A replication of Bayesian Inference with Latent Hamiltonian Neural Networks

CHEN, Hongxiao

Department of Economics

The Chinese University of Hong Kong

Table of contents

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 point estimation, it provides an estimation of the distribution of underlying variables, 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 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.

In the paper to be replicated, Dhulipala et al. (2023) introduced a new method that improves the performance of NUTS. They use Hamilton Neural Networks (HNN) to simulate Hamiltonian and use auto-differentiation of HNN to find the gradient of Hamiltonian with respect to positions and momenta instead of computing gradients numerically. They firstly compare the trajectory from HMC and L-HNN HMC using 1-D Gaussian mixture density to show the result from L-HNN HMC is valid through figure 2 and 3. Then, they exhibited that LHNN-NUTS suffers from large integration error that causes NUTS to stop prematurely through figure 4 and introduce online error monitoring mechanism. Figure 5 and 6 shows that after introducing online error monitoring, the previous problem disappears, and L-HNN is more efficient than HNN.

Figure 7 uses 3-D Rosenbrock density to compare traditional NUTS and LHNN-NUTS with three sets of parameters. Similar applications are presented in Figure 8, 9, 10, 11 and 12, where the authors compare the implementation of NUTS and LHNN-NUTS for 2-D Neal's funnel density, 5-D ill conditioned Gaussian, 10-D degenerate Rosenbrock density, stationary distribution of Allen-Cahn stochastic PDE and elliptic PDE.

1.2 Methods

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.

1.2.1 Allen-Cahn and Elliptic PDE

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 discretize 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 discretize 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.

1.2.2 TensorFlow Probability

I modified the source of TensorFlow Probability to perform traditional HMC, LNN-HMC, traditional NUTS, LHNN-NUTS without online monitoring and LHNN-NUTS with online monitoring. The logic is as below

First, I rewrite a new HNN Leapfrog Integrator class that inherits from tensorflow_probability.python.mcmc.internal.leapfrog_integrator.LeapfrogIntegrator. This new class is initialized by taking HNN and target log probability function as input. This integrator uses HNN to perform the one-step leapfrog in Dhulipala (2023) and uses the target log probability to calculate the log probability.

The LHNN-HMC inherits from kernel_base.TransitionKernel, and is initialized with HNN and target log probability function. The integrator is the customized HNN Leapfrog Integrator.

LHNN-NUTS inherits from kernel_base.TransitionKernel, and is initialized with HNN and the target probability. The integrator is the customized HNN Leapfrog Integrator. MULTINOMAIL_SAMPLE and GENERGALIZED_UTRUN are both set to false to implement Hoffman (2014)'s method.

LHNN-NUTS with online monitoring also inherits from kernel_base.TransitionKernel, and is initialized with HNN, Hamiltonian function, and the target log probability. In this transition kernel, two integrators are used. The first integrator performs leapfrog with HNN and the second integrator performs leapfrog with Hamiltonian function, while both integrators perform the one-step leapfrog.

Following Dhulipala, when the error from using HNN exceeds the max HNN error threshold, the methods switch back to traditional NUTS, and that is where the second integrator plays a role. The modification is mainly in _loop_build_sub_tree.

An increase in the number of traditional NUTS steps and whether to switch back to lHNN-NUTS are placed at where a tree-building is finished, so the realization of this part is in the main structure of _loop_tree_doubling.

The three new transition kernels, LHNN-HMC, LHNN-NUTS and LHNN-NUTS with online monitoring are all performed through standard Markov Chain Monte Carlo. Specifically, tfp.mcmc.sample_chain(), which lies in run_sampling in utils.py and, is decorated by TensorFlow decorator.

Also it is worth noting that TensorFlow Probability builds trees and subtrees by using tf.while_loop to implement NUTS, instead of using function recursion. This approach makes TensorFlow's static graph mode feasible, which significantly accelerates sample and is not possible in the original version that used function recursion.

Part II Answers to the questions.

2.1 Parts that are hard to understand

A 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.

Another part that worths more explanation is why we are training on the gradients of the Hamiltonian, but not Hamiltonian itself, nor letting the neural network simply output the two time derivatives. A short answer is that constructing HNN this way best ensures the symplectic structure.

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, and similar cases are shown in Table 2, 3 and 4. Accurately counting the gradients evaluated is the crucial part of replicating these tables.

The solution for counting the gradients is as follows: in HNN-NUTS, when switching to traditional method,

each leapfrog increases the counting by one, because in the one-step leapfrog in Dhulipala (2023), only $\partial H/\partial q(t+\Delta t)$ needs to be evaluated, and $\partial H/\partial q(t)$ comes from last sample.

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 the latent layer is set to 25. Similar issues exist 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 $j = -1$.

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

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

which the author mistakenly discretizes it as

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

and the correct form should be

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

Mistake 3:

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

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

with the 2-d Neal's funnel density

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

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

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

The derivation is as the following:

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

Taking negative log and neglecting constant term,

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

2.3 Replication results

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

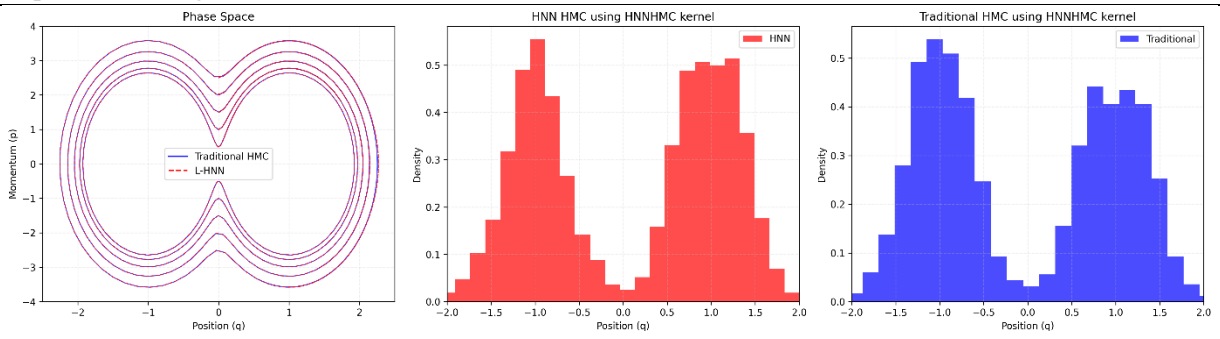
2.3.1 Replication for 1-D Gaussian mixture density

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. The left panel shows the property of time reversion for both traditional HMC and L-HNN HMC. The right panel of Figure 3 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).

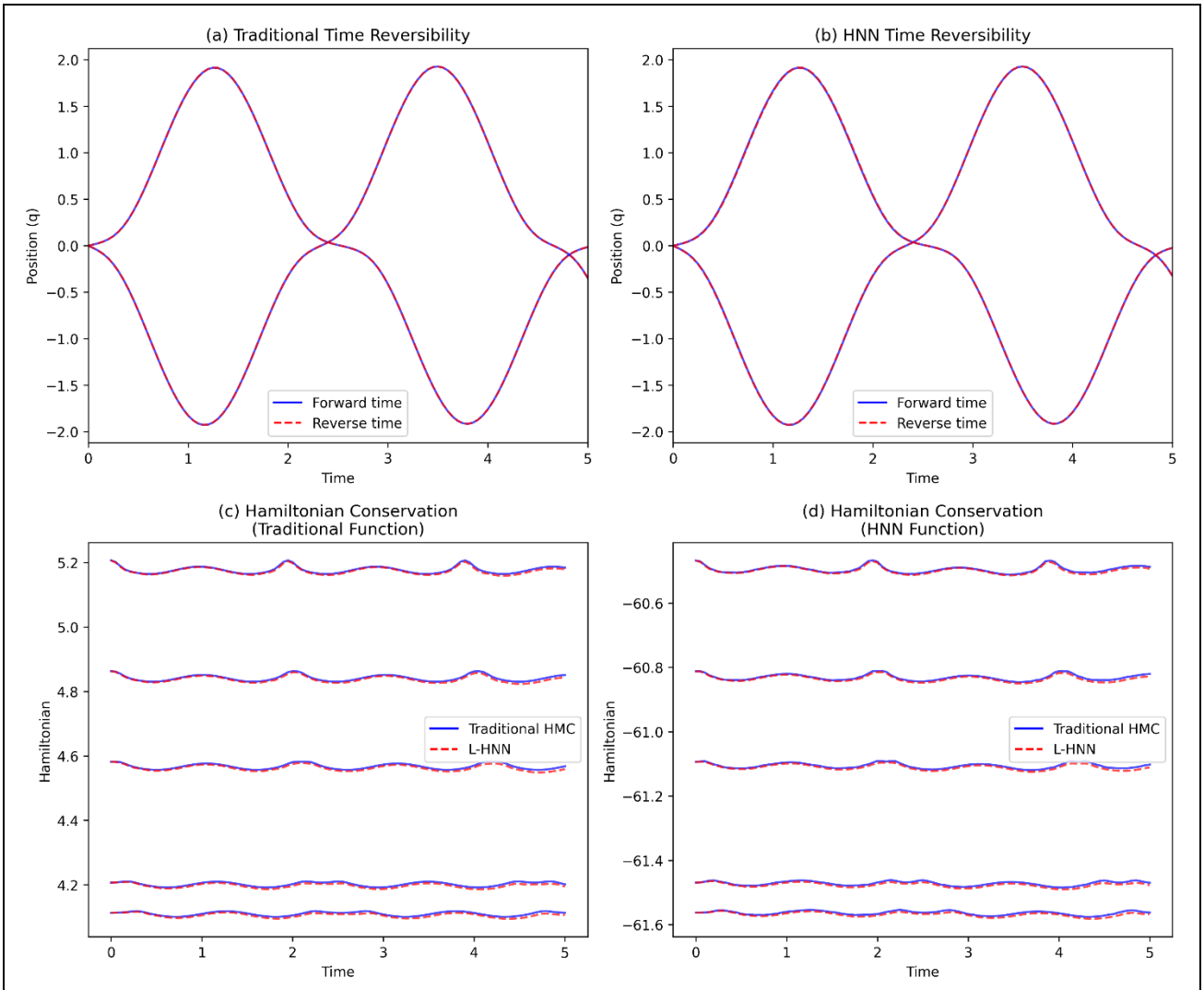
Note that the total number of gradients evaluated for L-HNN NUTS is 8,000 which equals the gradients evaluated in training. It suggests that the threshold is not reached, and the sampling is not switched to traditional NUTS in the case of 1-d Gaussian mixture density.

However, even under the condition of static graphs, the speed of NUTS is much slower than HMC, and the speeds of traditional HMC and L-HNN HMC are very close under static graphs. So I would not say it is very meaningful to compare the number of gradients evaluated.

Replication of Figure 2



Replication of Figure 3



Replication of Table 1

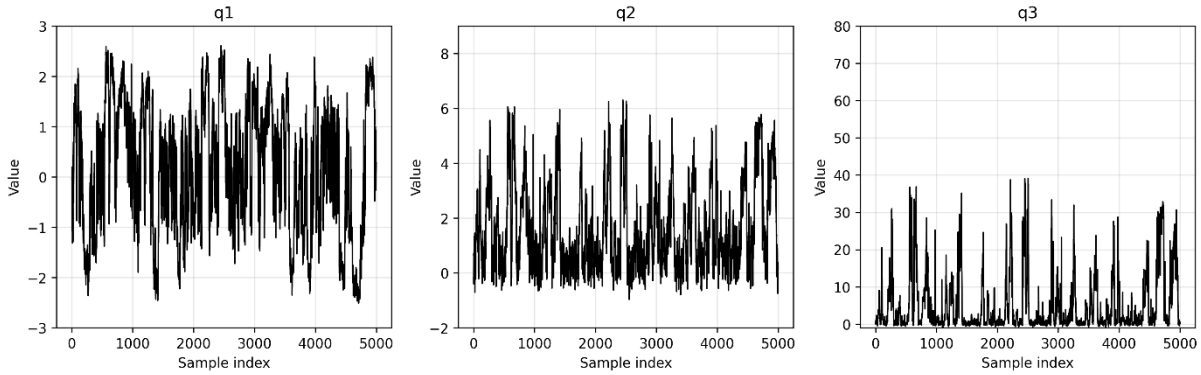
Method	ESS	# of Gradients	ESS/grad	Running Time
Traditional HMC	84.74	1000000	0.000847	62.29 seconds
L-HNN HMC	47.65	8000	0.005957	59.49 seconds
Tradtional NUTS	40.31	67918	0.000593	201.13 seconds
L-HNN NUTS	53.43	8000	0.006678	321.92 seconds

2.3.2 Replication for 3-D Rosenbrock density

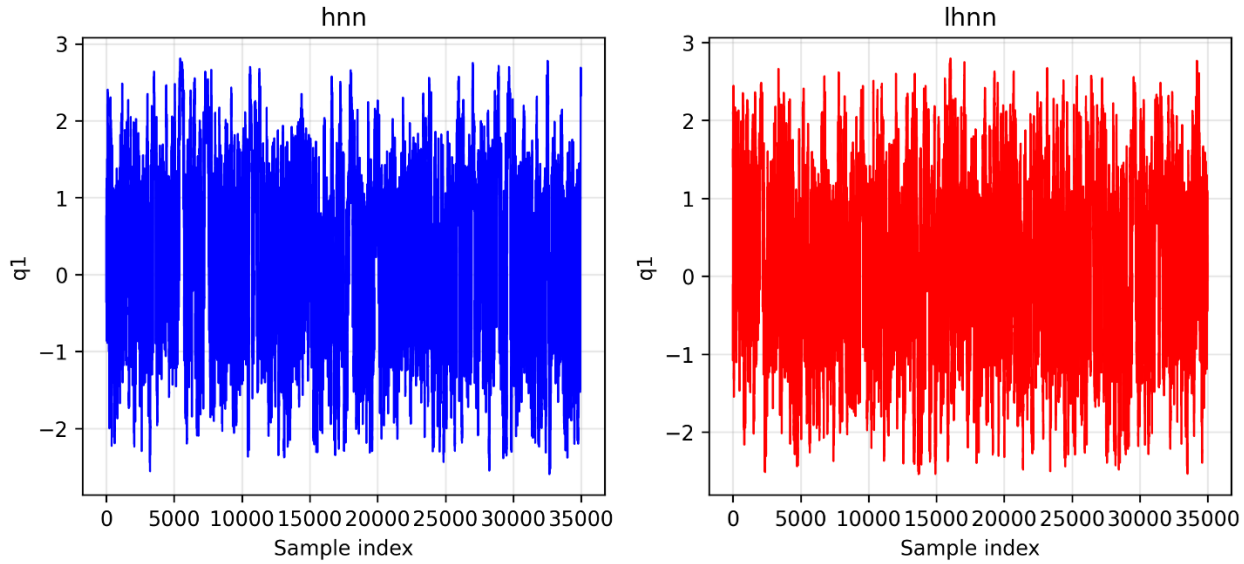
The second set of replications focuses on 3-D Rosenbrock density, which includes figure 4, 5, 6 and 7, as well as Table 2.

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.

Replication of Figure 4



Replication of Figure 5



Replication of Figure 6

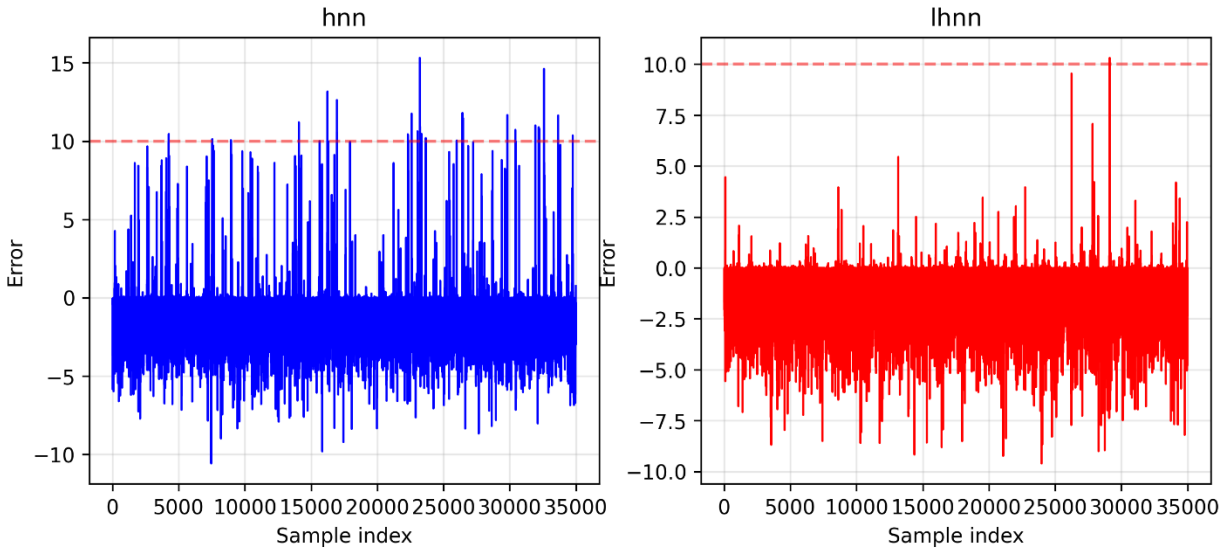
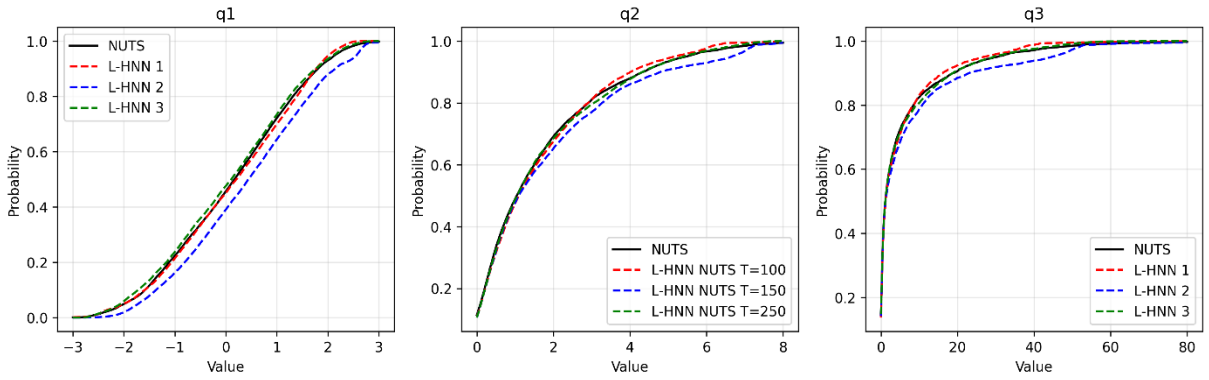


Figure 7 presents empirical cumulative functions plots for three positions in 3-D Rosenbrock density in order to

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.

Still, I don't think differentiating gradients evaluated in traditional NUTS and in LHNN NUTS is very meaningful, especially under static graphs.

Replication of Figure 7



Replication of Table 2

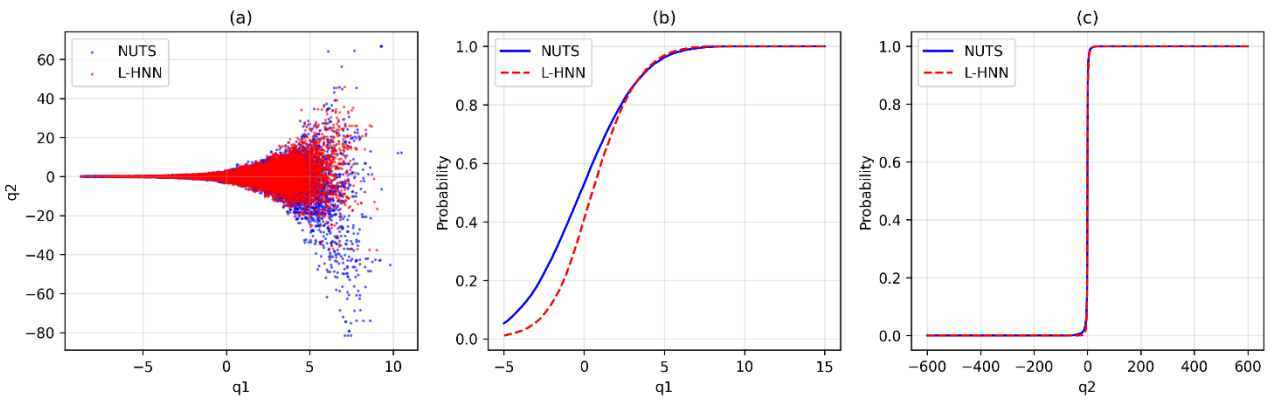
Method	ESS	# of Gradients	ESS/grad
NUTS	(210.40, 346.15, 354.21)	1957854	1.55e-4
[LHNN-NUTS 1] Mt = 40; T = 100	(236.87, 279.06, 271.64)	Evaluation: 96888 Training: 160,000	10.22e-4
[LHNN-NUTS 2] Mt = 40; T = 150	(148.57, 161.27, 139.37)	Evaluation: 66623 Training: 240,000	4.88e-4
[LHNN-NUTS 3] Mt = 40; T = 250	(139.46, 250.09, 259.61)	Evaluation: 1151 Training: 400,000	6.95e-4

2.3.3 Replication for Chapter 6

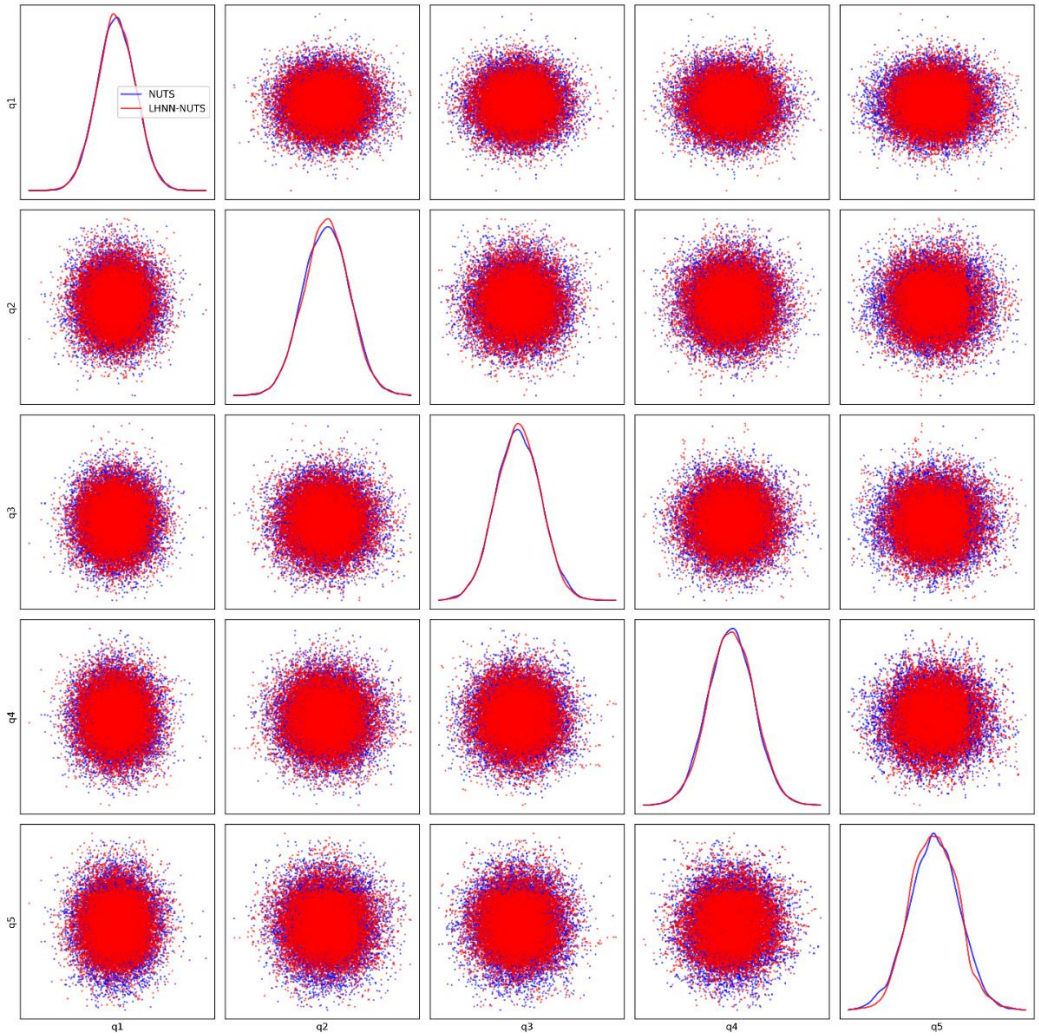
Figure 8, 9 and 10 display the application of LHNN-NUTS and NUTS on 2-D Neal's funnel density, 5-D ill conditioned Gaussian and 10-D degenerate density. I simulate 20,000 samples for 2-D Neal's funnel density with 5000 samples as burn-in, 20,000 samples for 5-D ill conditioned Gaussian with 5,000 samples 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.

It should be noticed that my result for figure 8 is different from the one in Dhulipala (2023), because I am using a different log probability, as what I have shown in mistake 3.

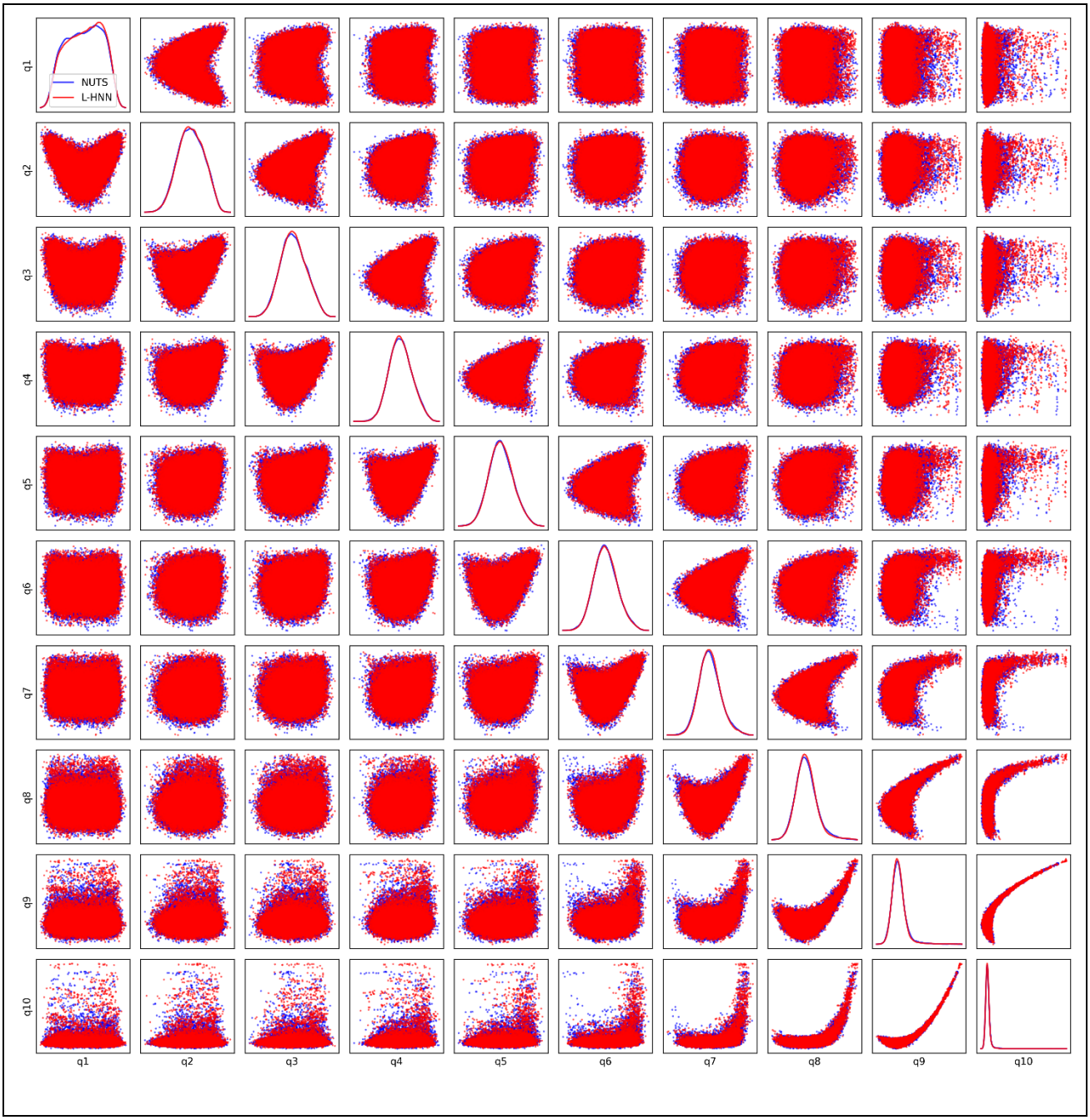
Replication of Figure 8



Replication of Figure 9



Replication of Figure 10



Replication of Table 3

Method	ESS	# of Gradients	ESS/grad
2-D Neal's funnel			
NUTS	(1248.58, 1033.36)	3704376	0.000308
LHNN-NUTS	(1410.72, 2445.27)	Evaluation: 739181 Training: 400000	0.001692
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,	Evaluation: 10273704	0.000816

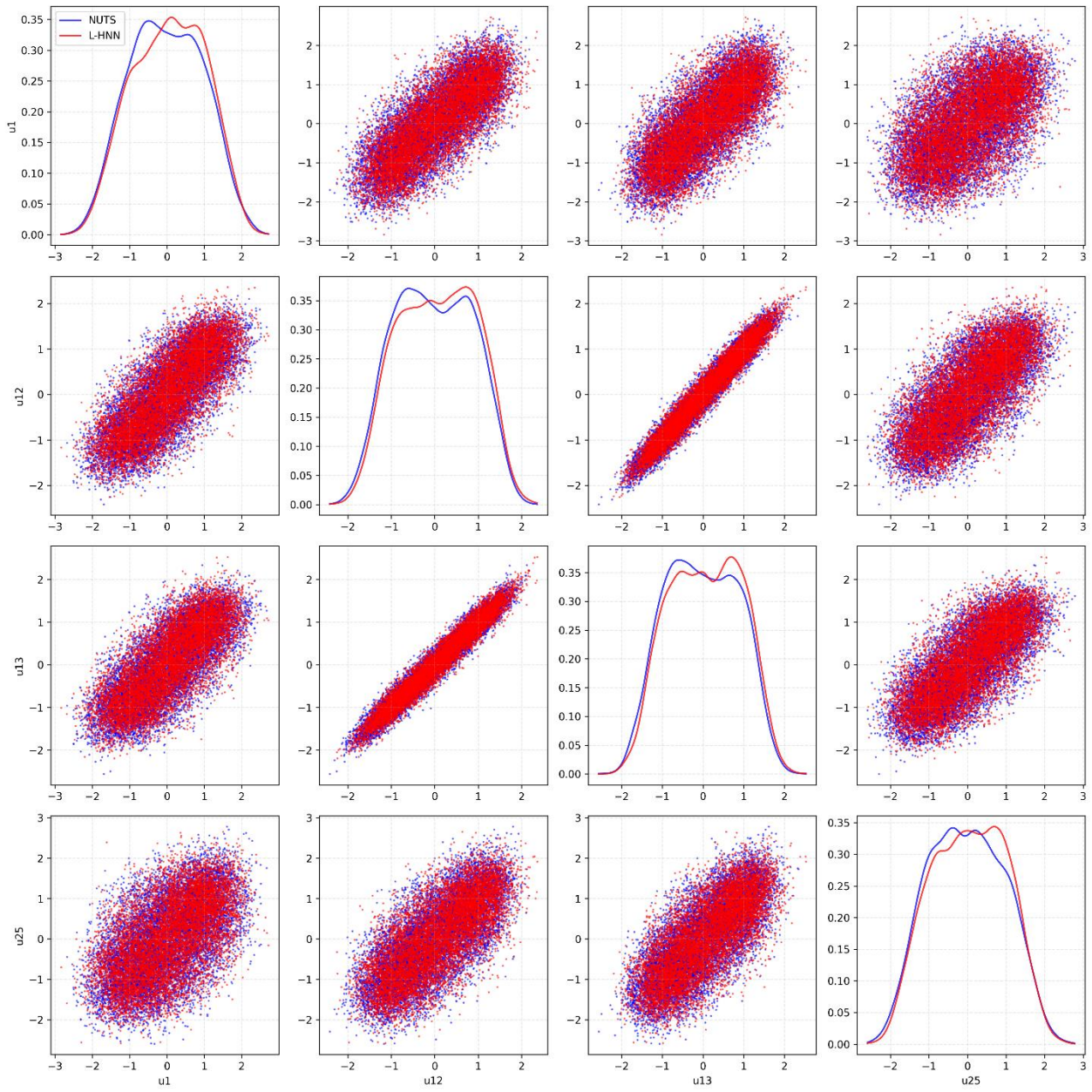
	9332.47, 4630.92, 1639.86)	Training: 400000 10673704	
10-D degenerate Rosenbrock			
NUTS	(6168.97, 11784.13, 11880.01, 12486.34, 10990.00, 9135.07, 6708.66, 5139.51, 3106.68, 1595.47)	2661160	0.002968
LHNN-NUTS	(4480.35, 7843.12, 7255.15, 7832.61, 7445.53, 6168.70, 4210.76, 2934.36, 1617.22, 928.27)	Evaluation: 1733243 Training: 400000	0.002377

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. 10000 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	877.57	1248272	0.000703
LHNN-NUTS	629.82	Evaluation: 103679 Training: 10000	0.00554

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 $g(x, y) = k(x, y)\frac{\partial u}{\partial x}$, 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 $\mu = (\mu_1, \mu_2, \dots, \mu_{50})$ 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, 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.

2.4 TFP codes

Details on how TFP are implemented is described in part I, 1.2 Methods.

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.

Source codes of TFP that are modified are stored in *codes/tfp_modified_kernels*, which includes *hnn_leapfrogs.py, tfp_hnn_hmc.py, tfp_hnn_nuts.py, tfp_hnn_nuts_online.py*

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*.

To replicate the results, use *run.bash.*

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 *compute_ess* function is tested to confirm it produces valid ESS values for different input dimensions. The outputs are checked to ensure they are positive and correctly computed.

The *hamiltonian_wrapper* function is tested to verify it returns scalar outputs for both single and batch inputs.

The *hnn_wrapper* function is tested using a simple neural network model, ensuring it correctly processes both single and batch inputs while maintaining the expected output shape.

3.1.2 test_hnn

The *test_hnn.py* file tests the HNN (Hamiltonian Neural Network) and MLP (Multi-Layer Perceptron) models, focusing on neural network behavior, Hamiltonian computations, and model loading. Below is a concise test plan for the file. 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 MLP model is tested for different activation functions (sine, tanh, relu), ensuring correct output shapes and finiteness. It also verifies that an invalid activation function raises an error.

The HNN model is tested by loading various configurations to check its ability to compute kinetic energy and Hamiltonians. The outputs are validated for correct shapes and expected properties (e.g., kinetic energy being non-negative).

The mass matrix functionality in HNN is tested to ensure the default matrix is positive and a custom mass matrix is correctly assigned.

The model loading process is tested to verify weight file existence and correct handling of missing or corrupt files.

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. The files saved 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.

get_target_log_prob is the key function of *functions.py.* It converts Hamiltonian function to target log probability, and serves as the most important input to transition kernels. So an additional class *TestGetTargetLogProb(unittest.TestCase)* is created for testing this function. The class tests the *get_target_log_prob* function for correct log probability computation across different distributions, dimensions, and batch sizes. It ensures proper handling of invalid distributions and verifies input type conversion, including support for numpy arrays.

3.2 Testing results

All tests are successfully passed.

Part IV Summary

In this article, I replicate the work of Dhulipala (2023) using TensorFlow and TensorFlow Probability, and write unit tests for functions that help realize the replication. Among 5 tables and 11 figures, I replicate 4 tables and 10 figures. The part that I failed to replicate is in Chapter 7.2 of Dhulipala (2023), which consider an elliptic partial differential equation in 2-D. The author discretizes the (x, y) space into 50 random sensor locations and infers the posterior using LHNN-NUTS.

The tests show that my implementation is valid, for the functions are able to take different dimensions of variables, different densities, and different type of data as input, and generate the expected and correct results.

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.