$$

we know the total energy U equals

$$U = \frac{1}{2}(Y - X\theta)^T(Y - X\theta)$$

and

$$\nabla U = -X^TY + X^TX\theta.$$

We compare our SGHMC algorithm to naive SGHMC and HMC under the same set of parameters. The following plot shows the accuracies of three algorithms.

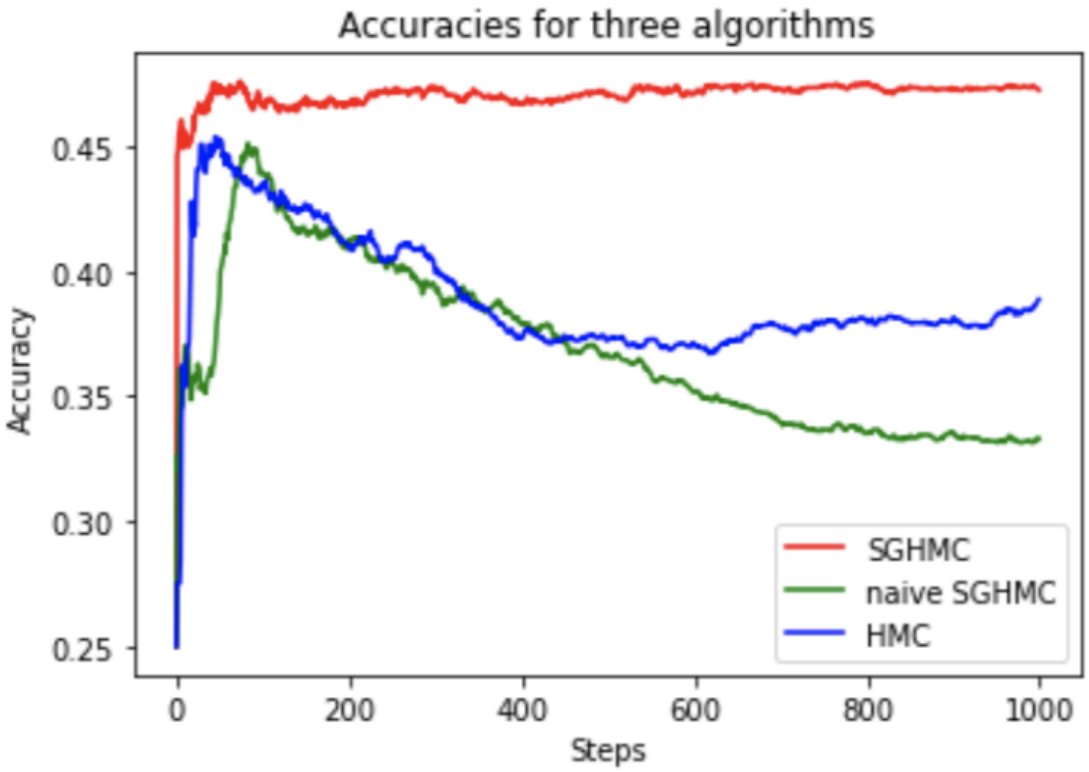

FIGURE 7: Application of SGHMC on the Airbnb dataset

From the plot, we can see that the SGHMC algorithm attains the largest accuracy and the optimal accuracy roughly stays as we increase the number of steps. HMC and naive SGHMC, however, appear to go into trouble of overfitting. Therefore, the SGHMC algorithm has a clear advantage than other two algorithms in this application. Note that the MH step is in HMC and still behaves worse than our SGHMC.

Finally, we provide two remarks on this application. First, it is reasonable to explain the fairly low accuracy of this application. Since the price classes are labeled by 1,2,3,4, it is common to predict a class while the true class is one above or one below the predicted class. Second, our SGHMC may not have a better performance than some machine learning algorithms. Even if this happens, the SGHMC algorithm may still have advantage in the following ways. It takes less computational complexity than complicated machine learning algorithms. Also, the intrinsic nature of SGHMC is derived from physics, so SGHMC is easy to understand and interpret.

5 Conclusion

Combining our results on the simulated data and real world data, we can see that SGHMC provides a better approximation of the posterior distribution in general than the competing algorithms (HMC, naive SGHMC, SGD, SGLD). The estimates given by SGHMC have smaller test errors, smaller variances, and higher accuracy. Sometimes, however, the estimates given by HMC can be closer to the true parameter than those given by SGMHC. We also find that estimating the parameter $\hat{B}$ can have large computational complexity. While we employ $\hat{B} = 0$ as a short cut in our analysis, exploring an efficient algorithm to estimate B is a potential research direction. Also, the results are sometimes sensitive to the choice of ϵ and number of steps in each draw. To further improve our SGHMC implementation, an adaptive parameter-tuning process may be inserted so that we obtain a combination of parameters that produces the best performance.

Contribution

Weiting Miao: SGHMC module, simulated applications, report write-up
Chen An: Real world applications, repository maintenance, report write-up
Ryan Fang: Optimization, packaging, report write-up

Reference

Sungjin Ahn, Anoop Korattikara, and Max Welling. Bayesian posterior sampling via stochastic gradient fisher scoring. *arXiv preprint arXiv:1206.6380*, 2012.

Tianqi Chen, Emily Fox, and Carlos Guestrin. Stochastic gradient hamiltonian monte carlo. In *International conference on machine learning*, pages 1683–1691. PMLR, 2014.

Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth. Hybrid monte carlo. *Physics letters B*, 195(2):216–222, 1987.

Max Welling and Yee W Teh. Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 681–688. Citeseer, 2011.

Appendix

A. Github link to this paper

To reproduce the results in this paper, please check the GitHub repository: https://github.com/QuietMorning/SGHMC.

B. Python package installation instruction

To install the package, please run the following command in your terminal:

```
python3 -m pip install --index-url https://test.pypi.org/simple/ --no-deps maf==0.0.4
```

Then the functions in the package could be imported using the following command in the python3 environment.

```
from maf_sghmc import alg
from maf_sghmc import grad
alg.sghmc_with_grad(...) # use sghmc_with_grad algorithm
grad.U_tilde(...)
```

Examples codes and more details about the package can be found in our github repository using the link shown above.