A replication of Bayesian Inference with Latent Hamiltonian Neural Networks

CHEN, Hongxiao

Department of Economics

The Chinese University of Hong Kong

Part I Literature review and potential methods.

1.1 Literature review

Bayesian inference is a way of using both prior knowledge and observed data to estimate parameters. Instead of giving a single answer, it provides a full distribution, which helps us understand uncertainty more clearly. The core idea can be written as:

$$f(q \mid D) \propto f(D \mid q) f(q)$$

This equation shows how Bayesian inference combines the likelihood of the data and the prior knowledge. While the formula looks simple, calculating the posterior distribution directly is very hard for complex problems. To solve this, we often use computational methods like Markov Chain Monte Carlo (MCMC).

MCMC is a popular tool for sampling from the posterior distribution. It works by creating a random process (a Markov chain) that eventually gives us samples from the target distribution. These samples can then be used to estimate things like the mean, variance, or confidence intervals (Metropolis et al., 1953; Hastings, 1970). One important detail in MCMC is the burn-in period. This is the initial part of the chain, which is discarded because it may not represent the true distribution yet (Gelman & Rubin, 1992). Skipping this step can lead to biased results.

However, traditional MCMC methods, like the Metropolis-Hastings algorithm, can be slow or inefficient when dealing with high-dimensional problems or when parameters are strongly related. To address this, Hamiltonian Monte Carlo (HMC) was developed. HMC uses gradient information from the posterior distribution to guide the sampling process (Neal, 2011). By simulating the dynamics of physical systems, it moves through the parameter space in larger, more efficient steps. This avoids the "random-walk" behavior that slows down traditional MCMC. But HMC is not perfect: it needs parameters like step size and trajectory length to be carefully tuned, which can be tricky.

To make HMC easier to use, the No-U-Turn Sampler (NUTS) was introduced. NUTS automatically decides when to stop a trajectory, so users don’t have to set a trajectory length themselves (Hoffman & Gelman, 2014). This makes NUTS more user-friendly and efficient. Because of this, it is the default sampler in many popular Bayesian software tools, such as Stan. One feature that makes NUTS special is how it builds "sampling trees" using recursion. This means the algorithm calls itself repeatedly to explore the parameter space. Understanding this process can be difficult without prior knowledge of recursion and Hamiltonian sampling (Dhulipala, 2023).

When using MCMC, it’s important to check the quality of the samples. One way to do this is by calculating the effective sample size (ESS), which measures how many independent samples the chain actually represents (Gelman et al., 2013). If the ESS is low, it means the samples are highly correlated, and the chain may need more iterations. Improving ESS has been a key goal in creating better methods like HMC and NUTS.

Aside from traditional methods, Dhulipala et al. (2023) introduced a new method that improve the performance of NUTS. They use Hamilton Neural Networks (HNN) to simulate Hamiltonian, and use auto-differentiation of HNN to find the

1.2 Methods

With respect to algorithms, I follow Dhulipala (2023) to apply traditional HMC, LHNN-HMC, NUTS, and LHNN-NUTS with online monitoring to replicate the results. The codes are written in Python 3 and TensorFlow for vectorized computation. Most results shown in Dhulipala (2023) are accompanied by detailed instructions of which methods are used. However, in part 7, the author did not clearly state his methods, and I have to pick the best method among several potentials.

First, in 7.1 where the author discusses how LHNN-NUTS is applied to stationary distribution of Allen-Cahn stochastic partial differential equations, he did not mention how it is discretized. One way of discretization is to discretize the whole range into intervals that have same length. Another way is to discretized it into uneven intervals, in which values that are farther away from the mean value falls into intervals with wider range, while values that are close to the mean value and with higher densities falls into intervals with narrower range. Since in Dhulipala (2023) the choice of the way of discretization is not mentioned, I assume the default method is to discretized the distribution into even intervals, which is also the method I applied.

In 7.2, there are two methods to calculate $f(x,y)$ and replicate figure 12. The first method is to infer $k(x,y)$ and then use the PDE in equation (27) to obtain $f(x,y)$, which is the method the authors use. The other method is to directly infer $f(x,y)$. I tried both methods to see which one performs better.

Another choice to be considered is how to perform traditional HMC. I can use HMC that TensorFlow probability api provide, or I can write my own script. I chose the second method, because aside from traditional HMC, LHNN-HMC is also implemented in this replication. The benefit of writing my own script for traditional HMC is that traditional HMC and LHNN-HMC methods can be more consistent with each other, with same inputs, structures, and outputs.

The last choice I made is on what method to be used to evaluate effective sample size. One choice is that I can write my own code based on the following logics of how effective sample size is calculated. Suppose the samples I obtained from HMC are $\{x_1, x_2, \ldots, x_N\}$, I need to compute the correlation coefficient between each variable and its lag term.

$$\rho_k = \frac{Cov(x_t, x_{t+k})}{Var(x_t)}$$

then compute the ESS

$$ESS = \frac{N}{1 + 2\sum_{k=1}^{\infty} \rho_k}$$

Sometimes, we can apply Geyer truncation: by picking an upper limit of the lag represented as k^*, we can write the ESS as $\frac{N}{1+2\sum_{k=1}^{k^*} \rho_k}$.

Another method is to use TensorFlow Probability API to calculate the ESS, which is also the approach I use. First, TFP is well-tested. If implementing ESS myself, small coding mistakes could lead to incorrect results or subtle bugs, especially when the data is high-dimensional. Also, ESS in TFP leverages TensorFlow’s ability to use GPU acceleration. Lastly, ESS in TFP has built-in techniques like Geyer truncation.

Part II Answers to the questions.

2.1 Parts that are hard to understand

Several parts in Dhulipala (2023) take more time than expected to be fully understood. The first part is on how to relate equation (10) to the familiar form of Bayesian Equation. The derivation is as follows:

$$f(q,p) \propto \exp\big(-U(q)\big)\exp\big(-K(p)\big) \propto [L(q|D)g(q)]\exp\big(-K(p)\big)$$

Rewrite $L(q|D)$ as $f(D|q)$, representing the likelihood function, and $g(q)$ as $f(q)$, representing the prior, we can turn it into the form that we are familiar with in Bayesian estimation:

$$f(q|D) \propto f(D|q)f(q)$$

Therefore, in application, when we need to calculate $U(q)$ in Python, we can calculate it as

$$U(q) \leftarrow -\log\big(f(q|D)\big)$$

When reading materials on the sampling process of using HMC, either in Dhulipala (2023) and his Pytorch code, or in the code for HMC in TensorFlow probability, I found that samples are generated using for-loop rather than generated in a vectorized way. Initially I was wondering if it could be possible to improve the performance by applying vectorized method, which I found not applicable after understanding the underlying mechanism. In HMC, since a second sample should start and the end position of its previous sample, a for loop is necessary for we have to wait for the previous sample to reach its end. Of course, parallel computation is possible if we generate multiple independent chains, within which for-loop is used for HMC algorithms. In this replication work, only one chain is used.

A second part that needs more explanation in Dhulipala (2023) is why latent variables are applied. In figure 6, the author compares the errors of HNN and L-HNN and shows that L-HNN is more efficient. But it seems to me that using the latent variable is simply equivalent to adding one more hidden layer in which each neuron has same weight and no bias, just as equation (15) and (16) shows in Dhulipala (2023). Below I re-demonstrated these equations.

$$u_p = \phi(w_{p-1}u_{p-1} + b_{p-1})$$
$$\lambda = w_p u_p + b_p$$
$$H_\theta = \Sigma_{i=1}^{d}\lambda_i$$

The last equation implies that H_θ can be the single output of HNN, with its previous hidden layer being a layer with neurons that have same weight, no bias and no activation. Therefore, a similar result as figure (6) may be shown if we add an additional hidden layer but still use HNN to output Hamiltonian.

A third part that is not possible to be understood without reference to Hoffman and Gelman [4] is what NUTS is, how it works, and how trees are built. Recursion within a function in Python, which is used in the BuildTree function, also requires readings on external materials to be fully understood.

Chapter 7 in Dhulipala (2023) also differs from previous work. For the part of Allen-Cahn PDE, the discretization of stationary distribution seems incorrect. To build the Allen-Cahn stable distribution. I take $i\Delta x$ as a current position in the discretized space, and $i\Delta x + \Delta x$ as the position next to the current position in the space, which is similar to the case in N-D Rosenbrock density. Therefore, $\Delta x = \frac{1}{Dimension}$, where $Dimension = 25$ indicating the distribution space is discretized into 25 intervals.

2.2 Missing items and typos

2.2.1 Missing items

Several items are missing in this paper on how specific results are generated. The first missing items are with respect to the training process. How many neurons in the last layer that generate latent variables should be in the last layer of L-HNN for 1-D Gaussian mixture density and 3-D Rosenbrock density is unknown. Also, the author mentioned that in most cases the training step is 100,000, but the batch size of each step is unknown. In my application, I assume the number of neurons of the 'latent layer' is the same as it is in the hidden layers, which is 100. The batch size I use follows the default case in Dhulipala (2023)'s Pytorch code, which is 1,000.

A second factor to be figured out when replicating the results is how numbers of gradients evaluated in each case are counted. For example, in table 1 of Dhulipala (2023), information on ESS per gradient of target density is presented. How gradients for traditional HMC and L-HNNs HMC are relatively simple, for the length of trajectories and total steps that required are known, and how many gradients were evaluated when generating samples for training L-HNNs is also known. However, the parts for counting gradients for traditional and L-HNNs NUTS is unclear. First, when simulating using L-HNNs NUTS, it is important to record how many times the algorithm switches to traditional NUTS. Second, we need to figure out how many gradients are evaluated when we are building the trees, including in the root case and during the recursion periods. Similar cases are presented in Table 2, 3 and 4. Therefore, accurately counting the gradients evaluated is the crucial part of replicating these tables.

The solution for counting the gradients is as follows: within BuildTree, when the recursion does not reach the root, number of gradients counted are passed in as input and out as output to track record. When the recursion reaches the root, numbers counted will plus 2 if traditional leapfrog is implemented, because each traditional leapfrog step evaluates twice gradients, and only one step is taken in the root case. The total gradients equal to the sum of gradients returned from BuildTree each time BuildTree is called in the main loop.

The third item I find missing is how the slice variable u, which is used for evaluating whether the sampling error of using L-HNNs exceeds threshold as presented in equation (20) in Dhulipala (2023), is defined. Fortunately, in Hoffman and Gelman (2014), details of this variable are given.

Another missing item is how traditional NUTS is implemented and how the recursion process in BuildTree function is realized. The author omitted these parts because from Hoffman and Gelman (2014) exact algorithms can be obtained.

In Chapter 7, the author did not give details on how HNN for Allen-Cahn is trained, and how training data is generated. He mentioned that 10,000 gradient computations are required by generating HMC trajectories. So I assume there are 25 samples, each with a trajectory with length equal to 10 and step size equal to 0.025. The training process follows previous examples, and the dimension for latent layer is set to 25. Similar issue exists for the part of elliptic partial differential equation. The author mentioned that a total of 64000 gradient was evaluated during training, so I set sample size equal 40, trajectory length 40 and step size 0.025.

The description on elliptic partial differential equation is also too simple in this paper. The author does not provide the procedure on how this experiment is conducted. A detailed description on how I replicate this part will be presented in the chapter of replication.

2.2.2 Typos and mistakes

Mistake 1:

With respect to typos that are essential for implementation, in line 16 of Algorithm 4 of Dhulipala (2023), $v_j = -1$ is mistakenly written as $j = -1\ WRONG$. The results would be wrong and confusing if following this algorithm.

Mistake 2:

A second typo is in the discretization of stationary distribution of Allen-Cahn stochastic PDE in 7.1. The stationary distribution is defined as

$$\pi(u) \propto \exp\left(-\int_0^1 \left(\frac{1}{2}\left(\frac{\partial u}{\partial x}\right)^2 + V\big(u(x)\big)\right) dx\right)$$

which the author mistakenly discretizes it as

$$\pi(u) = \exp\left(-\sum_0^{d-1} \frac{1}{2\Delta x}\big(u(i\Delta x + \Delta x) - u(i\Delta x)\big)^2 + \frac{\Delta x}{2}\Big(V\big(u(i\Delta x + \Delta x)\big) - V\big(u(i\Delta x)\big)\Big)\right) WRONG$$

and the correct form should be

$$\pi(u) = \exp\left(-\sum_1^{d-1} \frac{1}{2\Delta x}\big(u(i\Delta x + \Delta x) - u(i\Delta x)\big)^2 - \frac{\Delta x}{2}\sum_1^{d} V\big(u(i\Delta x)\big)\right)$$

Mistake 3:

Another mistake lies in both his Pytorch code and in the paper of how the author calculates the potential energy of 2-D Neal Funnel distribution. In his code, the potential energy is calculated as

$$U(q_1, q_2) = \frac{q_1^2}{2 \times 3^2} + \frac{q_2^2}{2 \times (0.5 \times \exp(q_1))^2} WRONG$$

with the 2-d Neal's funnel density

$$f(q) \propto \begin{cases} q_1 = N(0,3) \\ q_2 = N(0, \exp(q_1)) \end{cases} WRONG$$

where he takes both 3 and $\exp(q_1)$ as standard deviation. However, the correct form should be

$$f(q) \propto \begin{cases} q_1 = N(0,3^2) \\ q_2 = N(0, \exp(q_1)) \end{cases}$$

with 3 and standard deviation while $\exp(q_1)$ as variance, and the correct potential energy should be represented as

$$U(q_1, q_2) = \frac{q_1^2}{2 \times 3^2} + \frac{q_2^2}{2 \times \exp(q_1)} + \frac{1}{2}q_1$$

with the following derivation:

$$f(q_1, q_2) = \frac{1}{\sqrt{2\pi} \times 3}\exp\left(-\frac{q_1^2}{2 \times 3^2}\right) \times \frac{1}{\sqrt{2\pi} \times \exp\left(\frac{q_1}{2}\right)}\exp\left(-\frac{q_2^2}{2 \times \exp(q_1)}\right)$$

Taking negative log and neglecting constant term,

$$U(q_1, q_2) = \frac{q_1^2}{2 \times 3^2} + \frac{q_2^2}{2 \times \exp(q_1)} + \frac{1}{2}q_1$$

Even if I take both 3 and $\exp(q_1)$ as standard deviation, the correct formula of potential energy is still different from the author's version. Following the same derivation, we can obtain:

$$f(q_1, q_2) = \frac{1}{\sqrt{2\pi} \times 3}\exp\left(-\frac{q_1^2}{2 \times 3^2}\right) \times \frac{1}{\sqrt{2\pi} \times \exp(q_1)}\exp\left(-\frac{q_2^2}{2 \times \exp(q_1)^2}\right)$$

Neglecting constant term and taking negative log,

$$U(q_1, q_2) = -\log[f(q_1, q_2)] \propto \frac{q_1^2}{2 \times 3^2} + \frac{q_2^2}{2 \times \exp(q_1)^2} + q_1$$

2.3 Replication results

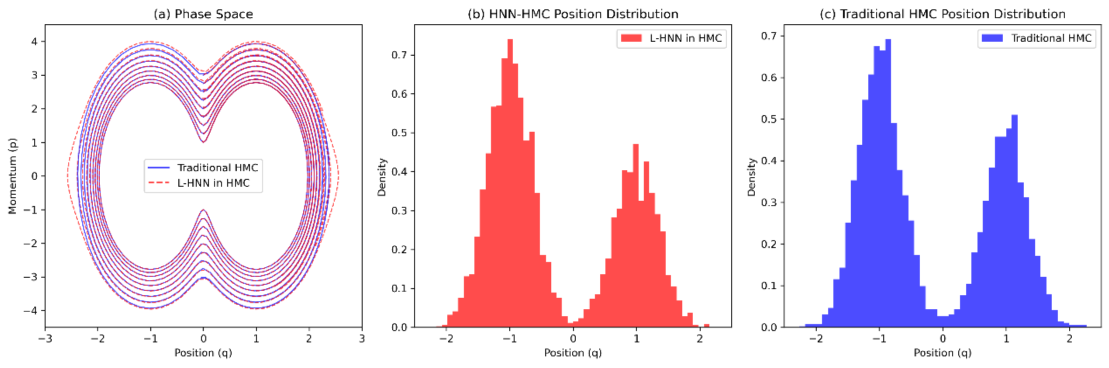
In the replication part, I failed in replicating figure 12 but successfully replicated all other results.

2.3.1 Replication for 1-D Gaussian mixture density

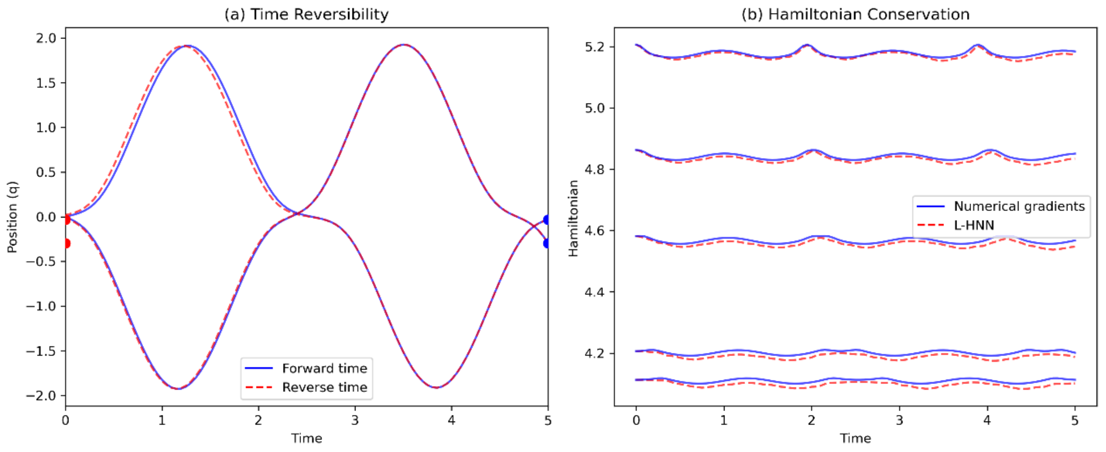
The first set of replications focuses on 1-D Gaussian mixture density, which includes figure 2 and 3, and Table 1. Figure 2 compares traditional HMC and L-HNN in HMC. Figure 3(a) shows two trajectories generated by using L-HNN HMC for 1-D Gaussian mixture density, which start at same position with different initial momentum, have the property of time reversibility. Figure 3(b) shows both traditional HMC and L-HNNs have the property of Hamiltonian conservation at different energy levels for 1-D Gaussian mixture density. Table 1 compares effective samples per gradient for four methods. The scale of my result is close to that in Dhulipala (2023).

Note that the total number of gradients evaluated for L-HNN NUTS is 8,000 which equals the gradients evaluated in training. It suggests that the threshold is not reached, and the sampling is not switched to traditional NUTS in the case of 1-d Gaussian mixture density.

Replication of Figure 2



Replication of Figure 3



Replication of Table 1

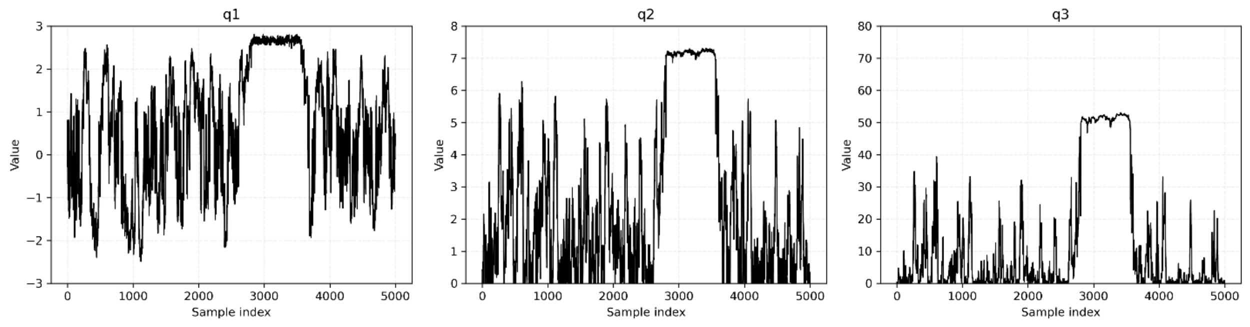
Method	ESS	# of Gradients	ESS/grad
Traditional HMC	62.86	1000000	6e-5
L-HNN HMC	43.46	8000	5.43e-3
Tradtional NUTS	42.84	137032	3.1e-4
L-HNN NUTS	53.72	8000	6.72e-3

2.3.2 Replication for 3-D Rosenbrock density

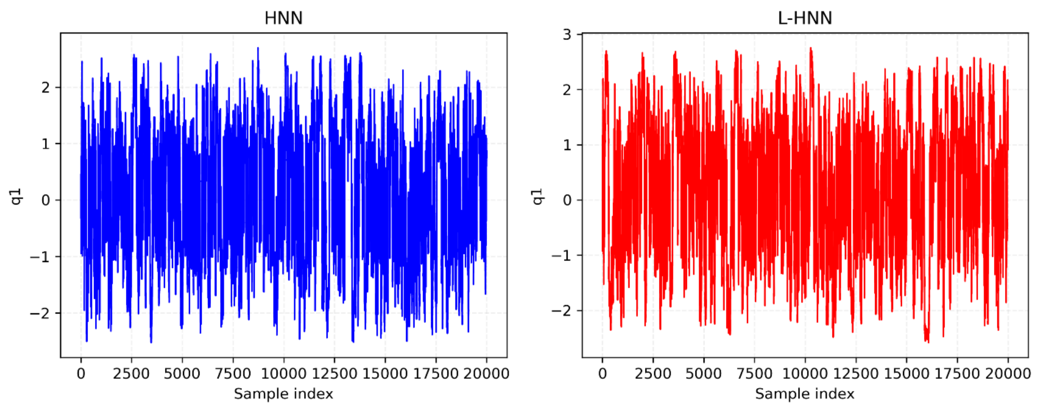
The second set of replication focuses on 3-D Rosenbrock density, which includes figure 4, 5, 6 and 7, as well as Table 2.

Figure 4 presents sampling using L-HNNs in NUTS without online error monitoring mechanism of the 3 positions in 3-D Rosenbrock density, in order to capture the limitation of L-HNN in regions of low probability density. To eliminate this limitation, samples of the first position in 3-D Rosenbrock are used to present the effect of both HNNs and L-HNNs in NUTS with online error monitoring in Figure 5 and proved that online error monitoring is an effective method. Figure 6 is presented to show L-HNN is more efficient than HNN because L-HNN has lower error level. As the computation resource is limited, I sampled 20,000 times for figure 5 and 6, instead of 35,000 times as it is in the paper.

Replication of Figure 4



Replication of Figure 5



Replication of Figure 6

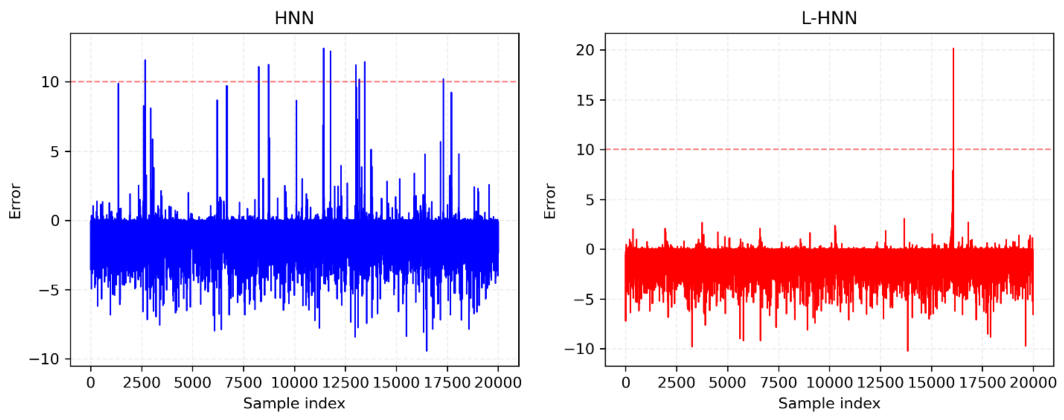
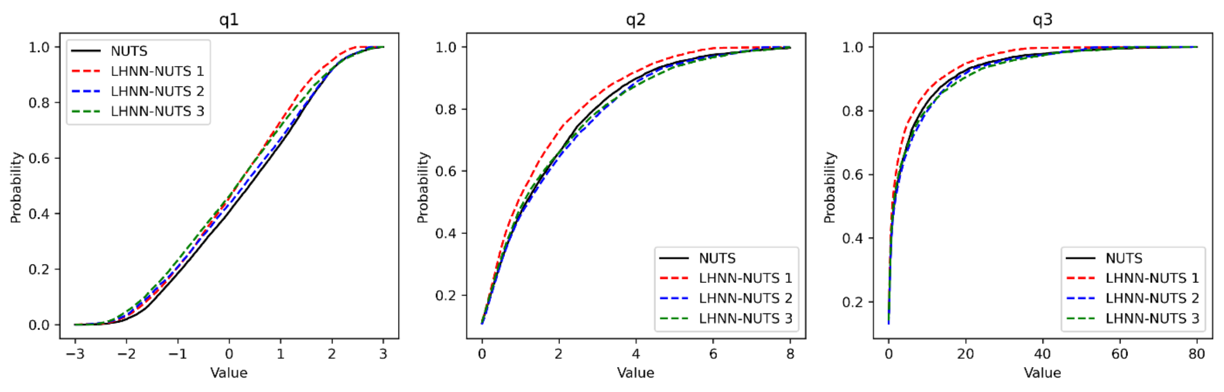


Figure 7 presents empirical cumulative functions plots for three positions in 3-D Rosenbrock density in order to compare the influence of different training dataset size of L-HNNs. Three datasets, all with 40 samples but with different end times – T = 100, 150, and 250 units – are used. To further investigate, Table 2 compares the performance between traditional NUTS and HNN-NUTS with three different training datasets. Because of the computation limitations, I simulate only 20,000 samples instead of 125,000 samples in the paper, and with the burn-in samples kept as 5000. The level of my ESS replication and the ESS in the paper should be at around 15,000/120,000 = 0.125. Ratio of level of number of gradients evaluated should be at around 20,000/125,000 = 0.16. Since I simulate fewer samples in the replication, the average ESS per gradient of target density should be lower.

Replication of Figure 7



Replication of Table 2

Method	ESS	# of Gradients	ESS/grad
NUTS	(156.67, 273.84, 311.82)	4,028,500	0.000061
[LHNN-NUTS 1] Mt = 40; T = 100	(194.35, 335.52, 339.61)	Evaluation: 435,840 Training: 160,000 595,840	0.000486
[LHNN-NUTS 2] Mt = 40; T = 150	(144.38, 204.71, 205.22)	Evaluation: 99,576 Training: 240,000 339,576	0.000544
[LHNN-NUTS 3] Mt = 40; T = 250	(139.46, 250.09, 259.61)	Evaluation: 34,924 Training: 400,000 434,924	0.000498

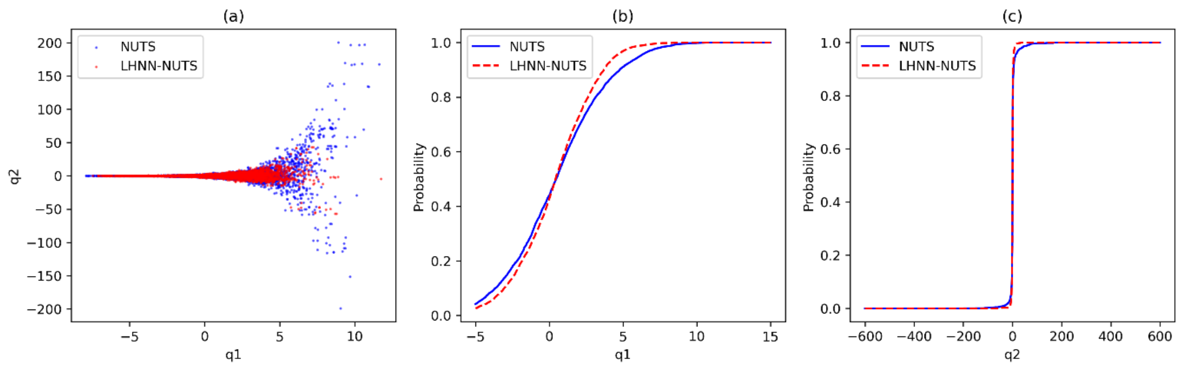
2.3.3 Replication for Chapter 6

Figure 8, 9 and 10 display the application of LHNN-NUTS and NUTS on 2-D Neal's funnel density, 5-D ill conditioned Gaussian and 10-D degenerate density. I simulate 10,000 samples for 2-D Neal's funnel density with 5000 samples as burn-in, 20,000 samples for 5-D ill conditioned Gaussian with 5,000 samples as burn-in, and 30,000 samples for 10-D RosenBrock with 5,000 samples as burn-in. Choices of number of samples are related to my computation resource.

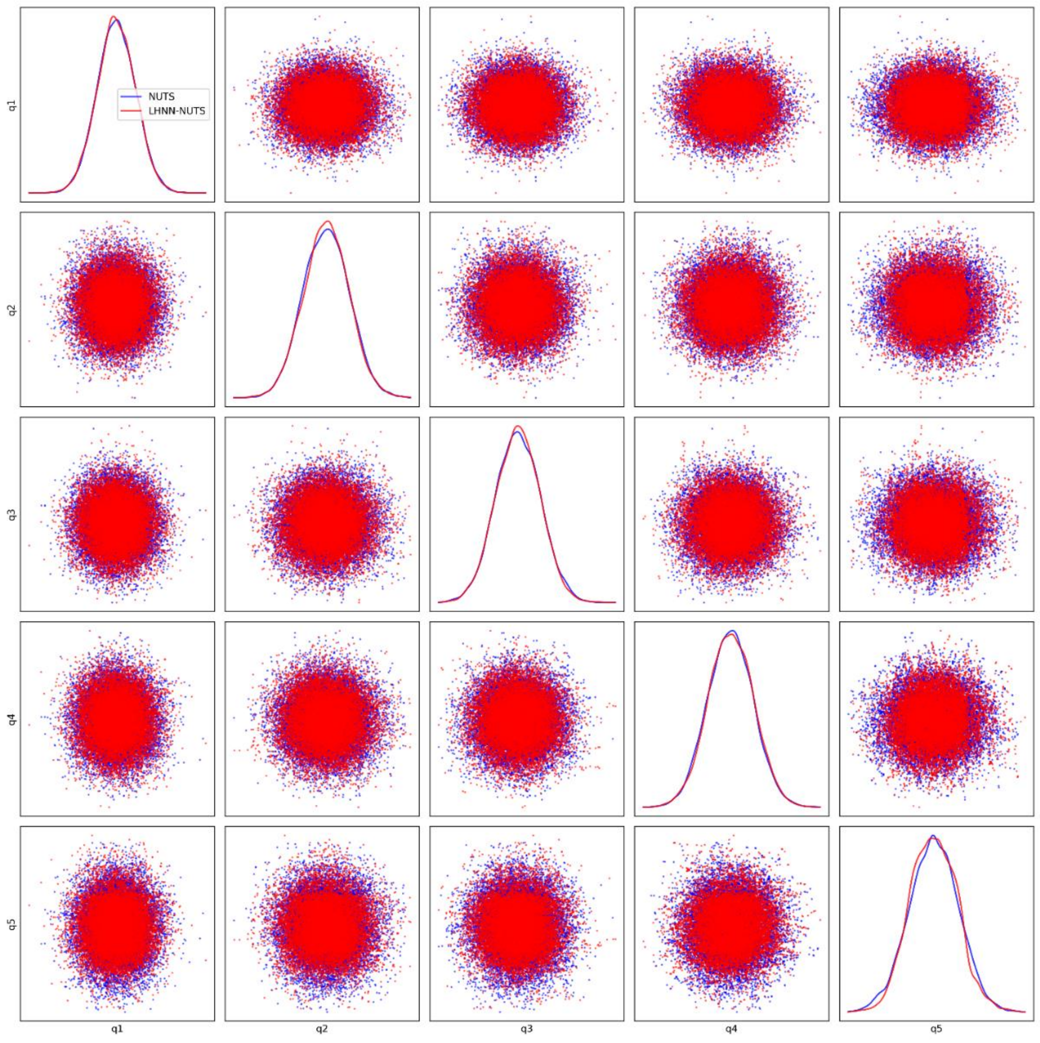
The shape of replication of figure 8 is similar to that in the paper. The ranges differs, however, and it could be out of the difference between the number of simulated samples. In my simulations, I simulate 10000 samples with 5000 burn-in, which leads to only 5000 samples left for plotting. In the paper, 25000 samples were simulated with 5000 burn-in, so that there were 20,000 samples left of plotting. The effective sample size in the replication is of similar scale to the paper. We have 5000 and 20000 samples after burn-in respectively, and our effective sample size for LHNN-NUTS are (242.97, 260.76) and (1012.45, 718.07). The number of gradients evaluated in my work is higher than that in the paper.

Replication of figure 9 is identical to the one in the paper. Effective sample size shares similar pattern. Since the first dimension is a Gaussian distribution, the effective sample size should be equal to the number of total samples net of burn-in samples. In my case, the ESS for the first dimension is 15,000 and in the replicated paper, it is 20,000. However, the number of gradients evaluated in my works is still higher than that in the paper.

Replication of Figure 8



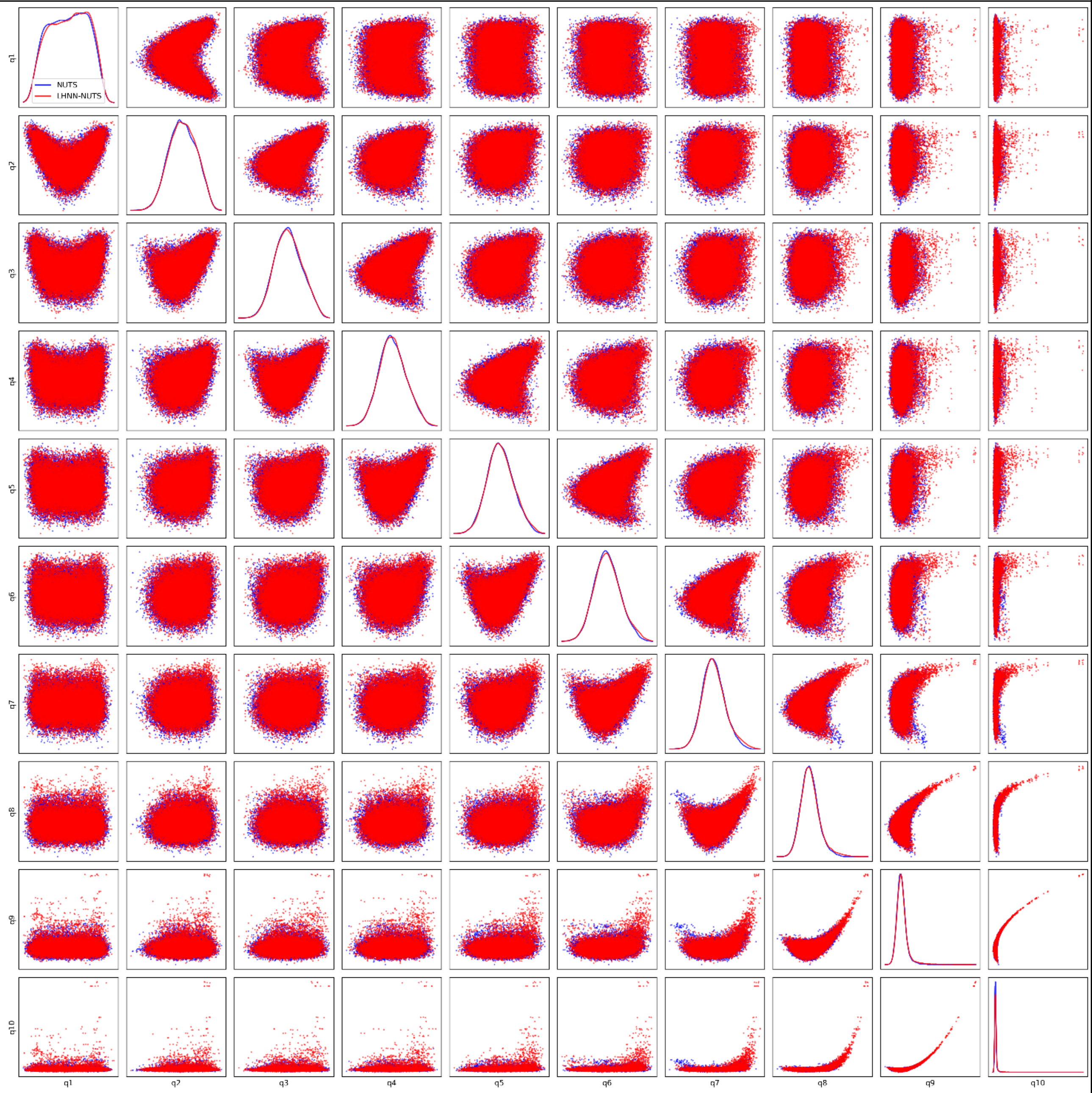
Replication of Figure 9



Replication of Figure 10 presents a result that is similar to Figure 10 in the paper. If we take the difference in the number of samples into consideration, the ESS of 10-D Rosenbrock is also similar. However, the number of gradients evaluated is much higher in my case, leading to a lower ESS per grad. This suggests that online-monitoring mechanism is triggered more often in my replication and the algorithm is switched to traditional leapfrog. This could be out of unsatisfactory training of the model.

A summary of these three replications is that I can generate similar graphs and effective sample sizes, but with larger numbers of gradients evaluated. The difference in the number of gradients evaluated can be out of several reasons. First, our ways of training HNN may differ. For example, we may have different batch size within each training step. A second reason can be that our ways of counting how many gradients are evaluated differ. The author does not point out clearly how he counts the gradients, and his method could be different from what I present in 2.2.1 in this replication report.

Replication of Figure 10



Replication of Table 3

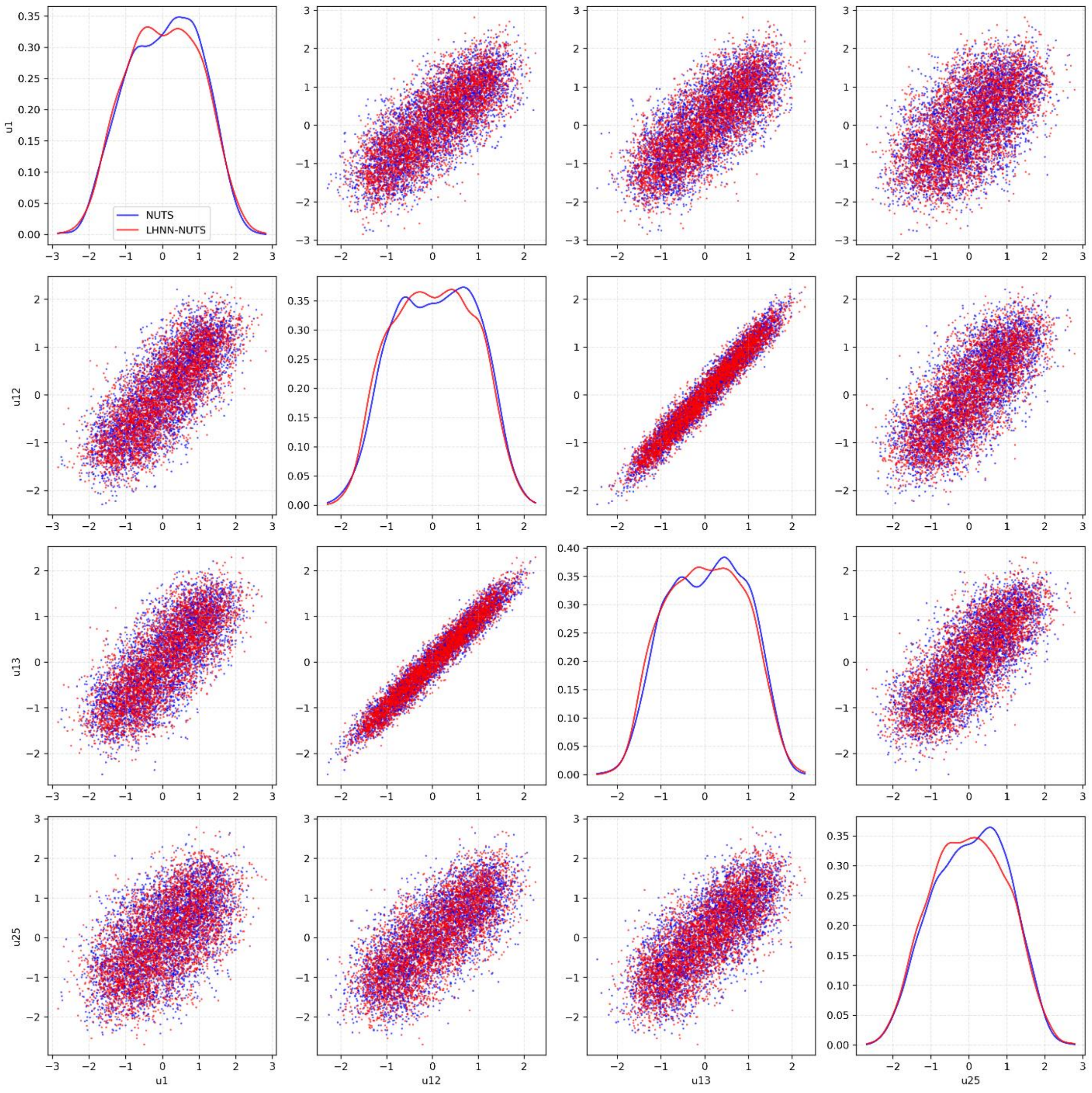
Method	ESS	# of Gradients	ESS/grad
2-D Neal's funnel			
NUTS	(179.56, 168.79)	3140956	0.000055
LHNN-NUTS	(242.97, 260.76)	Evaluation: 2469628 Training: 400000 2869628	0.000088
5-D ill-conditioned Gaussian			
NUTS	(15000.00, 13423.01, 10837.83, 5969.50, 2057.69)	14813328	0.000638
LHNN-NUTS	(15000.00, 12947.75, 9332.47, 4630.92, 1639.86)	Evaluation: 10273704 Training: 400000 10673704	0.000816
10-D degenerate Rosenbrock			
NUTS	(3106.89, 5863.48, 5720.49, 5682.46, 6026.40, 5987.77, 4759.87, 3803.30, 3212.24, 2400.10)	3179968	0.001464
LHNN-NUTS	(3139.47, 5785.96, 5478.38, 5682.69, 3705.39, 2456.14, 1608.48, 1124.30, 691.41, 566.10)	Evaluation: 2958972 Training: 400000 3358972	0.000900

2.3.4 Replication for Chapter 7

2.3.4.1 Allen-Cahn stochastic PDE

The first part of Chapter 7 discusses how LHNN-NUTS can be applied to updating the posterior of stationary distribution of Allen-Cahn stochastic partial differential equation. The stationary distribution is discretized into 25 parameters, with each interval being as a parameter to be inferred. 5000 samples are generated using NUTS and LHNN-NUTS with 1000 samples as burn-in. The replication result is as below:

Replication of Figure 11



Replication of Table 4 Allen-Cahn stochastic PDE

Method	ESS	# of Gradients	ESS/grad
NUTS	336.76	1249264	0.000270
LHNN-NUTS	357.69	Evaluation: 635132 Training: 10000 645132	0.000554

The replication of figure 11 and the magnitude of effective sample size are similar those in the paper. LHNN-NUTS also requires lower level of number of gradients evaluated. However, the number of gradients evaluated in my replication is still much higher than that in the paper.

2.3.4.2 Method 1 for Elliptic PDE

Before replicating Figure 12 that display results for elliptic partial differential equation, I want to supple the paper with more details on how it is realized. The Bayesian equation is given by

$$p(k|f) \propto p(f|k)p(k)$$

50 observations are generated, and the likelihood function follows

$$p(f_i|k) \propto \exp\left(-\frac{1}{2}\sum_{i=1}^{50}\left(\hat{f}(x_i,y_i;k)-f_i\right)^2\right)$$

where f_i is the observed data of each 50 points that is generated from equation (27) plus a zero-mean unit variance Gaussian noise. $\hat{f}(x_i,y_i;k)$ is obtained from the PDE in equation (27) after we have an inference of k. Then, the posterior becomes

$$p(k|f) \propto \exp\left(-\frac{1}{2}\sum_{i=1}^{50}\left(\hat{f}(x_i,y_i;k)-f_i\right)^2\right)p(k)$$

Let prior $p(k)$ be a constant,

$$U(q) = -\log p(k|f) \propto \frac{1}{2}\sum_{i=1}^{50}\left(\hat{f}(x_i,y_i;k)-f_i\right)^2$$

where q is a discretized form of $k(x,y)$ with 50 dimensions. So we can also represent $U(q)$ as

$$U(q) = \frac{1}{2}\sum_{i=1}^{50}\left(\hat{f}(x_i,y_i;q_i)-f_i\right)^2$$

On the other hand, if I assume the prior of k follows a zero-mean Gaussian distribution (which I will not use in the replication), then

$$p(k|f) \propto \exp\left(-\frac{1}{2}\sum_{i=1}^{50}\left(\hat{f}(x_i,y_i;k)-f_i\right)^2\right)\exp\left(-\frac{1}{2\tau^2}\sum_{i=1}^{50}k_i^2\right)$$

so that

$$U(q) = \frac{1}{2}\sum_{i=1}^{50}\left(\hat{f}(x_i,y_i;q_i)-f_i\right)^2 + \frac{1}{2\tau^2}\sum_{i=1}^{50}q_i^2$$

The most unclear part is how $\hat{f}(x_i,y_i;q_i)$ is obtained. Equation (27) states that

$$\hat{f}(x,y;q) = \frac{\partial}{\partial x}\left(k(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(k(x,y)\frac{\partial u}{\partial y}\right)$$

where $u = sin2x + sin2y$ is known with $k(x,y)$ unknown. Let $g(x,y) = (x,y)\frac{\partial u}{\partial x}$, we need to evaluate $\partial g/\partial x$ and $\partial g/\partial y$. Traditional finite difference method, for example,

$$\frac{\partial g}{\partial x} \approx \frac{g(x_{i+1},y)-g(x_i,y)}{x_{i+1}-x_i}$$

does not apply, because (x_i,y_i) pairs are randomly selected from uniform distribution and so we cannot find the difference between $g(x_{i+1},y_i)-g(x_i,y_i)$. Instead, I use nearest neighbor approximation to calculate the first order difference. Specifically,

$$\frac{\partial g}{\partial x} \approx \frac{g(x_j,y_j)-g(x_i,y_i)}{x_j-x_i}$$

where (x_j, y_j) is the nearest neighbor of (x_i, y_i). After updating $q_i = k_i$, I am able to evaluate

$$\hat{f}(x, y; q) = \frac{\partial}{\partial x}\left(k(x, y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(k(x, y)\frac{\partial u}{\partial y}\right)$$

for each sample at each 50 dimensions. Lastly, I take the mean of each sample's $\hat{f}(x, y; q)$ as the result for replicating Figure 12, in which the partial differential equation are still calculated using nearest neighbor approximation. I did not generate an ideal result using this method. The momentum explodes and soon leads to outputs with NANs when I was generating training data, which could be out of inaccurate calculation of the PDE from using nearest neighbor approximation.

2.3.4.3 Method 2 for Elliptic PDE

Another method is to infer $f(x, y)$ rather than $k(x, y)$. Initially, the author claims that he is inferring $k(x, y)$ but his result suggests that he is inferring $f(x, y)$. These two questions are two sides of one coin but differ in computational complexity. Inferring $f(x, y)$ does not require solving the PDE in equation (27), and we do not have to assume a target distribution for $k(x, y)$. Below I give a detailed description of this part. The Bayesian equation is given by

$$p(\mu|f_{obs}) \propto p(f_{obs}|\mu)p(\mu)$$

where $\mu = (\mu_1, \mu_2, \ldots, \mu_{50})$ is the mean of $f(x, y)$ at 50 "sensor" locations and is a 50-dimension parameter to be inferred. f_{obs} is the 'observed' data generated through equation (27) plus a unit variance Gaussian noise. $p(\mu)$ is the prior, $p(f_{obs}|\mu)$ is the 50-dimension likelihood and $p(\mu|f_{obs})$ is the posterior and $U(\mu) \leftarrow -\log p(\mu|f_{obs})$ is the posterior in the form of potential energy that we need to sample from. Specifically, to start with, assuming the variance is 1, we can represent the posterior after discretization as

$$p(\mu|f_{obs}) \propto \exp\left(-\frac{1}{2}\sum_{i=1}^{50}(f_{obs} - \mu_i)^2\right)$$

So, we have

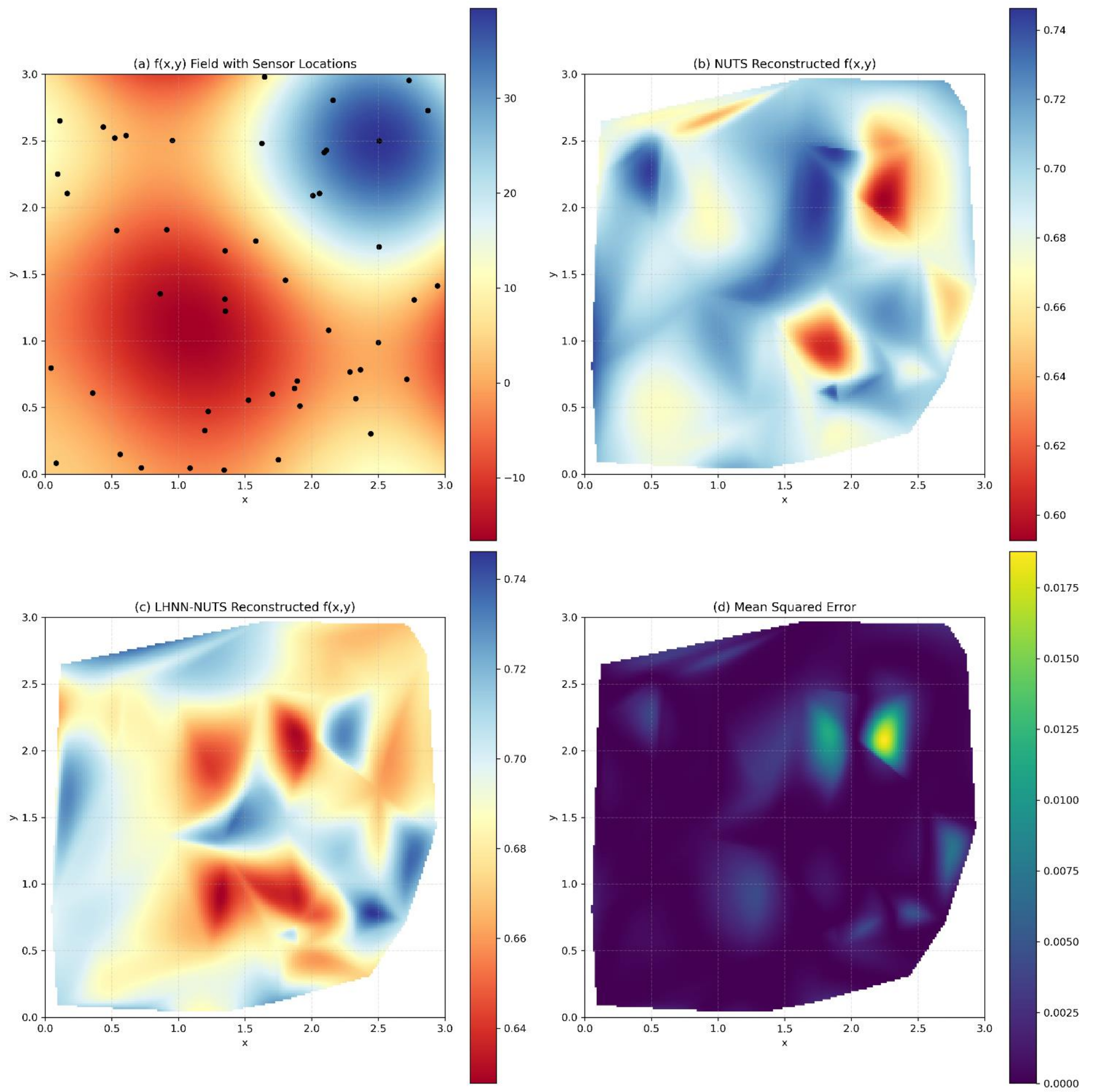
$$U(\mu) = \frac{1}{2}\sum_{i=1}^{50}(f_{obs} - \mu_i)^2$$

to be the potential energy to be used. After updating each μ_i, which is the position of a sensor, $f(\mu_i)$ can be calculated as

$$f(\mu_i) = \exp\left(-\frac{1}{2}(f_{obs} - \mu_i)^2\right)$$

This method does not work as well. Below is the result of an attempt.

Replication of Figure 12 (Failed)



2.4 TFP codes

The TFP codes on HMC do help a lot in implementing the replication. It gives clear and concise short examples on how HMC is conducted in its description of functions, and the logic of its complete codes of HMC also help organize my thought. However, I chose not to use HMC in TFP, because aside from traditional HMC, LHNN-HMC is also implemented in this replication. I wrote my own script of traditional HMC, in order to exactly match the logic in the LHNN-HMC script, so that the traditional HMC and LHNN-HMC scripts are consistent with each other, with same inputs, structures, and outputs.

Instead, I used API in TFP to calculate effective sample size (ESS), just as Dhulipala (2023) has displayed in his Pytorch code.

2.5 Replication Materials

Main codes are written in Python 3 with TensorFlow for vectorization management and are stored in the folder called *codes* under the root directory.

functions.py contains Hamiltonian functions to be studied.

utils.py contains leapfrog algorithms and other frequently used functions.

Parameters used in this replication project are all stored in *get_args.py*.

nn_models.py is used to generate Multilayer Perception and *hnn.py* is used to package MLP in *nn_models.py* and generate L-HNNs.

To train the model, run *train_hnn.py*.

hnn_hmc.py and *traditional_hmc.py* performs L-HNN HMC and traditional HMC. *hnn_nuts_test.py* is used to perform both L-HNNs NUTS and traditional NUTS.

Run *Figure2.py, Figure3.py, Figure4_5_6.py, Table1.py, Figure7_table2.py, Figure8_table3.py, Figure9_table3.py, Figure10_table3.py, Figure11_table4.py* and *Figure12_table4.py* to replicate the respective results presented in Dhulipala (2023).

Weights of networks and generated data are stored in folder *files*. Logs are stored in folder *logs*. Figures are saved into folder *figures*.

Part III Testing

Test codes are written in Python 3 with TensorFlow and unittest. Test codes are stored in the folder called *tests* under the root directory.

3.1 Testing plans

3.1.1 test_utils

The *test_utils.py* file tests utility functions from utils.py, covering Hamiltonian dynamics, leapfrog integration, loss computation, data serialization, logging, and effective sample size (ESS) calculations. Below is a concise test plan for the file.

The *dynamics_fn* function is tested to ensure it computes derivatives with the correct shape for various distributions (e.g., 1D Gaussian mixture, Neal Funnel). The outputs are checked for finiteness and validity across different dimensions.

The *traditional_leapfrog* function is tested for generating trajectories and derivatives with correct shapes, ensuring trajectory continuity and Hamiltonian conservation for conservative systems. The tests validate its behavior

for different input shapes, time spans, and step types, including edge cases like negative or zero steps, which should raise errors.
The L2_loss function is tested for correctness, ensuring the loss is a scalar, positive, and matches the expected value precisely.

The *to_pickle* and *from_pickle* functions are tested by saving and reloading data to confirm they preserve data integrity during serialization and deserialization.

The *setup_logger* function is tested to ensure it creates a logger with two handlers (console and file) and properly cleans up generated logs.

The *compute_ess* and *compute_average_ess* functions are tested to confirm they produce valid ESS values for even-dimensional inputs and raise errors for odd dimensions. The outputs are checked to ensure they are positive and correctly sized.

3.1.2 test_hnn_utils

The *test_hnn_utils.py* file tests utility functions from utils.py and their integration with the Hamiltonian Neural Network (HNN) framework. It covers gradient computation, training loss, leapfrog integration, MLP and HNN models, and error handling. Below is the test plan.

The *compute_gradients* function is tested to ensure it computes gradients for various densities, with output dimensions matching half the input dimensions. The outputs are validated for finiteness, and errors are raised for invalid inputs.

The *compute_training_loss* function is tested to ensure it computes a scalar, non-negative loss for different densities, comparing predicted and true gradients accurately.

The *leapfrog* function is tested for generating correctly shaped trajectories and time steps, ensuring continuity and Hamiltonian conservation for conservative systems. It is validated for different input shapes, step types, and time spans, with errors raised for invalid inputs.

The *MLP* model is tested for correct output dimensions and finite values with various activation functions. Invalid activations are checked to raise errors.

The *HNN* model is tested for accurate computations of kinetic energy, Hamiltonian, time derivatives, and adherence to Hamiltonian mechanics. Custom mass matrices and error handling for invalid inputs are also tested.

3.1.3 test_data

The *test_data.py* file tests data generation functions from the data module, specifically *get_trajectory* and *get_dataset*. It verifies their correctness for generating trajectories, datasets, and saved files across different configurations and distributions.

The *get_trajectory* function is tested for producing trajectories and derivatives with correct dimensions for various distributions (e.g., Rosenbrock, Neal Funnel). It validates the number of time steps, ensures trajectory continuity, and checks if outputs are finite. Custom parameters like time span, time step size, and initial conditions are also tested.

The *get_dataset* function is tested to ensure datasets are generated with the correct structure and dimensions. It checks if training and testing splits are correctly calculated and the expected number of data points matches the configuration. Saved files are verified by reloading and comparing them to the original dataset. The tests also ensure all data values are finite and consistent.

Temporary directories for saving data are created and cleaned up after each test, ensuring no residual files remain. Errors during file operations or data inconsistencies are also monitored.

3.1.4 test_functions

The *test_functions.py* file tests mathematical operations implemented in functions.py, specifically *nearest_neighbor_derivative*, *compute_f_hat_with_nearest_neighbor*, and *f_obs*. It validates their correctness, dimension handling, and robustness to invalid inputs.

The *nearest_neighbor_derivative* function is tested to ensure it produces derivatives with correct dimensions for valid inputs. It validates the finiteness of the output and checks error handling for invalid cases, such as missing batch dimensions or mismatched input sizes.

The *compute_f_hat_with_nearest_neighbor* function is tested for correct output dimensions and finiteness. It ensures the function handles invalid inputs properly, including missing batch dimensions and mismatched input sizes. The computed values (*f_hat*) are checked to confirm they remain within a reasonable range (e.g., clipped to ±200).

The *f_obs* function is used as a data provider for testing, and its outputs are validated indirectly through the other functions.

3.1.5 test_traditional_hmc

The *test_traditional_hmc.py* file tests the implementation of the TraditionalHMC class, which performs Hamiltonian Monte Carlo (HMC) sampling. It validates initialization, sampling, trajectory management, energy conservation, and error handling across a variety of configurations and distributions.

The HMC initialization is tested by verifying the correct setup of parameters, such as *state_dim, total_dim, step_size, trajectory_length*, and *trajectory storage*. Tests ensure proper initialization for various distributions and configurations.

The HMC sampling method is tested to ensure that it generates samples with the correct dimensions for multiple chains and post-burn-in acceptance rates. The finiteness of generated samples is validated, and acceptance rates are checked to ensure they remain within a valid range (0 to 1), with mean values in a reasonable range (e.g., 0.2–0.9).

The trajectory storage functionality is tested to confirm that trajectories are correctly stored for each non-burn-in sample. The number and dimensions of stored trajectories are validated, ensuring they match the expected trajectory count and time step counts derived from trajectory_length and step_size.

Energy conservation is tested by checking the Hamiltonian energy values at the initial and final states of each trajectory. The energy difference is validated to ensure it remains small, demonstrating the numerical stability of the HMC implementation.

Error handling is thoroughly tested for invalid configurations: Negative step size or trajectory length. Invalid input dimensions (e.g., negative or odd values). Extreme step sizes causing numerical instability. Invalid sample count or burn-in count exceeding the total sample count.

3.1.6 test_hnn_hmc

The *test_hnn_hmc.py* file tests the implementation of the HNNSampler class, which combines Hamiltonian Neural Networks (HNN) with Hamiltonian Monte Carlo (HMC) sampling. It validates initialization, HMC sampling, trajectory storage, energy conservation, and error handling across a variety of configurations and distributions.

The HNN-HMC initialization is tested to ensure the sampler correctly initializes parameters such as *state_dim, total_dim, step_size,* and *trajectory_length*. Weight files for the HNN model are verified to exist, and the sampler's HNN model structure is validated, ensuring it uses the correct HNN and MLP models with proper dimensions.

The HMC sampling process is tested for generating samples with the correct dimensions for multiple chains and post-burn-in acceptance rates. The finiteness of generated samples is validated, and acceptance rates are checked to ensure they fall within the range of 0 to 1, with mean acceptance rates tested to confirm reasonable values (e.g., >0.1).

The trajectory storage functionality is tested to confirm that trajectories are stored correctly for each non-burn-in sample. The number of stored trajectories is validated, and their dimensions are checked to ensure consistency with the calculated number of time steps based on trajectory_length and step_size.

Energy conservation is tested by validating the Hamiltonian energy computed by the HNN model at the initial and final states of each trajectory. The energy difference is checked to ensure it remains small, demonstrating numerical stability and the accuracy of the HNN model.

Error handling is tested for various invalid configurations, including: Negative or invalid step_size, trajectory_length, input_dim, or latent_dim. Non-existent weight files for the HNN model. Invalid paths or configurations causing TensorFlow exceptions.

Edge cases, such as invalid parameter combinations or extreme input values, are tested to ensure meaningful errors are raised.

3.1.7 test_hnn_nuts

The *test_hnn_nuts.py* file tests the No-U-Turn Sampler (NUTS) implementation integrated with Hamiltonian Neural Networks (HNN). It validates the correctness and robustness of NUTS sampling in both HNN-based and traditional modes, ensuring energy conservation, proper gradient computations, recursive tree building, and error handling. Below is the test plan.

The NUTS sampling modes are tested for both HNN-based and traditional methods. The tests validate that: Outputs (samples) have the correct dimensions based on the number of chains, total samples, and input dimensions. Acceptance rates are reasonable (greater than 0.1). Gradient computations occur as expected, with HNN mode reducing gradient evaluations compared to traditional mode. Errors are tracked for each sample, ensuring they remain within acceptable bounds.

The energy conservation test verifies that for both HNN and traditional modes, the Hamiltonian energy differences between consecutive samples in a chain remain within predefined thresholds specific to each distribution. This ensures the numerical stability and accuracy of the sampling process.

The gradient counting test ensures that the number of gradients computed during HNN sampling is lower than during traditional sampling. This validates the efficiency of the HNN-based method while maintaining correct sampling behavior.

The build_tree function is tested comprehensively to check its behavior at different recursion depths and for both directions of tree growth:

- The test validates the dimensions and types of the function's outputs, including positions (q_minus, q_plus, q_prime), momenta (p_minus, p_plus, p_prime), and scalar values (e.g., n_prime, alpha, error, grad_count).
- It ensures energy conservation at depth 0 and verifies valid trajectory growth for both leftward and rightward directions.
- Tree growth conditions, stopping criteria, and trajectory validity are tested for correctness.

The error handling tests ensure that appropriate exceptions are raised for invalid configurations, such as:

- Negative or zero values for nuts_step_size, total_samples, burn_in, or num_chains.
- burn_in exceeding total_samples.
- Invalid cooldown period (n_cooldown) or negative input dimensions.

3.2 Testing results

Most tests are passed, except for one certain type relating to energy conservation. In 3.1.6 test_hnn_hmc, the tests of energy conservation for densities, including 5D-ill conditioned Gassian density and 10D-Rosenbrock density, failed. This result indicates that HNN HMC does not have the property of Hamiltonian conservation. This is an expected result. In Chapter 4.2, the author uses figure 4 to illustrate that limited training data is available in the regions of low probability density. When such a region of low probability density is reached, large integration errors can arise, and that is why the author introduces online error monitoring mechanism.

The energy conservation in 3.1.7 test_hnn_nuts fails for 2-D Neal's funnel density. This can be a consequence of under-trained HNN, or could be out of a small threshold for checking energy conservation.

References

1. Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.
2. Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., & Rubin, D. B. (2013). *Bayesian Data Analysis* (3rd ed.). CRC Press.
3. Gelman, A., & Rubin, D. B. (1992). Inference from iterative simulation using multiple sequences. *Statistical Science*, 7(4), 457–472.
4. Hastings, W. K. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1), 97–109.
5. Hoffman, M. D., & Gelman, A. (2014). The No-U-Turn Sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15(1), 1593–1623.
6. Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., & Teller, E. (1953). Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6), 1087–1092.
7. Neal, R. M. (2011). MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo* (pp. 113–162). Chapman & Hall/CRC.