Part-I Report on "Hamiltonian Neural Network and Hamiltonian Monte Carlo" Topic

Yuxi Cai *caiyuxi@connect.hku.hk*

March 12, 2025

Contents

1 Paper understanding (answers to Question (b))

1.1 More explanations for paper

1.1.1 Properties of Hamiltonian dynamics

This subsection elaborates on some concepts mentioned in Section 2 of the paper.

Time reversibility. Time reversibility refers to that "the mapping T_s from the state at time t, $(\boldsymbol{q}(t), \boldsymbol{p}(t))$, to the state at time $t+s$, $(\boldsymbol{q}(t+s), \boldsymbol{p}(t+s))$, is one-to-one, and hence an inverse, T_{-s}" (Neal et al., 2011). The time reversibility of Hamiltonian dynamics is necessary for the reversibility of Markov chain transitions, where the latter can be used to prove the MCMC updates with Hamiltonian dynamics leave the desired distribution invariant.

Conservation of the Hamiltonian The Hamiltonian dynamics keep the Hamiltonian invariant which can be verified by examining its derivative with respect to time:

$$\frac{\mathrm{d}H}{\mathrm{d}t} = \sum_{i=1}^{d} \left[\frac{\mathrm{d}H}{\mathrm{d}q_i} \frac{\mathrm{d}q_i}{\mathrm{d}t} + \frac{\mathrm{d}H}{\mathrm{d}p_i} \frac{\mathrm{d}p_i}{\mathrm{d}t} \right] = \sum_{i=1}^{d} \left[\frac{\mathrm{d}H}{\mathrm{d}q_i} \frac{\mathrm{d}H}{\mathrm{d}p_i} - \frac{\mathrm{d}H}{\mathrm{d}q_i} \frac{\mathrm{d}H}{\mathrm{d}p_i} \right] = 0,$$

where the second equality utilizes Equation (4) in the paper. Note that in practice, Hamiltonian is only approximately invariant due to the numerical error and thus will not make the acceptance probability always equal to one in Metropolis updates (Neal et al., 2011).

Volume preservation. Volume preservation refers to that if the mapping T_s is applied to the points in some region R of $(\boldsymbol{q}, \boldsymbol{p})$ space that has a volume V, then the image of R under T_s will also have volume V (Neal et al., 2011). Sometimes, volume preservation can be equivalent to the determinant of the Jacobian matrix of T_s, denoted as $\boldsymbol{B}_s$, having an absolute value of one.

Symplecticness. The symplecticness condition states that the Jacobian matrix of T_s, denoted as $\boldsymbol{B}_s$, satisfies $\boldsymbol{B}_s^\top \boldsymbol{J}^{-1} \boldsymbol{B}_s = \boldsymbol{J}^{-1}$, where $\boldsymbol{J}$ is defined in Equation (3) of the pa-

per. The symplecticness further implies volume preservation as the equation indicates that $[\det(\boldsymbol{B}_s)]^2 = 1$.

1.1.2 Detailed balance and invariant distribution

Detailed balance is important because it can be used to prove the Metropolis update leaves the canonical distribution for $\boldsymbol{q}, \boldsymbol{p}$ invariant. The concept has also appeared in the NUTS paper (Hoffman and Gelman, 2014) which will be introduced later. Suppose the space of $(\boldsymbol{q}, \boldsymbol{p})$ is partitioned into regions A_k, each with volume V. Denote the image of A_k with respect to N leapfrog steps and a negation of $\boldsymbol{p}$ as B_k, which also has volume V because of the volume preservation of leapfrog. Then detailed balance holds if for any i, j, $P(A_i)T(B_j|A_i) = P(B_j)T(A_i|B_j)$, where $P(\cdot)$ is the probability under the canonical distribution, and $T(X|Y)$ is the probability of proposing and moving to a state in X given the current state is in Y. With the detailed balance condition, we can show the Metropolis update with HMC leaves the canonical distribution by the following. Consider the probability of the next state being in region B_k:

$$
\begin{aligned}
P(B_k) &= P(B_k)R(B_k) + \sum_j P(A_j)T(B_k|A_j) && (1)\\
&= P(B_k)R(B_k) + \sum_j P(B_k)T(A_j|B_k) && (2)\\
&= P(B_k)R(B_k) + P(B_k)(1 - R(B_k)) = 1
\end{aligned}
$$

where $R(B_k)$ denotes the probability of rejecting the proposed state when current state is in B_k. Equation (1) utilizes the fact that there are only two scenarios: the current state is in B_k and the next state remains in B_k because of the proposal rejection, and moving from another region to B_k. The detailed balance condition is applied in Equation (2).

1.1.3 No-U-Turn Sampling (NUTS)

This subsection elaborates more on the NUTS mentioned in Section 4 of the paper based on the original paper that proposed the method (Hoffman and Gelman, 2014). The choice of the end time T in MCMC is important. If T is too small, subsequent samples will be near each other, leading to unfavorable random walk behavior and slow mixing; in contrast, if T is too large, the trajectories generated by HMC will circle back and retrace previous steps, wasting computations (Hoffman and Gelman, 2014). Even worse when T happens to be near the periodicity for some state, the trajectory will be close to the starting point and thus slow down the HMC considerably. To address the issue, the NUTS presents a way to eliminate the need to manually tune T.

Main idea. An intuitive stopping criterion is to stop when the distance between the proposal $\boldsymbol{q}'$ and the initial state $\boldsymbol{q}$ no longer increases. The derivative of the half of the squared distance between $\boldsymbol{q}'$ and $\boldsymbol{q}$ with respect to time can be expressed as

$$\frac{\mathrm{d}(\boldsymbol{q}'-\boldsymbol{q})^\top(\boldsymbol{q}'-\boldsymbol{q})}{\mathrm{d}t} = (\boldsymbol{q}'-\boldsymbol{q})^\top\boldsymbol{p}',$$

and thus we may wish to stop when this quantity is smaller than zero. However, this leads to a non-reversible MCMC, calling for a more calibrated scheme. Overall, there are four main components in NUTS: (i) the introduction of a slice variable u, (ii) the construction of a balanced binary tree $\mathcal{B}$ with the doubling process and repeatedly checking the stopping criteria, (iii) the construction of a candidate set $\mathcal{C}$ from $\mathcal{B}$, and (iv) choose the next sample $(\boldsymbol{q}', \boldsymbol{p}')$ by uniformly sampling from $\mathcal{C}$. Importantly, the i-th iteration of NUTS can be summarized into four steps:

1. sample $\boldsymbol{p} \sim N(\boldsymbol{0}, \boldsymbol{I})$;
2. sample $u \sim \text{Uniform}(0, \exp(H(\boldsymbol{q}^i, \boldsymbol{p})))$;
3. sample $\mathcal{B}, \mathcal{C}$ from $p(\mathcal{B}, \mathcal{C}|\boldsymbol{q}^i, \boldsymbol{p}, u, \epsilon)$;

4. sample $\boldsymbol{q}^{i+1}, \boldsymbol{p}$ from $T(\boldsymbol{q}', \boldsymbol{p}' | \boldsymbol{q}^i, \boldsymbol{p}, \mathcal{C})$,

where $T(\boldsymbol{q}', \boldsymbol{p}' | \boldsymbol{q}, \boldsymbol{p}, \mathcal{C})$ must be a transition kernel that leaves the uniform distribution on $\mathcal{C}$ invariant. Note that $(\boldsymbol{q}, \boldsymbol{p})$ uniform on $\mathcal{C}$ given $u, \mathcal{B}, \mathcal{C}, \epsilon$ is a result from the conditions in the coming paragraph.

Conditions for $p(\mathcal{B}, \mathcal{C} | \boldsymbol{q}, \boldsymbol{p}, u, \epsilon)$. To ensure the detailed balance holds, Hoffman and Gelman (2014) imposes four conditions on $p(\mathcal{B}, \mathcal{C} | \boldsymbol{q}, \boldsymbol{p}, u, \epsilon)$. They are

C.1 All elements of $\mathcal{C}$ must be chosen in a way that preserves volume;

C.2 $p((\boldsymbol{q}, \boldsymbol{p}) \in \mathcal{C} | \boldsymbol{q}, \boldsymbol{p}, u, \epsilon) = 1$;

C.3 $p(u \leq \exp(H(\boldsymbol{q}', \boldsymbol{p}')) | (\boldsymbol{q}', \boldsymbol{p}') \in \mathcal{C}) = 1$;

C.4 If $(\boldsymbol{q}, \boldsymbol{p}) \in \mathcal{C}$ and $(\boldsymbol{q}', \boldsymbol{p}') \in \mathcal{C}$, then for any $\mathcal{B}$, $p(\mathcal{B}, \mathcal{C} | \boldsymbol{q}, \boldsymbol{p}, u, \epsilon) = p(\mathcal{B}, \mathcal{C} | \boldsymbol{q}', \boldsymbol{p}', u, \epsilon)$.

Construction of $\mathcal{B}$. The deterministic construction of $\mathcal{B}$ defines $p(\mathcal{B} | \boldsymbol{q}, \boldsymbol{p}, u, \epsilon)$, where ϵ is the step size. At the current height of the tree, denoted as j, the doubling proceeds by sampling a random direction $v_j \sim U(\{-1, 1\})$ ($v_j = 1$ and -1 stands for forward and backward steps, respectively) and taking 2^j leapfrog steps of size $v_j \epsilon$. The construction of $\mathcal{B}$ will stop at height j in the following two scenarios:

- The trajectory makes "U-turns": the states corresponding to the leftmost and rightmost leaves for one of the $2^j - 1$ subtrees (non-zero height) satisfies

$$(\boldsymbol{p}^+ - \boldsymbol{p}^-)^\top \boldsymbol{q}^- < 0 \quad \text{or} \quad (\boldsymbol{p}^+ - \boldsymbol{p}^-)^\top \boldsymbol{q}^+ < 0. \tag{3}$$

- The integration error becomes too large: if a tree includes a leaf node whose states satisfies

$$H(\boldsymbol{q}, \boldsymbol{p}) + \log u > \Delta_{\max}, \tag{4}$$

where $\Delta_{\max}$ is a predefined threshold.

Construction of $\mathcal{C}$ from $\mathcal{B}$. The construction of $\mathcal{C}$ must satisfy the following to meet the conditions mentioned previously:

- (fulfilling Condition C.2) the initial state $(\boldsymbol{q}, \boldsymbol{p}) \in \mathcal{C}$;
- (fulfilling Condition C.3) exclude $(\boldsymbol{q}', \boldsymbol{p}')$ if $\exp(-H(\boldsymbol{q}', \boldsymbol{p}')) < u$;
- (fulfilling Condition C.4) if the stopping conditions (3) and (4) are satisfied during the final doubling iteration, then any elements added during this iteration should be excluded from $\mathcal{C}$; if the stopping condition (4) is satisfied for the final full tree, then no element is excluded.

Note that Condition C.1 is automatically satisfied as the leapfrog step is volume preserving.

Naive NUTS Algorithm. Figure 1 is a copy of the naive version of NUTS in (Hoffman and Gelman, 2014). One can tell the construction of $\mathcal{B}$ is in a recursive way and the variable s records whether the stopping criteria are met or not. The construction of $\mathcal{C}$ is in an incremental way, following the previous explanations.

Efficient NUTS. The naive NUTS can lead to three potential concerns and thus a more efficient version is also proposed in (Hoffman and Gelman, 2014). Firstly, the naive version may waste time and computations if a stopping criterion is met during the doubling. Thus, in efficient NUTS, the recursion of "BuildTree" function is broken out once the stopping criterion is met, i.e., $s = 0$. Secondly, the naive version requires the storage of 2^j states $(\boldsymbol{q}, \boldsymbol{p})$ in memory as all the candidate states are kept until the sampling in the last step. Thirdly, the uniform sampling on $\mathcal{C}$ does not guarantee a large jump from the initial states. To mitigate the second and third problems, the efficient NUTS adopts a new transition kernel that still preserves the detailed balance and leaves the uniform distribution on $\mathcal{C}$ invariant.

Algorithm 2 Naive No-U-Turn Sampler

Given θ^0, ϵ, $\mathcal{L}$, M:
for $m = 1$ to M **do**
  Resample $r^0 \sim \mathcal{N}(0, I)$.
  Resample $u \sim \text{Uniform}([0, \exp\{\mathcal{L}(\theta^{m-1} - \frac{1}{2} r^0 \cdot r^0\}])$
  Initialize $\theta^- = \theta^{m-1}$, $\theta^+ = \theta^{m-1}$, $r^- = r^0$, $r^+ = r^0$, $j = 0$, $\mathcal{C} = \{(\theta^{m-1}, r^0)\}$, $s = 1$.
  while $s = 1$ **do**
    Choose a direction $v_j \sim \text{Uniform}(\{-1, 1\})$.
    if $v_j = -1$ **then**
      $\theta^-, r^-, -, -, \mathcal{C}', s' \leftarrow \text{BuildTree}(\theta^-, r^-, u, v_j, j, \epsilon)$.
    else
      $-, -, \theta^+, r^+, \mathcal{C}', s' \leftarrow \text{BuildTree}(\theta^+, r^+, u, v_j, j, \epsilon)$.
    end if
    if $s' = 1$ **then**
      $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$.
    end if
    $s \leftarrow s' \mathbb{I}[(\theta^+ - \theta^-) \cdot r^- \geq 0] \mathbb{I}[(\theta^+ - \theta^-) \cdot r^+ \geq 0]$.
    $j \leftarrow j + 1$.
  end while
  Sample θ^m, r uniformly at random from $\mathcal{C}$.
end for

function BuildTree$(\theta, r, u, v, j, \epsilon)$
if $j = 0$ **then**
  Base case—take one leapfrog step in the direction v.
  $\theta', r' \leftarrow \text{Leapfrog}(\theta, r, v\epsilon)$.
  $\mathcal{C}' \leftarrow \begin{cases} \{(\theta', r')\} & \text{if } u \leq \exp\{\mathcal{L}(\theta') - \frac{1}{2} r' \cdot r'\} \\ \emptyset & \text{else} \end{cases}$
  $s' \leftarrow \mathbb{I}[\mathcal{L}(\theta') - \frac{1}{2} r' \cdot r' > \log u - \Delta_{\max}]$.
  return $\theta', r', \theta', r', \mathcal{C}', s'$.
else
  Recursion—build the left and right subtrees.
  $\theta^-, r^-, \theta^+, r^+, \mathcal{C}', s' \leftarrow \text{BuildTree}(\theta, r, u, v, j - 1, \epsilon)$.
  if $v = -1$ **then**
    $\theta^-, r^-, -, -, \mathcal{C}'', s'' \leftarrow \text{BuildTree}(\theta^-, r^-, u, v, j - 1, \epsilon)$.
  else
    $-, -, \theta^+, r^+, \mathcal{C}'', s'' \leftarrow \text{BuildTree}(\theta^+, r^+, u, v, j - 1, \epsilon)$.
  end if
  $s' \leftarrow s' s'' \mathbb{I}[(\theta^+ - \theta^-) \cdot r^- \geq 0] \mathbb{I}[(\theta^+ - \theta^-) \cdot r^+ \geq 0]$.
  $\mathcal{C}' \leftarrow \mathcal{C}' \cup \mathcal{C}''$.
  return $\theta^-, r^-, \theta^+, r^+, \mathcal{C}', s'$.
end if

Figure 1: The naive NUTS algorithm in (Hoffman and Gelman, 2014), where the notations θ and r correspond to $\boldsymbol{q}$ and $\boldsymbol{p}$ in this report, respectively.

Specifically, the newly-adopted transition kernel is of the form

$$T(\boldsymbol{\omega}'|\boldsymbol{\omega},\mathcal{C}) = \begin{cases} \frac{\mathbb{I}\{\boldsymbol{\omega}'\in\mathcal{C}^{\text{new}}\}}{|\mathcal{C}^{\text{new}}|} & \text{if } |\mathcal{C}^{\text{new}}| > |\mathcal{C}^{\text{old}}| \\ \frac{|\mathcal{C}^{\text{new}}|}{|\mathcal{C}^{\text{old}}|}\frac{\mathbb{I}\{\boldsymbol{\omega}'\in\mathcal{C}^{\text{new}}\}}{|\mathcal{C}^{\text{new}}|} + \left(1-\frac{|\mathcal{C}^{\text{new}}|}{|\mathcal{C}^{\text{old}}|}\right)\mathbb{I}\{\boldsymbol{\omega}'=\boldsymbol{\omega}\} & \text{if } |\mathcal{C}^{\text{new}}| \le |\mathcal{C}^{\text{old}}| \end{cases}$$

where $\boldsymbol{\omega} = (\boldsymbol{q},\boldsymbol{p})$, $\boldsymbol{\omega}' = (\boldsymbol{q}',\boldsymbol{p}')$, $\mathcal{C}^{\text{new}}$ and $\mathcal{C}^{\text{old}}$ are disjoint, $\mathcal{C} = \mathcal{C}^{\text{new}} \cup \mathcal{C}^{\text{old}}$, and $\boldsymbol{\omega} \in \mathcal{C}^{\text{old}}$. Intuitively, T proposes a move from $\boldsymbol{w}$ to a random state in $\mathcal{C}^{\text{new}}$, which will be accepted with probability $|\mathcal{C}^{\text{new}}|/|\mathcal{C}^{\text{old}}|$. The transition kernel can be applied after every doubling, where the invariance of uniform distribution on partial $\mathcal{C}$ keeps the same property for the full $\mathcal{C}$. Furthermore, the memory consumption can be reduced from $\mathcal{O}(2^j)$ to $\mathcal{O}(j)$. It is because

$$p(\boldsymbol{\omega}|\mathcal{C}') = \frac{1}{|\mathcal{C}'|} = \frac{|\mathcal{C}_{\text{subtree}}|}{|\mathcal{C}'|}\frac{1}{|\mathcal{C}_{\text{subtree}}|} = p(\boldsymbol{\omega}\in\mathcal{C}_{\text{subtree}}|\mathcal{C})p(\boldsymbol{\omega}|\boldsymbol{\omega}\in\mathcal{C}_{\text{subtree}},\mathcal{C}).$$

This suggests that the sampling of $\mathcal{C}'$ can be conducted in an incremental and bottom-up approach. For example, for a subtree who has two smaller subtrees of size n_1 and n_2 with their respective representative being $\boldsymbol{\omega}_1$ and $\boldsymbol{\omega}_2$, we will choose $\boldsymbol{\omega}_1$ with a probability of $n_1/(n_1+n_2)$. The detailed algorithm can be found in Figure 2.

1.2 Typos in paper

Equation (15) in Section 3.1. The correct form of a hidden layer should be

$$\boldsymbol{u}_p = \phi(\boldsymbol{w}_p\boldsymbol{u}_{p-1} + \boldsymbol{b}_p) \quad \forall p \in \{1,\dots,P\}.$$

Equation (26) in Section 7.1. There is a typo when discretizing Equation (25) to derive Equation (26) if assuming the Trapezoidal rule is used. The correct form should be

$$\pi(\boldsymbol{u}) = \exp\left(-\sum_{i=0}^{d-1}\left(\frac{1}{2\Delta x}\left(u(i\Delta x+\Delta x)-u(i\Delta x)\right)^2 + \frac{\Delta x}{2}\left(V(u(i\Delta x+\Delta x))+V(u(i\Delta x))\right)\right)\right).$$

The same form can be found at Equation (22) in (Goodman and Weare, 2010) Besides, there is a contradiction involved here if letting $\boldsymbol{u} \in \mathbb{R}^{25}$ and $d = 25$ at the same time. When plugging $d = 25$ in the joint distriution above, the values of $u(0), u(1), \dots, u(24), u(25)$ are needed for calculation, suggesting that $\boldsymbol{u}$ is 26-dimensional. Hence, to be consistent with the paper results, I keep $\boldsymbol{u} \in \mathbb{R}^{25}$ but change d to 24 for the joint distribution.

Algorithm 3 Efficient No-U-Turn Sampler

Given θ^0, ϵ, $\mathcal{L}$, M:
for $m = 1$ to M **do**
  Resample $r^0 \sim \mathcal{N}(0, I)$.
  Resample $u \sim \text{Uniform}([0, \exp\{\mathcal{L}(\theta^{m-1} - \frac{1}{2} r^0 \cdot r^0\}])$
  Initialize $\theta^- = \theta^{m-1}$, $\theta^+ = \theta^{m-1}$, $r^- = r^0, r^+ = r^0, j = 0, \theta^m = \theta^{m-1}, n = 1, s = 1$.
  while $s = 1$ **do**
    Choose a direction $v_j \sim \text{Uniform}(\{-1, 1\})$.
    if $v_j = -1$ **then**
      $\theta^-, r^-, -, -, \theta', n', s' \leftarrow \text{BuildTree}(\theta^-, r^-, u, v_j, j, \epsilon)$.
    else
      $-, -, \theta^+, r^+, \theta', n', s' \leftarrow \text{BuildTree}(\theta^+, r^+, u, v_j, j, \epsilon)$.
    end if
    if $s' = 1$ **then**
      With probability $\min\{1, \frac{n'}{n}\}$, set $\theta^m \leftarrow \theta'$.
    end if
    $n \leftarrow n + n'$.
    $s \leftarrow s' \mathbb{I}[(\theta^+ - \theta^-) \cdot r^- \geq 0] \mathbb{I}[(\theta^+ - \theta^-) \cdot r^+ \geq 0]$.
    $j \leftarrow j + 1$.
  end while
end for

function BuildTree$(\theta, r, u, v, j, \epsilon)$
if $j = 0$ **then**
  Base case—take one leapfrog step in the direction v.
  $\theta', r' \leftarrow \text{Leapfrog}(\theta, r, v\epsilon)$.
  $n' \leftarrow \mathbb{I}[u \leq \exp\{\mathcal{L}(\theta') - \frac{1}{2} r' \cdot r'\}]$.
  $s' \leftarrow \mathbb{I}[\mathcal{L}(\theta') - \frac{1}{2} r' \cdot r' > \log u - \Delta_{\max}]$
  return $\theta', r', \theta', r', \theta', n', s'$.
else
  Recursion—implicitly build the left and right subtrees.
  $\theta^-, r^-, \theta^+, r^+, \theta', n', s' \leftarrow \text{BuildTree}(\theta, r, u, v, j - 1, \epsilon)$.
  if $s' = 1$ **then**
    if $v = -1$ **then**
      $\theta^-, r^-, -, -, \theta'', n'', s'' \leftarrow \text{BuildTree}(\theta^-, r^-, u, v, j - 1, \epsilon)$.
    else
      $-, -, \theta^+, r^+, \theta'', n'', s'' \leftarrow \text{BuildTree}(\theta^+, r^+, u, v, j - 1, \epsilon)$.
    end if
    With probability $\frac{n''}{n' + n''}$, set $\theta' \leftarrow \theta''$.
    $s' \leftarrow s'' \mathbb{I}[(\theta^+ - \theta^-) \cdot r^- \geq 0] \mathbb{I}[(\theta^+ - \theta^-) \cdot r^+ \geq 0]$
    $n' \leftarrow n' + n''$
  end if
  return $\theta^-, r^-, \theta^+, r^+, \theta', n', s'$.
end if

Figure 2: The efficient NUTS algorithm in (Hoffman and Gelman, 2014), where the notations θ and r correspond to $\boldsymbol{q}$ and $\boldsymbol{p}$ in this report, respectively.

1.3 Implementation details missing in paper

Counting of numerical gradient computations. The counting mechanism of "# gradients of target density" for evaluation reported in tables of the paper is unclear to me. Hence for evaluation stage, my attempt is to count the number of times that Lines 7-10 in Algorithm 5 of the paper is called during the execution.

Training and sampling details in Section 7.1. In Section 7.1, the following training details are missing: (i) number of hidden layers in HNN, the corresponding number of neurons, and choice of activation functions; (ii) the number of training steps and learning rate for HNN; (iii) number of training samples and the corresponding end time and step size. The following sampling details are missing: (iv) the values of $\Delta_{\max}^{hnn}$ and N_{lf} and (v) the number of burn-in samples. In my attempt, I adopt the settings of (i), (ii) and (iv) as those in Section 6. According to the information "required 10000 gradient computations of the target density", I set (iii) to be 10, 25, and 0.025, respectively.

Inference from elliptic PDE in Section 7.2. Section 7.2 is a Bayesian inverse problem (Stuart, 2010; Li et al., 2023; Aristoff and Bangerth, 2023), while many implementation details are missing in the paper and the description "samples from the 50-dimensional target joint density of $k(x, y)$" confuses me. Hence I consider a simplified scenario: the coefficient $k(x, y)$ has a parametric form, say associated with the parameter $\boldsymbol{\theta}$; and given the noisy observations $w = f(x, y) + \epsilon$ with $\epsilon \sim N(0, 1)$ specified by the paper, one wishes to conduct Bayesian inference for $\boldsymbol{\theta}$. Specifically, a prior distribution $p(\boldsymbol{\theta})$ is assumed for $\boldsymbol{\theta}$. The distribution of ϵ gives the likelihood

$$L(\boldsymbol{\theta}|w) \propto \prod_{i=1}^{N} \exp\left(-\frac{(w_i - f_{\boldsymbol{\theta}}(x_i, y_i))^2}{2}\right),$$

where N is the number of random "sensor" locations defined in the paper and the subscript in $f_{\boldsymbol{\theta}}(x_i, y_i)$ is to emphasize the value of f is a function of $\boldsymbol{\theta}$. Combined with the prior

distribution $p(\boldsymbol{\theta})$, the posterior distribution of $\boldsymbol{\theta}$ follows $\pi(\boldsymbol{\theta}|w) \propto L(\boldsymbol{\theta}|w)p(\boldsymbol{\theta})$. Thus, to use Hamiltonian dynamics, we may define the potential energy $U(\boldsymbol{\theta})$ in Equation (1) of the paper as $U(\boldsymbol{\theta}) = -\log \pi(\boldsymbol{\theta}|w)$.

If the problem formulation above is adopted, the following information is missing in the paper: (a) the parametric form of $k(x, y)$; (b) the prior distribution $p(\boldsymbol{\theta})$; (c) how the "sensor" locations (x_i, y_i) are sampled. Therefore, in my attempt, I assume $\boldsymbol{\theta} = (\theta_1, \theta_2) \in \mathbb{R}^2$, $k(x, y) = \theta_1 x + \theta_2 y$, and θ_1, θ_2 are independent and identically distributed normal random variables with mean one and variance one. Consequently,

$$U(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{N} \left(w_i - f_{\boldsymbol{\theta}}(x_i, y_i)\right)^2 + \frac{1}{2}\left((\theta_1 - 1)^2 + (\theta_2 - 1)^2\right).$$

In addition, I assume x and y are uniformly distributed in $[0, 3]$, respectively.

Apart from the settings of the Bayesian inference problem, the training and sampling details (i) - (iv) listed for Section 7.1 are also missing for Section 7.2. In my attempt, I adopt the same settings of (i), (ii), and (iv) as for Section 7.1. Then set (iii) to be 40, 40, and 0.025, respectively, using the information "requiring a total of 64,000 gradient estimations of the elliptic PDE".

2 Replication results (answers to Question (c))

This section reports the replication results of experiments in Sections 5, 6, and 7 of the paper with TensorFlow and TensorFlow Probability. Note that due to the limitations in computational resources, the results reported in paper are directly copied here instead of being reproduced with the PyTorch code provided by paper authors. For the implementations whose details are missing in the paper and the PyTorch code, the approaches I take have been listed in Section 1.3.

Table 1: Performance comparison between results of the paper and the replication with TensorFlow in sampling from 3D Rosenbrock density. The label "PyTorch" are the results reported in Table 2 of the paper, whereas "TensorFlow" corresponds to the replicated results. The notations "T" and "E" stand for the training and evaluation, respectively.

	ESS	**# of gradients of target density**	**Avg. ESS per gradient of target density**
(a) $M_t = 40$, $T = 100$			
PyTorch	(1339.13, 1665.76, 1648.85)	2,711,240 (T: 160,000 + E: 2,551,240)	0.000572
TensorFlow	(1321.54, 1754.10, 1847.42)	331,344 (T: 160,000 + E: 171,344)	0.004953
TensorFlow (new)	(1231.71, 2041.85, 2089.13)	1,837,037 (T: 160,000 + E: 1,677,037)	0.000973
(b) $M_t = 40$, $T = 150$			
PyTorch	(1217.63, 1710.97, 1654.11)	607,298 (T: 240,000 + E: 367,298)	0.00251
TensorFlow	(1260.05, 2235.17, 2163.68)	991,782 (T: 240,000 + E: 751,782)	0.00190
TensorFlow (new)	(1166.04, 1942.98, 1932.36)	2,403,488 (T: 160,000 + E: 2,243,488)	0.00069
(c) $M_t = 40$, $T = 250$			
PyTorch	(1445.03, 1729.98, 1387.91)	642,414 (T: 400,000 + E: 242,414)	0.00236
TensorFlow	(1118.22, 1502.56, 1478.52)	4,771,917 (T: 400,000 + E: 4,371,917)	0.000286
TensorFlow (new)	(908.31, 1477.30, 1564.15)	5,442,709 (T: 160,000 + E: 5,282,709)	0.000242

2.1 Sampling from a 3D Rosenbrock density

This subsection considers the sampling from a 3D Rosenbrock density with different end times $T = 100, 150, 250$ units. The experiment settings are the same as described in Section 5 of the paper. Table 1 lists the the performance of TensorFlow LHNN-NUTS with online error monitoring in terms of ESS, number of gradients of the target density, and average ESS across all the dimensions per gradient of the target density.

2.2 Analytical case studies

This subsection follows Section 6 in the paper to consider three analytical studies, i.e., sampling from 2D Neal's funnel density, 5D ill-conditioned Gaussian, 10D degenerate Rosenbrock density. The experiment settings are the same as described in the paper. Table 2

Table 2: Performance comparison between results of the paper and the replication with TensorFlow across different analytical studies. The label "PyTorch" are the results reported in Table 2 of the paper, whereas "TensorFlow" corresponds to the replicated results. The notations "T" and "E" stand for the training and evaluation, respectively.

	ESS	**# of gradients of target density**	**Avg. ESS per gradient of target density**
(a) 2D Neal's funnel			
PyTorch	(1019.87, 666.73)	3,596,688 (T: 400,000 + E: 3,196,688)	0.000234
TensorFlow	(894.10, 441.27)	1,044,343 (T: 400,000 + E: 644,343)	0.000639
(b) 5D ill-conditioned Gaussian			
PyTorch	(20000, 17741.21, 13906.42, 8382.5, 2763.38)	408, 800 (T: 400,000+ E: 8,800)	0.0307
TensorFlow	(20000, 17813.52, 14366.35, 7339.89, 2867.55)	400,000 (T: 400,000 + E: 0)	0.0312
TensorFlow (new)	(20000, 17552.68, 14573.15, 7636.58, 2775.28)	400,000 (T: 400,000 + E: 0)	0.0313
(c) 10D degenerate Rosenbrock			
PyTorch	(15535.69, 28217.40, 28965.00, 27280.67, 21268.00, 13532.28, 8418.80, 5731.09, 3299.92, 2045.02)	418,936 (T: 400,000 + E: 18,936)	0.0368
TensorFlow	(14773.49, 24428.78, 23739.14, 21146.39, 18646.44, 12394.88, 8126.24, 5634.24, 2789.04, 1322.50)	445,351 (T: 400,000 + E: 45,351)	0.0299
TensorFlow (new)	(13552.64, 27242.34, 26276.79, 23249.33, 19674.39, 15874.33, 11244.67, 7184.20, 4212.61, 2286.77)	436,167 (T: 400,000 + E: 36,167)	0.0346

records the results. Following Figures 8-10 of the paper, the scatter plots and empirical CDF plots of samples generated from the three distributions by TensorFlow version of L-HNN in NUTS with online error monitoring are also displayed in Figures 3, 4, and 5.

2.3 Computational case studies

This subsection considers two computational case studies: sampling from the stationary distribution of Allen-Cahn stochastic PDE with 25 inference parameters, and elliptic Bayesian inverse problem with 50 inference parameters, following Section 7 of the paper. Since there are many implementation details missing in these studies, I adopt the strategies mentioned in Section 1.3. The performance comparison results are listed in Table 3. Figure 6 shows the scatter plots of four dimensions of samples from the Allen-Cahn PDE.

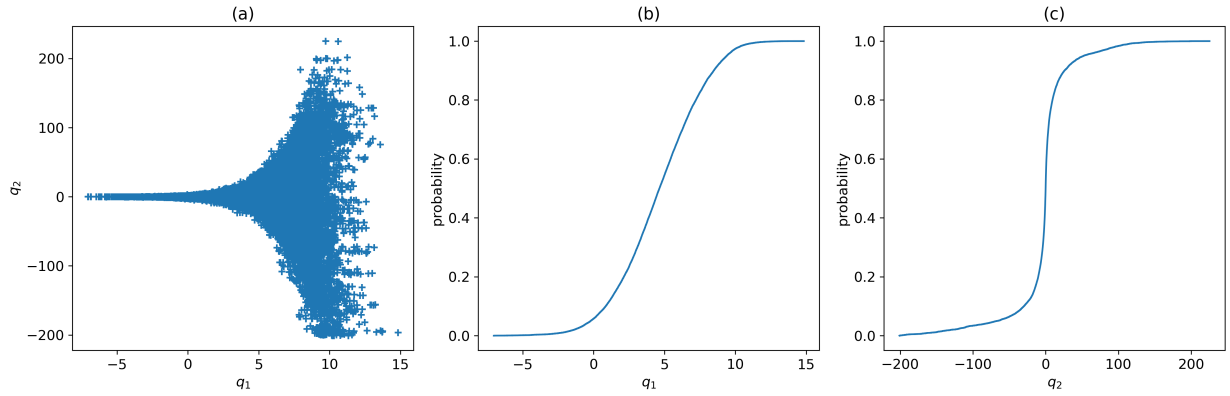

Figure 3: Samples generated from 2D Neal's funnel density by TensorFlow version of L-HNN in NUTS with online error monitoring: (a) scatter plot; (b) the eCDF plot of q_1; (c) the eCDF plot of q_2.

3 Testing plan and results

To ensure my implementation in TensorFlow is correct, I have conducted unit tests, regression tests and integration tests. The following subsections describe how the unit tests and integration tests are designed. In many cases, the function correctness is checked by comparing their outputs with those obtained from the PyTorch codes provided by authors. Note that all the tests are passed, which can be verified by running `pytest` for the submitted code.

3.1 Unit test

This subsection briefly the design of unit tests for functions, where the dependencies from other functions or data are removed.

1. `nn_models_tf.py` and `hnn_tf.py` contain the definition of MLP and HNN classes, respectively. The outputs of their methods are checked by comparing them with those from PyTorch version. Specifically,

Table 3: Performance comparison between results of the paper and the replication with TensorFlow across different computational studies. The label "PyTorch" are the results reported in Table 2 of the paper, whereas "TensorFlow" corresponds to the replicated results. The notations "T" and "E" stand for the training and evaluation, respectively.

	Avg. ESS	**# of gradients of target density**	**Avg. ESS per gradient of target density**
(a) Allen-Cahn stochastic PDE (25 inference parameters)			
PyTorch	414.5	97,904 (T: 10,000 + E: 87,904)	0.00423
TensorFlow	376.6	409,920 (T: 10,000 + E: 399,920)	0.000919
(b) Elliptic PDE (50 inference parameters)			
PyTorch	3229.9	217,824 (T: 64,000+ E: 153,824)	0.015
TensorFlow	1377.24	72,219 (T: 64,000+ E: 8,219)	0.019

- For `MLP`, the TensorFlow version of the model is first initialized and its weights are copied to the PyTorch version. Given the same inputs which are randomly drawn from normal distribution, it has been shown that the two models give the same outputs.
- For `HNN`, a simple `differentiable_model` that gives its input as the output is passed to both the TensorFlow and PyTorch versions of the model. Then we compare whether the outputs of methods `call`, `time_derivative`, `permutation_tensor` are the same.

2. `hnn_nuts_online_tf.py` contains `integrate_model_tf` that performs leapfrog integrator with HNN gradients, `stop_criterion_tf` that decides whether the stopping criterion at Equation (19) in paper is met, and `build_tree_tf` that implements Algorithm 5 in paper.

   - With `leapfrog_tf` mocked, the function output of `integrate_model_tf` is checked against a known deterministic result.

- The output `stop_criterion_tf` is compared with its PyTorch counterpart using same inputs that are randomly drawn from normal distribution.
- To check `build_tree_tf`, the two function above and `functions_tf` are mocked. Furthermore, uniform random number generators in `build_tree_tf` and its PyTorch version are replaced with deterministic numbers. The outputs from the two are compared.

3. `data_tf.py` contains functions `dynamics_fn_tf` that calculates numerical gradients, `get_trajectory_tf` that returns the trajectory from leapfrog steps, and `get_dataset_tf` that returns datasets.
   - To test `dynamics_fn_tf`, gradient calculation is mocked. The function output is compared to a known deterministic result.
   - With `dynamics_fn_tf` and `leapfrog_tf` mocked, we check whether the function returns the desired rearrangement of data.
   - For `get_dataset_tf`, we check whether the returned dataset has correct keys and shapes.

4. `utils_tf.py` contains `leapfrog_tf` that implements leapfrog integrator, `L2_loss` that defines the L_2 loss function, `to_pickle` and `from_pickle` that saves to and loads from pickle files, `choose_nonlinearity` that returns nonlinearity functions, `log_start` and `log_stop` that control writing the print outputs to both a file and stdout.
   - With gradient calculations mocked, we check `leapfrog_tf` and its PyTorch counterpart give the same output.
   - `L2_loss` is checked through comparing with the numpy implementation.
   - `to_pickle` and `from_pickle` are checked by creating a file, saving the file, and loading the file to see whether they have same contents.
   - In `to_pickle` and `from_pickle`, we check whether `sys.stdout` has been changed.

5. `functions_tf.py` contains the function `functions_tf` that computes the Hamiltonian function values. Its outputs are checked with those from PyTorch version with different arguments.

6. The newly added files `nuts_hnn.py` and `nuts_hnn_sample.py` adapt from the implementation of `tfp.mcmc.NoUTurnSampler`. Their correctness is checked through comparing each component when the differences are mocked.

3.2 Integration test

Integration tests are conducted for `train_hnn_tf.py`, `hnn_nuts_online_tf.py`, and `nuts_hnn.py`, which are the key to check the validness of the TensorFlow implementation.

Firstly, `train_hnn_tf.py` includes the training script of HNN. The design of test is to check given the same training data, and initialization of model weights, whether the model weights for TensorFlow and PyTorch versions are the same after one step of gradient update. On the other hand, `hnn_nuts_online_tf.py` includes the sampling script of HMC. For TensorFlow version and PyTorch one, HNN models with the same weights are passed to them. Further, we replace both the uniform and normal generators in both files with the deterministic numbers to avoid the discrepancies from randomness. Three HMC samples are drawn and the test checks whether the samples, trajectory length, probability, integration error, and the number of times that the leapfrog with numerical gradients is called, are all the same for both implementations. `nuts_hnn.py` is tested by comparing with `tfp.mcmc.NoUTurnSampler`, where the modifications are mocked. The outputs that `tfp.mcmc.NoUTurnSampler` originally has are expected to remain the same, while the additional outputs are compared with our expectations.

References

Aristoff, D. and Bangerth, W. (2023). A benchmark for the bayesian inversion of coefficients in partial differential equations. *SIAM Review*, 65(4):1074–1105.

Goodman, J. and Weare, J. (2010). Ensemble samplers with affine invariance. *Communications in Applied Mathematics and Computational Science*, 5(1):65 – 80.

Hoffman, M. D. and Gelman, A. (2014). The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo. *Journal of Machine Learning Research*, 15(47):1593–1623.

Li, S., Zhang, C., Zhang, Z., and Zhao, H. (2023). A data-driven and model-based accelerated hamiltonian monte carlo method for bayesian elliptic inverse problems. *Statistics and computing*, 33(4).

Neal, R. M., Brooks, S., Meng, X.-L., Gelman, A., Jones, G., Gelman, A., Jones, G., Meng, X.-L., and Brooks, S. (2011). Mcmc using hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*, pages 113–162. Chapman and Hall/CRC, United Kingdom.

Stuart, A. M. (2010). Inverse problems: A bayesian perspective. *Acta Numerica*, 19:451–559.

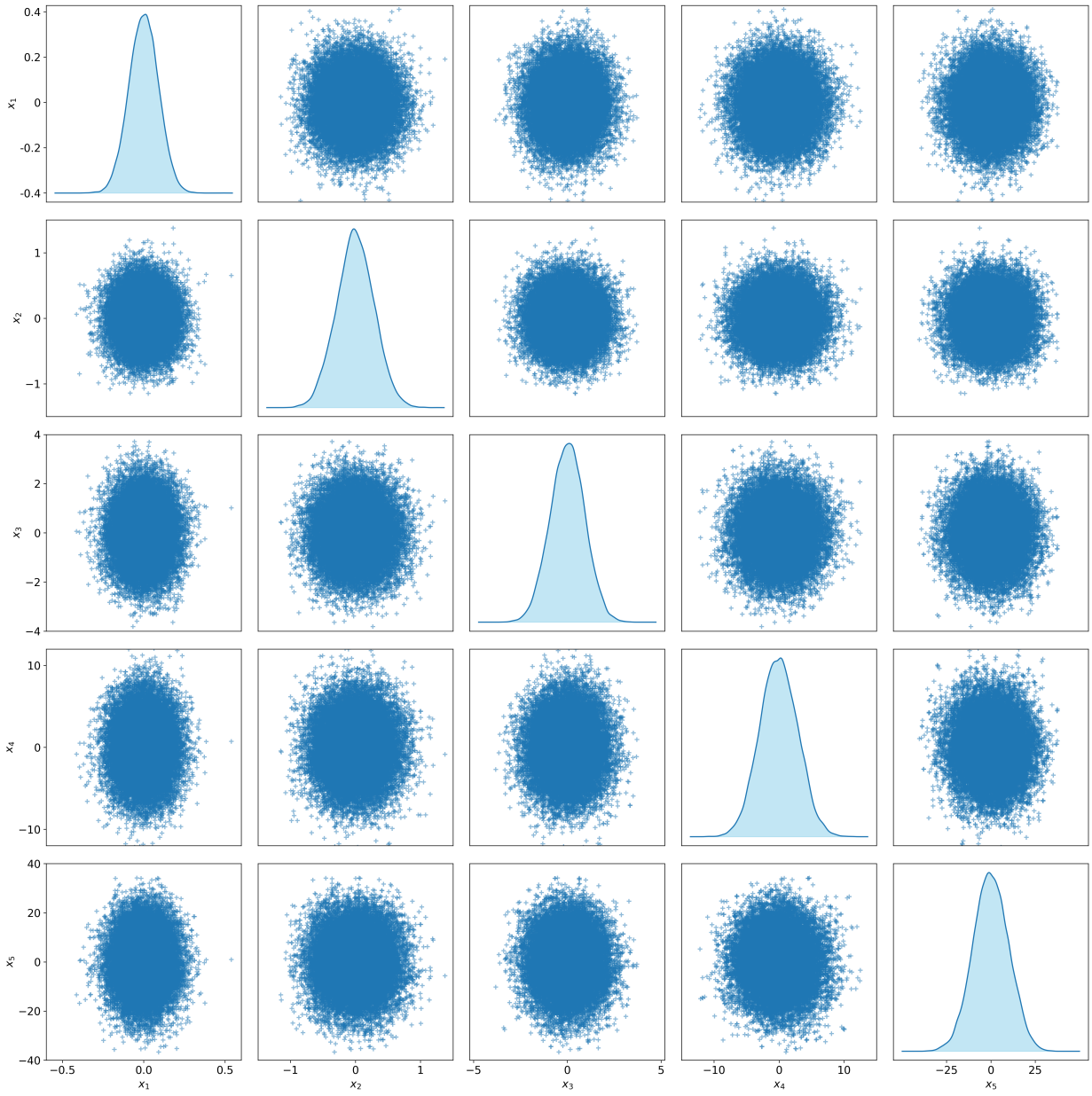

Figure 4: Scatter plots of samples generated from 5D ill-conditioned Gaussian distribution by TensorFlow version of L-HNN in NUTS with online error monitoring.

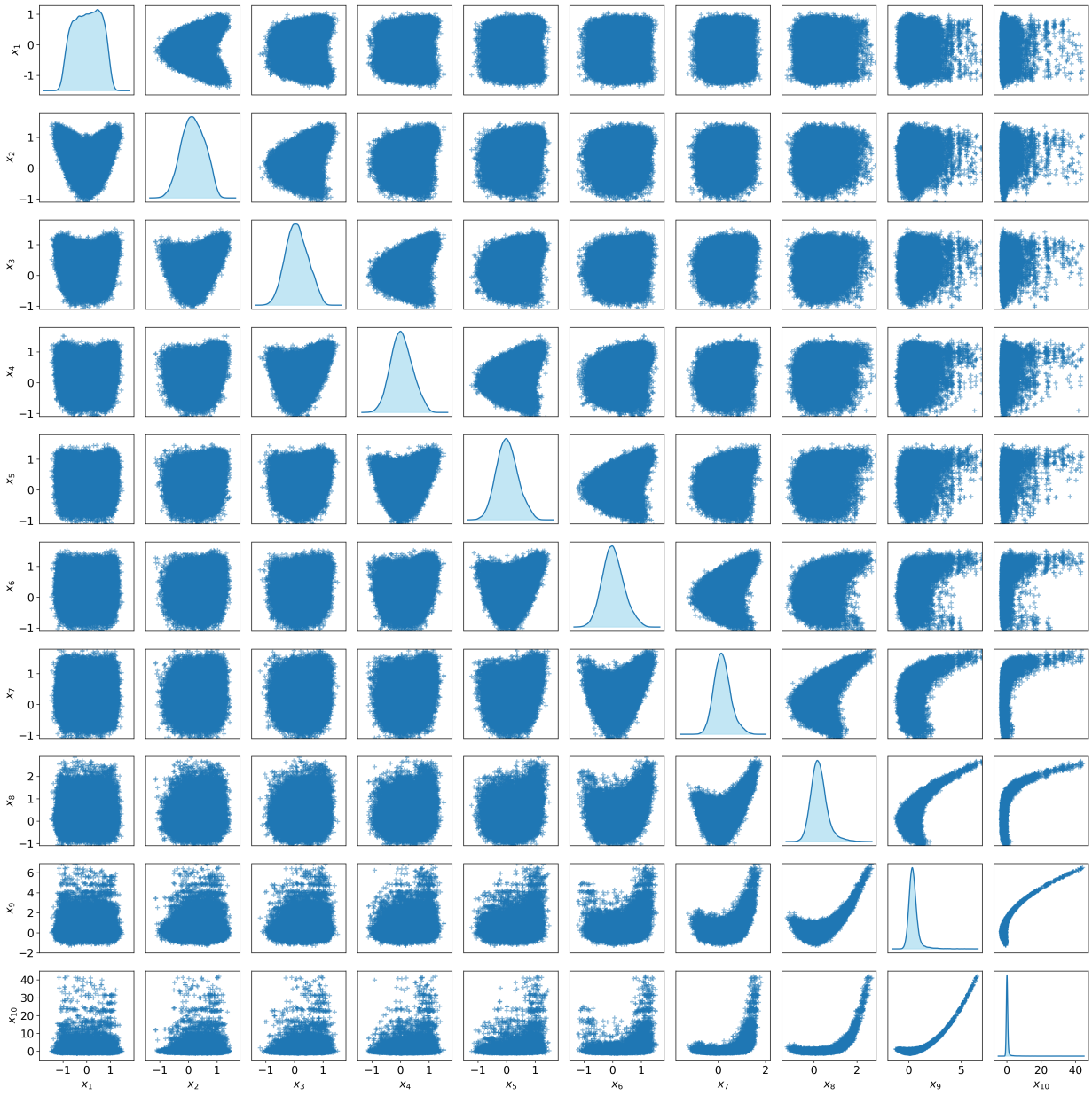

Figure 5: Scatter plots of samples generated from 10D degenerate Rosenbrock distribution by TensorFlow version of L-HNN in NUTS with online error monitoring.

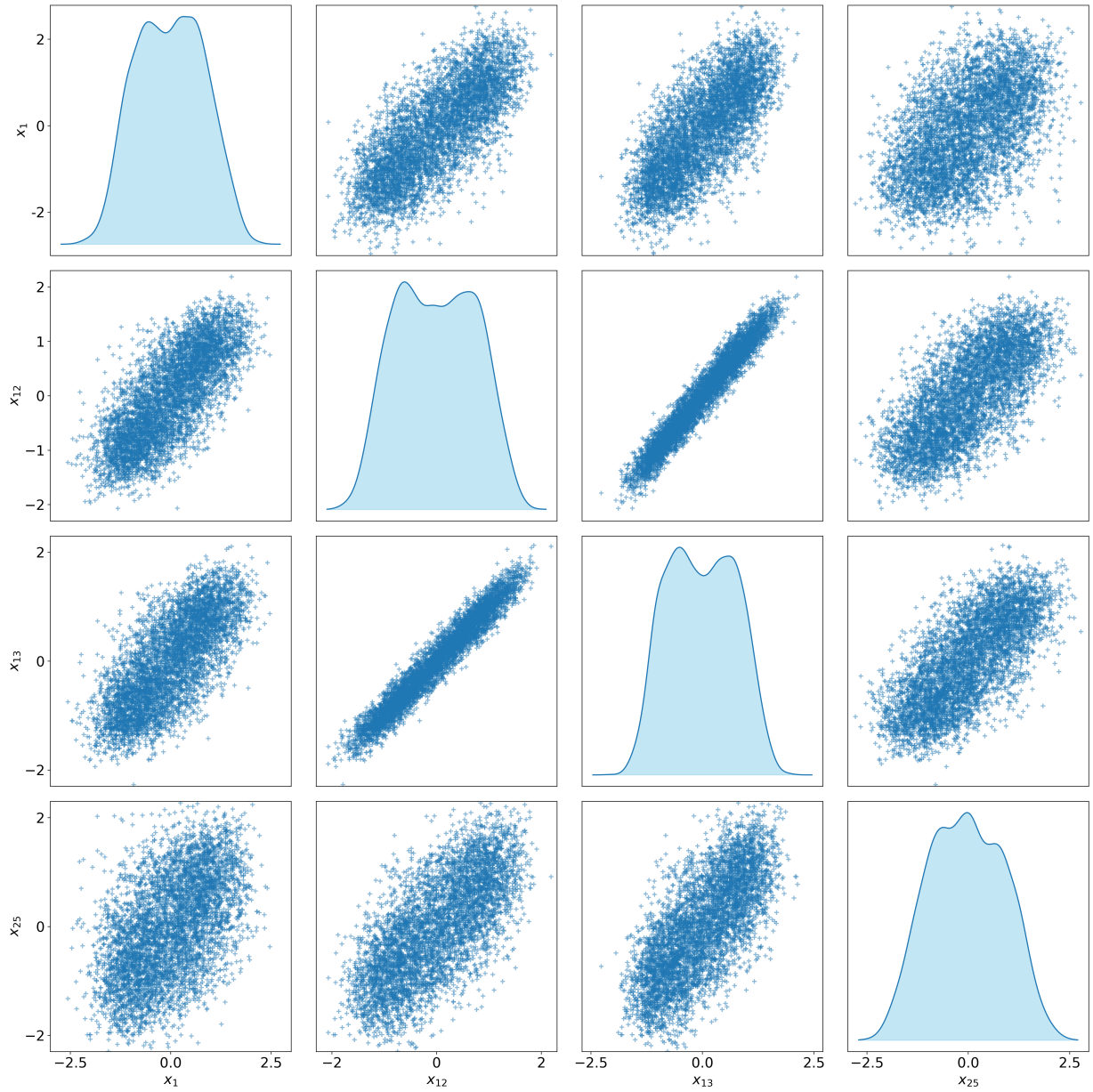

Figure 6: Scatter plots of four out of 25 dimensions, i.e., 1, 12, 13, and 25, of samples of Allen-Cahn equation by TensorFlow version of L-HNN in NUTS with online error monitoring.