Stochastic Gradient MCMC

Yichen Ji

Department of Statistics, University of Chicago

May 9, 2023

1 Large-Scale and High-Dimension: Computational Challenges

Markov Chain Monte Carlo (MCMC) methods have long been a cornerstone of statistical inference, providing a principled mechanism to draw samples from complex, high-dimensional distributions. These techniques are particularly essential in the context of Bayesian statistics, where the posterior distribution of model parameters is approximated by sampling. Despite their versatility and theoretical guarantees, traditional MCMC methods face significant computational challenges when applied to large-scale, high-dimensional problems, such as those encountered in Bayesian deep learning. These challenges primarily stem from the need to compute and manipulate full-gradient information, engendering a high computational cost that scales unfavorably with the size of the dataset and the dimensionality of the model.

In large-scale problems, the posterior distribution is typically proportional to the product of a likelihood term (which depends on the entire dataset) and a prior term. Consequently, computing the gradient requires summing over all data points, resulting in an operation that scales linearly with the size of the dataset. For modern applications involving millions or even billions of data points, this requirement can render traditional MCMC methods computationally infeasible.

Additionally, the efficiency of MCMC methods is often hampered by the curse of dimensionality. As the dimensionality of the parameter space increases, the volume of the space grows exponentially, leading to increasingly sparse and inefficient sampling. Moreover, traditional MCMC methods typically rely on global proposal distributions that do not adapt to the local geometry of the target distribution, which can result in slow mixing and poor exploration of the parameter space in high dimensions. This problem is exacerbated in the context of Bayesian deep learning, where models can have millions of parameters.

Stochastic Gradient Markov Chain Monte Carlo (SG-MCMC) methods have been developed to respond to these challenges. By leveraging stochastic optimization techniques, SG-MCMC methods replace the exact gradient computations of traditional MCMC with noisy estimates based on mini-batches of data. This modification significantly reduces the computational cost per iteration and makes it possible to apply MCMC methods to large-scale problems. Furthermore, specific SG-MCMC methods incorporate adaptive proposal mechanisms that account for the local geometry of the target distribution, thereby mitigating the issues associated with high-dimensional sampling. Therefore, SG-MCMC methods, by tackling the computational challenges of traditional MCMC, have emerged as a promising avenue for scalable Bayesian inference in the era of big data and high-dimensional modeling.

2 Preliminaries: Two Components

In the context of Bayesian statistics, let θ denote a parameter vector with prior $p(\theta)$ and data likelihood $p(x|\theta)$. The posterior distribution of a set of N observed data points $X = \{x_i\}_{i=1}^N$ is

$$p(\theta|X) \propto p(\theta) \prod_{i=1}^{N} p(x_i|\theta)$$

2.1 Stochastic Gradient Descent(SGD)

Stochastic Gradient Descent (SGD) is a popular optimization algorithm used in machine learning and deep learning for training models. Unlike standard Gradient Descent which computes the gradient using the entire dataset, SGD estimates the gradient using a single or a few randomly selected samples called mini-batch as a stochastic approximation, thereby making it more computationally efficient for large datasets.

Suppose our task is to find the maximum a posteriori (MAP) estimator θ^* of the true parameter θ. In SGD, given a subset of n data points $X_t = \{x_{t1}, ..., x_{tn}\}$, the iterative process of updating parameters at each iteration t is performed as follows:

$$\Delta\theta_t = \frac{\epsilon_t}{2}\left(\nabla \log p(\theta_t) + \frac{N}{n}\sum_{i=1}^{n} \nabla \log p(x_{ti} \mid \theta_t)\right)$$

The process is repeated for each minibatch until a stopping criterion is met. The key idea here is that: over multiple iterations the whole dataset is used and the noise in the gradient caused by using subsets rather than the whole dataset averages out.

2.2 Continuous Dynamics

Continuous dynamics leverage physical system simulations to propose future states in the Markov chain. Suppose we want to sample from the posterior distribution of θ in the form of

$$p(\theta|X) \propto \exp(-U(\theta))$$

where

$$U(\theta) = -\sum_{x} \log p(x|\theta) - \log p(\theta)$$

is called the potential energy function. We think of the posterior distribution as an energy landscape, and we would like to use continuous dynamics to explore such landscape as much as possible.

We will introduce two common dynamics, Hamiltonian and Langevin dynamics, as well as the corresponding sampling algorithms.

2.2.1 Hamiltonian Dynamics and Hamiltonian Monte Carlo (HMC)

Hamiltonian dynamics model the parameter space as a physical system, where the probability distribution corresponds to the potential energy and parameters move like particles with kinetic energy. The dynamics simulate the physical motion of an object with position θ, momentum r, and mass M on a frictionless surface, which can be mathematically formulated as a system of ordinary differential equations (ODEs):

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

Hamiltonian Monte Carlo (HMC) adds an auxiliary momentum variable r to augment the target posterior distribution of θ to be

$$\pi(\theta, r) \propto \exp(-U(\theta) - \frac{1}{2} r^T M^{-1} r = \exp(-H(\theta, r))$$

where $H(\theta, r) = U(\theta) + K(r)$ is called Hamiltonian or total energy, which is ideally conserved i.e. walking around a given level set of the energy landscape.

To achieve ergodicity, we resample the momentum variable to explore across different level sets for representative sampling. In practice, however, we cannot simulate exactly from the continuous system, so we need to do discretization of Hamilton equations by leveraging numerical integrators such as the Leapfrog method, which inevitably introduces some discretization errors. Nevertheless, an accept/reject mechanism like Metropolis-Hasting will correct such errors to leave the target distribution invariant.

Algorithm 2.1 Hamiltonian Monte Carlo (HMC)

Input: Starting position $\theta^{(1)}$ and step size ϵ
for $t \leftarrow 1, \ldots, N$ **do**
  # Momentum Refreshment
  $r^{(t)} \sim N(0, M)$
  # Leapfrog Discretization
  $r_0 \leftarrow r_0 - \frac{\epsilon}{2} \nabla U(\theta_0)$
  for $i \leftarrow 1, ..., m$ **do**
    $\theta_i \leftarrow \theta_{i-1)} + \epsilon M^{-1} r_{i-1}$
    $r_i \leftarrow r_{i-1} - \epsilon \nabla U(\theta_i)$
  end for
  $r_m \leftarrow r_m - \frac{\epsilon}{2} \nabla U(\theta_m)$
  $(\theta^*, r^*) \leftarrow (\theta_m, r_m)$
  # Metropolis-Hastings correction
  Set $\left(\theta^{(t+1)}, r^{(t+1)}\right) = \begin{cases} (\theta^*, r^*) \text{ w.p. } \min\left\{1, e^{H\left(\theta^{(t)}, r^{(t)}\right) - H(\theta^*, r^*)}\right\} \\ \left(\theta^{(t)}, -r^{(t)}\right) \text{ w.p. } 1 - \min\left\{1, e^{H\left(\theta^{(t)}, r^{(t)}\right) - H(\theta^*, r^*)}\right\} \end{cases}$
end for
Output: Samples $\theta^{(1)}, \ldots, \theta^{(N)}$

2.2.2 Langevin Dynamics and Langevin Monte Carlo (MALA)

Langevin dynamics introduces friction and random noise into the system's dynamics, which dissipates energy and injects stochasticity, respectively. The Langevin diffusion is defined by the stochastic differential equation (SDE):

$$d\theta = -\frac{1}{2}\nabla U(\theta)dt + dB_t$$

where $\nabla U(\theta)$ is the drift term and B_t denotes d-dimensional Brownian motion. Over infinitesimally small time intervals $\epsilon > 0$, the Langevin diffusion has approximate dynamics given by

$$\theta_{t+h} \approx \theta_t - \frac{\epsilon}{2}\nabla U(\theta_t) + \sqrt{h}Z$$

where Z is a vector of d independent standard Gaussian random variables.

This zoom-in approximation perspective also motivates us how to discretize the continuous Langevin dynamics whose equilibrium distribution is the target posterior distribution $p(\theta|X) \propto \exp(-U(\theta)) \propto p(\theta)\prod_{i=1}^{N} p(x_i|\theta)$, by taking gradient steps and injecting Gaussian noise into the parameter updates. Here's the update procedure:

$$\Delta\theta_t = \frac{\epsilon}{2}\left(\nabla \log p(\theta_t) + \sum_{i=1}^{N} \nabla \log p(x_i \mid \theta_t)\right) + \eta_t$$

where $\eta_t \sim N(0, \epsilon)$

Algorithm 2.2 Metropolis-Adjusted Langevin Algorithm (MALA)

Input: Starting position $\theta^{(1)}$ and step size ϵ
for $t \leftarrow 1, \ldots, N$ **do**
  # Gaussian Noise Generation
  $r^{(t)} \sim N(0, 1)$
  # Proposal Calculation
  $\theta^* \leftarrow \theta^{(t)} + \frac{\epsilon^2}{2}\nabla U(\theta^{(t)}) + \epsilon r^{(t)}$
  # Metropolis-Hastings correction
  Accept $\theta^{(t+1)} = \theta^*$ with probability $\min\left\{1, \frac{p(\theta^*)q(\theta^{(t)}|\theta^*)}{p(\theta^{(t)})q(\theta^*|\theta^{(t)})}\right\}$
  where $q(\theta, \theta') = N\left(\theta'; \theta + \frac{\epsilon^2}{2}\nabla U(\theta), \epsilon^2 I\right)$
  Otherwise set $\theta^{(t+1)} = \theta^{(t)}$
end for
Output: Samples $\theta^{(1)}, \ldots, \theta^{(N)}$

Langevin Monte Carlo (LMC), also known as the Metropolis-adjusted Langevin algorithm (MALA), uses a combination of Langevin dynamics and accept/reject mechanism to generate the states of a random walk that makes target posterior distribution invariant. Here's an intuitive breakdown of each step:

- Langevin dynamics employ gradient flow of the target to propose new states, driving the random walk towards regions of high probability (first-order information).
- Metropolis-Hastings decides to accept/reject the proposals from the dynamics by evaluating the target probability density for better mixing and convergence (zero-order information).

3 Connecting Dynamics with SGD: SG-MCMC

3.1 Framework

The general idea is to transform the posterior sampling process into the simulation of a continuous dynamic system, which is utilized to establish a Markov transition kernel. Generically speaking, all continuous Markov processes we have known for sampling can be written as a stochastic differential equation (SDE) specified by two matrices, a positive semi-definite diffusion matrix $\mathbf{D}$ and a skew-symmetric curl matrix $\mathbf{Q}$. The SDE is of the form

$$d\mathbf{z} = f(\mathbf{z})dt + \sqrt{2D(\mathbf{z})}dW(t)$$

where $\mathbf{z}$ can represent either the target parameter θ itself, or the augmented state space with auxiliary variables (θ, r). $W(t)$ denotes a d-dimensional Wiener process, and $f(\mathbf{z})$ denotes the deterministic drift term written in terms of the target distribution

$$f(\mathbf{z}) = -[D(\mathbf{z}) + Q(\mathbf{z})]\nabla H(\mathbf{z}) + \Gamma(\mathbf{z}), \quad \Gamma_i(\mathbf{z}) = \sum_{j=1}^{d} \frac{\partial}{\partial \mathbf{z}_j}(D_{ij}(\mathbf{z}) + Q_{ij}(\mathbf{z}))$$

[10] proves this framework as complete in two folds:

- Given any positive semi-definite diffusion $D(\mathbf{z})$ and skew-symmetric curl $Q(\mathbf{z})$, define a stochastic dynamic system using these two matrices, then this system will have a **unique** stationary distribution.
- Given a divergence-free vector field characterizing a stationary distribution, there **exists** a skew-symmetric curl matrix $Q(\mathbf{z})$ and a positive semi-definite diffusion matrix $D(\mathbf{z})$ such that the resulting stochastic system has the given vector field as its stationary distribution.

Under this framework, like HMC, we consider an ϵ-discretization of the continuous dynamics expressed by the above SDE with **full-data update** first:

$$\Delta(\mathbf{z_t}) = -\epsilon_t[(D(\mathbf{z}_t) + Q(\mathbf{z}_t))\nabla H(\mathbf{z}_t) + \Gamma(\mathbf{z}_t)] + \mathcal{N}(0, 2\epsilon_t D(\mathbf{z}_t))$$

Then, we connect such discretized dynamics with the spirit of SGD, which is replacing the full gradient evaluation with stochastic gradient

$$\widetilde{U}(\theta) = -\frac{|\mathcal{S}|}{|\widetilde{\mathcal{S}}|}\sum_{\mathbf{x}\in\widetilde{\mathcal{S}}} \log p(\mathbf{x} \mid \theta) - \log p(\theta); \quad \widetilde{\mathcal{S}} \subset \mathcal{S}$$

where $\widetilde{\mathcal{S}}$ are subsets i.e. mini-batches of the full dataset $\mathcal{S}$.

This stochastic gradient estimator of the full data gradient is noisy but unbiased. However, a corresponding concern is: whether the corresponding stochastic noise using $\widetilde{U}$ will lose the invariance of target distribution in the new dynamics. If not, how to offset such a negative effect to keep the target invariant?

It turns out that if we further assume

$$\nabla\widetilde{U}(\theta) = \nabla U(\theta) + \mathcal{N}(0, \mathbf{V}(\theta))$$

and by employing CLT, we can get a noisy version of the Hamiltonian gradient

$$\nabla\widetilde{H}(\mathbf{z}) = \nabla H(\mathbf{z}) + [\mathcal{N}(0, \mathbf{V}(\theta)), \mathbf{0}]^T$$

Then, we achieve a general version of the discretized dynamics using the stochastic Hamiltonian gradient as our update rule:

$$\mathbf{z}_{t+1} = \mathbf{z}_t - \epsilon_t\Big[(D(\mathbf{z}_t) + Q(\mathbf{z}_t))\nabla\widetilde{H}(\mathbf{z}_t) + \Gamma(\mathbf{z}_t)\Big] + \mathcal{N}\Big(0, \epsilon_t\Big(2D(\mathbf{z}_t) - \epsilon_t\widehat{B}_t\Big)\Big)$$

where $\widehat{B}_t$ is an estimate of Gaussian noise variance $\mathbf{V}(\theta)$.

Therefore, as $\epsilon_t \to 0$, that is, in an infinitesimal step size, the **full-data update** yields the correct invariant distribution, and our **stochastic update** approaches to the **full-data update**. Thus, our **stochastic update** leaves the target distribution invariant as $\epsilon_t \to 0$ as well. Note that this argument is only for intuitive understanding. Due to the page limit, see [10] for formal mathematical proofs.

3.2 Algorithms/ Samplers

Here, we illustrate how to apply such a complete framework to the actual samplers by specifying diffusion $D(\mathbf{z})$, curl $Q(\mathbf{z})$, and $H(\mathbf{z})$.

3.2.1 Stochastic Gradient Langevin Dynamics(SGLD)

SGLD follows first-order Langevin dynamics to generate samples by

$$\theta_{t+1} \leftarrow \theta_t - \epsilon_t D\nabla\tilde{U}(\theta_t) + \mathcal{N}(0, 2\epsilon_t D)$$

Unlike HMC, we don't have the auxiliary momentum variable in SGLD. This update rule corresponds to $\mathbf{z} = \theta, H(\theta) = U(\theta), D(\theta) = D, Q(\theta) = 0, \widehat{B}_t = 0$.

3.2.2 Stochastic Gradient Riemann Langevin Dynamics(SGRLD)

In SGLD, the diffusion matrix $D(\theta) = D$ is deterministic i.e. independent of θ, which motivates the following generalization of SGLD by using an adaptive diffusion matrix $D(\theta) = G^{-1}(\theta)$ where $G(\theta)$ is the Fisher information matrix. The update rule is

$$\theta_{t+1} \leftarrow \theta_t - \epsilon_t\Big[G(\theta_t)^{-1}\nabla\widetilde{U}(\theta_t) + \Gamma(\theta_t)\Big] + \mathcal{N}\Big(0, 2\epsilon_t G(\theta_t)^{-1}\Big)$$

corresponding to $D(\theta) = G^{-1}(\theta), Q(\theta) = 0, \widehat{B}_t = 0, \Gamma_i(\theta) = \sum_j \frac{\partial D_{ij}(\theta)}{\partial\theta_j}$

3.2.3 Stochastic Gradient Hamiltonian Monte Carlo(SGHMC)

The naive update rule is by replacing the full gradient $\nabla U(\theta)$ with the stochastic gradient $\widetilde{\nabla(U(\theta))}$ as we previously showed

$$\begin{cases} \theta_{t+1} \leftarrow \theta_t + \epsilon_t M^{-1} r_t \\ r_{t+1} \leftarrow r_t - \epsilon_t \nabla \widetilde{U}(\theta_t) \approx r_t - \epsilon_t \nabla U(\theta_t) + \mathcal{N}(0, \epsilon_t^2 V(\theta_t)) \end{cases}$$

However, this update rule doesn't converge to the target stationary distribution. The correct version is by adding a friction term and employing second-order Langevin dynamics

$$\begin{cases} \theta_{t+1} \leftarrow \theta_t + \epsilon_t M^{-1} r_t \\ r_{t+1} \leftarrow r_t - \epsilon_t \nabla \widetilde{U}(\theta_t) - \epsilon_t C M^{-1} r_t + \mathcal{N}\Big(0, \epsilon_t \Big(2C - \epsilon_t \widehat{B}_t\Big)\Big) \end{cases}$$

where $z = (\theta, r), H(\theta, r) = U(\theta) + \frac{1}{2} r^T M^{-1} r, \mathbf{Q}(\theta, r) = \begin{pmatrix} 0 & -\mathbf{I} \\ \mathbf{I} & 0 \end{pmatrix}, \mathbf{D}(\theta, r) = \begin{pmatrix} 0 & 0 \\ 0 & \mathbf{C} \end{pmatrix}$

4 Experiments

We compare the sampling performance of SGLD, SGRLD, and SGHMC on the 2-D multimodal problem, with 1/2/3/4 modes under Gaussian settings. Each scenario simulates 10000 samples with the same learning rate $\eta = 0.1$ for all 3 methods, and different choices of correlation values are considered for comparison. Under the same multimodal problem (i.e. fixed number of modes), we choose the same initialization point for all 3 methods where the probability density is relatively low. We also compare the stochastic methods with HMC in the unimodal case. HMC is particularly slow (10 minutes for 10000 iterations) even in the unimodal case possibly due to the accept/reject scheme i.e. Metropolis-Hastings, so we then compare the stochastic gradient methods with the samples from the true density distribution for multimodal settings.

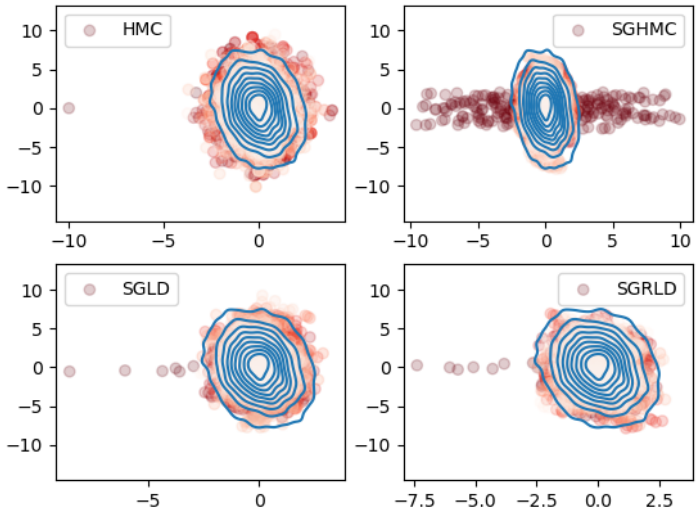

Figure 1 A distant initial point (-10,0) is set on purpose. HMC jumps to the right region in one step, but all stochastic gradient methods need to take multiple steps to reach the desired landscape. SGHMC is particularly worse than the other two, but all methods fully explore the distribution.

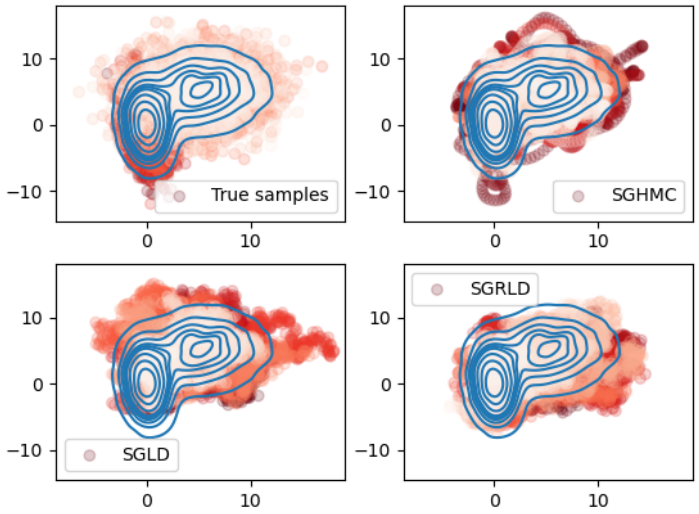

Figure 2 Initial point (10,0) is set at a low-density area on purpose. All SG-MCMC samples explore the level set corresponding to the initial point quite well but don't fully exploit the high-density area around (0,0).

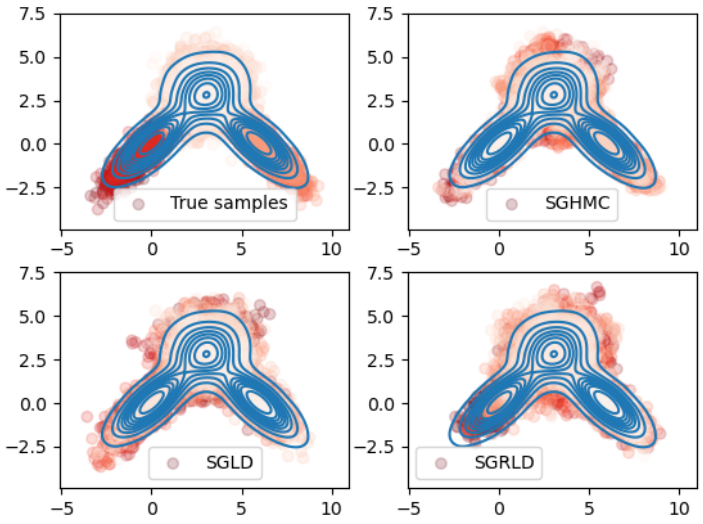

Figure 3 All SG-MCMC samples explore the triangle shape quite well, covering all parts of the density area.

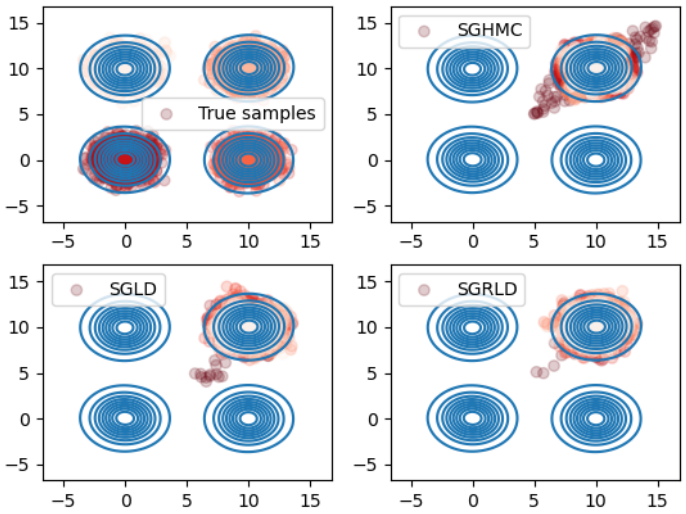

Figure 4 In the isolated scenario, a zero-density initial point (5,5) is set on purpose. Unsurprisingly, all SG-MCMC methods are trapped in a single Gaussian region and didn't explore the other three at all. Interestingly, they all choose the upper-right region, possibly due to similar dynamics behavior.

5 Discussion and Bibliography

Stochastic Gradient Langevin Dynamics (SGLD), Stochastic Gradient Riemann Langevin Dynamics (SGRLD), and Stochastic Gradient Hamiltonian Monte Carlo (SGHMC) are proposed in [14], [13], and [7], respectively. [10] provides a complete recipe for SG-MCMC based on continuous Markov processes specified via a curl matrix and a diffusion matrix, as we have shown in Section 3.

Here are some applications and recent advances of SG-MCMC:

5.1 Scalable Bayesian Inference

More than under the i.i.d assumption, SG-MCMC algorithms have proven useful for time-dependent data as well.

- [1] applies SG-MCMC on scalable distributed Bayesian matrix factorization and proposes an algorithm based on distributed stochastic gradient Langevin dynamics.
- [11] demonstrates outperforming results in learning the parameters of hidden Markov models(HMMs).
- [2] develops novel buffered SG-MCMC samplers for discrete, continuous, and mixed-type state space models(SSMs) with analytic message passing.
- [3] further proposes particle buffered SG-MCMC samplers to generalize the buffered gradient estimators to nonlinear SSMs.

5.2 Variational Inference and Optimization

A recent development combines ideas from SG-MCMC and variational inference (VI), which traditionally makes strong assumptions about the posterior's functional form. The new method, described as a non-parametric VI scheme, relaxes these assumptions and improves the mixing rate of the Markov chain. This is achieved by operating on a "self-averaged" posterior energy function, where parts of the latent variables are averaged over samples from earlier iterations.

The method can be modified further in a "dropout" manner for better scalability. The new MCMC/VI hybrid allows sampling from a fully joint posterior, a completely factorized posterior, and any in-between, thereby balancing approximation quality with speed. See [4],[5],[16],[12],[8].

5.3 Deep Learning and Neural Networks

SG-MCMC is also being used for Bayesian inference in deep learning models, especially Bayesian neural networks (BNNs). BNNs rely on ensemble averages over model parameters typically obtained from MCMC algorithms, contrasting with regular neural networks that depend on a single set of parameters. Due to their size, BNNs require scalable MCMC approaches like SGMCMC. The recently proposed structured SGMCMC scheme has shown improvements in both small and large-scale experiments on ResNet-20 architectures on CIFAR-10, Fashion MNIST, and SVHN in terms of runtime and final accuracy. See [15],[9],[6],[17].

References

[1] S. Ahn, A. Korattikara, N. Liu, S. Rajan, and M. Welling. Large-scale distributed bayesian matrix factorization using stochastic gradient mcmc. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 9–18, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336642. doi: 10.1145/2783258.2783373. URL https://doi.org/10.1145/2783258.2783373.

[2] C. Aicher, Y.-A. Ma, N. J. Foti, and E. B. Fox. Stochastic gradient mcmc for state space models, 2019.

[3] C. Aicher, S. Putcha, C. Nemeth, P. Fearnhead, and E. B. Fox. Stochastic gradient mcmc for nonlinear state space models, 2019.

[4] A. Alexos, A. Boyd, and S. Mandt. Structured stochastic gradient mcmc, 2022.

[5] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe. Variational inference: A review for statisticians. *Journal of the American statistical Association*, 112(518):859–877, 2017.

[6] L. Cardelli, A. Csikász-Nagy, N. Dalchau, M. Tribastone, and M. Tschaikowski. An activation function free neural network. *arXiv preprint arXiv:1912.03383*, 2019.

[7] T. Chen, E. Fox, and C. Guestrin. Stochastic gradient hamiltonian monte carlo. In *International conference on machine learning*, pages 1683–1691. PMLR, 2014.

[8] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley. Stochastic variational inference. *Journal of Machine Learning Research*, 2013.

[9] Y. Li, R. Turner, R. Chen, and J. Liu. Preconditioned stochastic gradient langevin dynamics for deep neural networks. *arXiv preprint arXiv:1512.07666*, 2016.

[10] Y.-A. Ma, T. Chen, and E. Fox. A complete recipe for stochastic gradient mcmc. *Advances in neural information processing systems*, 28, 2015.

[11] Y.-A. Ma, N. J. Foti, and E. B. Fox. Stochastic gradient MCMC methods for hidden Markov models. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2265–2274. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/ma17a.html.

[12] J. T. Ormerod and M. P. Wand. Explaining variational approximations. *The American Statistician*, 64(2):140–153, 2010.

[13] S. Patterson and Y. W. Teh. Stochastic gradient riemannian langevin dynamics on the probability simplex. *arXiv preprint arXiv:1301.3840*, 2013.

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

[15] A. Wilson and P. Izmailov. Bayesian deep learning and a probabilistic perspective of generalization. *arXiv preprint arXiv:2002.08791*, 2020.

[16] Q. Zhang, F. Nian, S. Patterson, and J. Liu. Advances in variational inference. *arXiv preprint arXiv:1711.05597*, 2018.

[17] R. Zhang, C. Li, J. Zhang, C. Chen, and A. G. Wilson. Cyclical stochastic gradient mcmc for bayesian deep learning, 2020.

article listings

```
# -*- coding: utf-8 -*-
"""### Unimodal Gaussian"""

import torch
import torch.optim as optim
import tqdm
import seaborn as sns
import numpy as np
from matplotlib import pyplot as plt
import scipy.linalg as la

class NomalDist:
    def __init__(self, mu, sigma):
        self.d = len(mu)
        self.mu = mu
        self.sigma = sigma
        self.sigma_inv = torch.inverse(self.sigma)
        self.sigma_cho = torch.cholesky(sigma)

    def loss(self, x):
        """Quantity to be optimized later (neg log density of distribution in
        return (x - self.mu).T @ (self.sigma_inv @ (x - self.mu))

    def sample(self, n_samples):
        """Helper to return samples of distribution"""
        z = torch.randn((n_samples, self.d))
        samples = (self.sigma_cho @ z.T).T
        samples += self.mu
        return samples

class BiModalNormal:
    def __init__(self, mu1, mu2, sigma1, sigma2):
        self.d = len(mu1)
        self.mu1 = mu1
        self.mu2 = mu2
        self.sigma1 = sigma1
        self.sigma2 = sigma2
        self.sigma1_inv = torch.inverse(self.sigma1)
        self.sigma2_inv = torch.inverse(self.sigma2)
        self.sigma1_cho = torch.cholesky(sigma1)
        self.sigma2_cho = torch.cholesky(sigma2)

    def loss(self, x):
```

```
        """
        Quantity to be optimized later (neg log density of distribution in thi
        This is sum of negative log densities for each mode.
        """
        loss1 = (x - self.mu1).T @ (self.sigma1_inv @ (x - self.mu1))
        loss2 = (x - self.mu2).T @ (self.sigma2_inv @ (x - self.mu2))
        return -torch.log(torch.exp(-0.5*loss1) + torch.exp(-0.5*loss2))

    def sample(self, n_samples):
        """
        Helper to return samples of distribution.
        Samples are generated by first sampling from each mode and then stacki
        """
        z1 = torch.randn((n_samples // 2, self.d))
        samples1 = (self.sigma1_cho @ z1.T).T
        samples1 += self.mu1

        z2 = torch.randn((n_samples - n_samples // 2, self.d))
        samples2 = (self.sigma2_cho @ z2.T).T
        samples2 += self.mu2

        return torch.cat((samples1, samples2), dim=0)

class MultiModalNormal:
    def __init__(self, mus, sigmas):
        """
        Initialize MultiModalNormal with a list of means and covariance matric

        mus : list of torch tensors
            List of means for each mode.
        sigmas : list of torch tensors
            List of covariance matrices for each mode.
        """
        assert len(mus) == len(sigmas), "Number of means and covariances must

        self.d = len(mus[0])
        self.mus = mus
        self.sigmas = sigmas
        self.sigmas_inv = [torch.inverse(sigma) for sigma in sigmas]
        self.sigmas_cho = [torch.cholesky(sigma) for sigma in sigmas]

    def loss(self, x):
        """
        Compute the negative log density.
```

```
        x : torch tensor
            Input sample.
        """
        losses = []
        for mu, sigma_inv in zip(self.mus, self.sigmas_inv):
            loss = (x - mu).T @ (sigma_inv @ (x - mu))
            losses.append(loss)

        return -torch.logsumexp(-0.5 * torch.stack(losses), dim=0)

    def sample(self, n_samples):
        """
        Sample from the multimodal distribution.

        n_samples : int
            Number of samples to generate.
        """
        num_modes = len(self.mus)
        samples_per_mode = n_samples // num_modes
        leftover_samples = n_samples % num_modes
        samples = []

        for mu, sigma_cho in zip(self.mus, self.sigmas_cho):
            z = torch.randn((samples_per_mode + (leftover_samples > 0), self.d
            samples.append((sigma_cho @ z.T).T + mu)
            leftover_samples -= 1

        return torch.cat(samples, dim=0)

class SGLD:
    def __init__(self, params, eta, log_density):
        """
        Stochastic gradilog_densityent monte carlo sampler via Langevin Dynam
        Parameters
        ----------
        eta: float
            learning rate param
        log_density: function computing log_density (loss) for given sample an
        """
        self.eta = eta
        self.log_density = log_density
        self.optimizer = torch.optim.SGD(params, lr=1, momentum=0.)
# momentum is set to zero
```

```
    def _noise(self, params):
        """We are adding param+noise to each param."""
        std = np.sqrt(2 * self.eta)
        loss = 0.
        for param in params:
            noise = torch.randn_like(param) * std
            loss += (noise * param).sum()
        return loss

    def sample(self, params):
        self.optimizer.zero_grad()
        loss = self.log_density(params) * self.eta
        loss += self._noise(params)  # add noise*param before calling backward
        loss.backward()  # let autograd do its thing
        self.optimizer.step()
        return params

class SGHMC:

    def __init__(self, params, alpha, eta, log_density):
        """
        Stochastic Gradient Monte Carlo sampler WITH momentum
        This is Hamiltonian Monte Carlo.

        Paramters
        ---------
        alpha: momentum param
        eta: learning rate
        log_density: loss function for given sample/batch of data
        """
        self.alpha = alpha
        self.eta = eta
        self.log_density = log_density
        self.optimizer = torch.optim.SGD(params, lr=1, momentum=(1 - self.alph

    def _noise(self, params):
        std = np.sqrt(2 * self.alpha * self.eta)
        loss = 0.
        for param in params:
            noise = torch.randn_like(param) * std
            loss += (noise * param).sum()
        return loss
```

```
    def sample(self, params):
        self.optimizer.zero_grad()
        loss = self.log_density(params) * self.eta
        loss += self._noise(params)
        loss.backward()
        self.optimizer.step()
        return params

class SGRLD:

    def __init__(self, params, eta, log_density, preconditioner=None):
        """
        Stochastic Gradient Riemannian Langevin Dynamics sampler

        Parameters
        ----------
        params : list of torch tensors
            Model parameters
        eta : float
            Learning rate
        log_density : function
            Log probability density function
        preconditioner : function (optional)
            Preconditioning matrix function
        """
        self.eta = eta
        self.log_density = log_density
        self.optimizer = torch.optim.SGD(params, lr=1, momentum=0.)
# momentum is set to zero
        self.preconditioner = preconditioner if preconditioner is not None els

    def _noise(self, params):
        std = np.sqrt(2 * self.eta)
        loss = 0.
        for param in params:
            noise = torch.randn_like(param) * std
            loss += (noise * param * self.preconditioner(param)).sum()
        return loss

    def sample(self, params):
        self.optimizer.zero_grad()
        loss = self.log_density(params) * self.eta
        loss += self._noise(params)  # add noise*param before calling backward
        loss.backward()  # let autograd do its thing
```

```
        self.optimizer.step()
        return params

# true data generation process (UniModal)
mu = torch.zeros(2)
sigma = torch.tensor([[1, .8],
                      [-.8, 10]])
dist = NomalDist(mu, sigma)
original_samples = dist.sample(10000)

# true data generation process (BiModal)
mu1 = torch.zeros(2)
mu2 = torch.tensor([5,5])
sigma1 = torch.tensor([[1, .9],
                       [-.9, 10]])
sigma2 = torch.tensor([[10, -.9],
                       [.9, 10]])
dist2 = BiModalNormal(mu1,mu2,sigma1,sigma2)
original_samples2 = dist.sample(10000)
# true density plot
sns.kdeplot(original_samples2[:,0], original_samples2[:,1])

# HMC
def E(A, u0, v0, u, v):
    """Total energy."""
    A = np.array([[0,1],[-1,0]])
    return (u0 @ tau @ u0 + v0 @ v0) - (u @ tau@u + v @ v)

def leapfrog(A, u, v, h, N):
    """Leapfrog finite difference scheme."""
    v = v - h/2 * A @ u
    for i in range(N-1):
        u = u + h * v
        v = v - h * A @ u

    u = u + h * v
    v = v - h/2 * A @ u

    return u, v

niter = 10000
h = 0.01
orbit = np.zeros((niter+1, 2))
u = np.array([-10.,0.])
```

```
orbit[0] = u
tau = la.inv(sigma)
# change tau for Gaussian mixture!!!!
A = np.array([[0,1],[-1,0]])
for k in tqdm.tqdm(range(niter)):
    v0 = np.random.normal(0,1,2)

    u, v = leapfrog(tau, u, v0, h, N)

    # accept-reject
    u0 = orbit[k]
    a = np.exp(E(A, u0, v0, u, v))
    r = np.random.rand()

    if r < a:
        orbit[k+1] = u
    else:
        orbit[k+1] = u0

# SGLD
x = torch.tensor([-10., 0.], requires_grad=True)
sgld = SGLD([x], eta=1e-1, log_density=dist.loss)
samples_SGLD = []

for epoch in tqdm.tqdm(range(niter)):
    x = sgld.sample(x)
    samples_SGLD.append(x.data.clone().detach().T)

samples_SGLD = np.array(np.vstack(samples_SGLD))

# SGRLD
x = torch.tensor([-10., 0.], requires_grad=True)
sgrld = SGRLD([x], eta=1e-1, log_density=dist.loss)
samples_SGRLD = []

for epoch in tqdm.tqdm(range(niter)):
    x = sgrld.sample(x)
    samples_SGRLD.append(x.data.clone().detach().T)

samples_SGRLD = np.array(np.vstack(samples_SGRLD))

# SGHMC
x = torch.tensor([-10., 0.], requires_grad=True)
sghmc = SGHMC([x], alpha=0.01, eta=1e-1, log_density=dist.loss)
```

```
samples_SGHMC = []

for epoch in tqdm.tqdm(range(niter)):
    x = sghmc.sample(x)
    samples_SGHMC.append(x.detach().clone().T)

samples_SGHMC = np.array(np.vstack(samples_SGHMC))

# HMC plot
#sns.kdeplot(orbit[:, 0], orbit[:, 1], alpha=.5)
#plt.plot(orbit[:,0], orbit[:,1], alpha=0.2, label='')
#plt.scatter(orbit[:1,0], orbit[:1,1], c='red', s=30)
plt.scatter(orbit[1:,0], orbit[1:,1],
            c=np.arange(niter)[::-1],
            cmap='Reds',alpha=.2,label='HMC')
sns.kdeplot(original_samples[:,0], original_samples[:,1])
plt.legend()
plt.axis([-5,5,-10,10])

# SGLD plot
sns.kdeplot(original_samples[:, 0], original_samples[:, 1])
plt.scatter(samples_SGLD[:, 0], samples_SGLD[:, 1],
            c=np.arange(10000)[::-1],
            cmap='Reds',alpha=.2, label='SGLD')
plt.legend()

# SGRLD plot
sns.kdeplot(original_samples[:, 0], original_samples[:, 1])
plt.scatter(samples_SGRLD[:, 0], samples_SGRLD[:, 1],
            c=np.arange(10000)[::-1], cmap='Reds',
            alpha=.2, label='SGRLD')
plt.legend()

# SGHMC plot
sns.kdeplot(original_samples[:, 0], original_samples[:, 1])
plt.scatter(samples_SGHMC[:, 0], samples_SGHMC[:, 1],
            c=np.arange(niter)[::-1], cmap='Reds',
            alpha=.2, label='SGHMC')
plt.legend()

fig = plt.figure()

# HMC
ax1 = fig.add_subplot(2,2,1)
```

```
#plt.subplot(221)
ax1.scatter(orbit[1:,0], orbit[1:,1],
            c=np.arange(niter)[::-1],
            cmap='Reds',alpha=.2,label='HMC')
sns.kdeplot(original_samples[:,0], original_samples[:,1])
ax1.legend()

#niter=1000
#plt.subplot(222)
# SGHMC plot
ax2 = fig.add_subplot(2,2,2)
sns.kdeplot(original_samples[:, 0], original_samples[:, 1])
ax2.scatter(samples_SGHMC[:, 0], samples_SGHMC[:, 1],
            c=np.arange(niter)[::-1], cmap='Reds',
            alpha=.2, label='SGHMC')
ax2.legend()

# SGLD
ax3 = fig.add_subplot(2,2,3)
#plt.subplot(223)
sns.kdeplot(original_samples[:, 0], original_samples[:, 1])
ax3.scatter(samples_SGLD[:, 0], samples_SGLD[:, 1],
             c=np.arange(niter)[::-1], cmap='Reds',
             alpha=.2, label='SGLD')
ax3.legend()

# SGRLD
ax4 = fig.add_subplot(2,2,4)
#plt.subplot(224)
sns.kdeplot(original_samples[:, 0], original_samples[:, 1])
ax4.scatter(samples_SGRLD[:, 0], samples_SGRLD[:, 1],
            c=np.arange(niter)[::-1], cmap='Reds',
            alpha=.2, label='SGRLD')
ax4.legend()


fig.savefig('unimodal Gaussian.png')

"""### Bimodal Gaussian"""

# true data generation process (BiModal)
N=10000
mu1 = torch.zeros(2)
mu2 = torch.tensor([5,5])
```

```
sigma1 = torch.tensor([[1, .9],
                       [-.9, 10]])
sigma2 = torch.tensor([[10, -.9],
                       [.9, 10]])
dist2 = BiModalNormal(mu1,mu2,sigma1,sigma2)
original_samples2 = dist2.sample(N)
# true density plot
sns.kdeplot(original_samples2[:,0], original_samples2[:,1])

# SGLD
x = torch.tensor([10., 0.], requires_grad=True)
sgld = SGLD([x], eta=1e-1, log_density=dist2.loss)
samples_SGLD = []

for epoch in tqdm.tqdm(range(N)):
    x = sgld.sample(x)
    samples_SGLD.append(x.data.clone().detach().T)

samples_SGLD = np.array(np.vstack(samples_SGLD))

# SGRLD
x = torch.tensor([10., 0.], requires_grad=True)
sgrld = SGRLD([x], eta=1e-1, log_density=dist2.loss)
samples_SGRLD = []

for epoch in tqdm.tqdm(range(N)):
    x = sgrld.sample(x)
    samples_SGRLD.append(x.data.clone().detach().T)

samples_SGRLD = np.array(np.vstack(samples_SGRLD))

# SGHMC
x = torch.tensor([10., 0.], requires_grad=True)
sghmc = SGHMC([x], alpha=0.01, eta=1e-1, log_density=dist2.loss)
samples_SGHMC = []

for epoch in tqdm.tqdm(range(N)):
    x = sghmc.sample(x)
    samples_SGHMC.append(x.detach().clone().T)

samples_SGHMC = np.array(np.vstack(samples_SGHMC))

fig = plt.figure()
```

```
# true density plot
ax1 = fig.add_subplot(2,2,1)
#plt.subplot(221)
ax1.scatter(original_samples2[:,0], original_samples2[:,1],
            c=np.arange(N)[::-1], cmap='Reds',
            alpha=.2,label='True samples')
sns.kdeplot(original_samples2[:,0], original_samples2[:,1])
ax1.legend()

#niter=1000
#plt.subplot(222)
# SGHMC plot
ax2 = fig.add_subplot(2,2,2)
sns.kdeplot(original_samples2[:, 0], original_samples2[:, 1])
ax2.scatter(samples_SGHMC[:, 0], samples_SGHMC[:, 1],
            c=np.arange(N)[::-1], cmap='Reds',
            alpha=.2, label='SGHMC')
ax2.legend()

# SGLD
ax3 = fig.add_subplot(2,2,3)
#plt.subplot(223)
sns.kdeplot(original_samples2[:, 0], original_samples2[:, 1])
ax3.scatter(samples_SGLD[:, 0], samples_SGLD[:, 1],
            c=np.arange(N)[::-1], cmap='Reds',
            alpha=.2, label='SGLD')
ax3.legend()

# SGRLD
ax4 = fig.add_subplot(2,2,4)
#plt.subplot(224)
sns.kdeplot(original_samples2[:, 0], original_samples2[:, 1])
ax4.scatter(samples_SGRLD[:, 0], samples_SGRLD[:, 1],
            c=np.arange(N)[::-1], cmap='Reds',
            alpha=.2, label='SGRLD')
ax4.legend()


fig.savefig('bimodal Gaussian.png')

"""### Trimodal Gaussian"""

# true data generation process (TriModal)
N=10000
```

```
mu1 = torch.zeros(2)
mu2 = torch.tensor([6,0])
mu3 = torch.tensor([3,3])
mus = [mu1,mu2,mu3]
sigma1 = torch.tensor([[1, .9],
                       [.9, 1]])
sigma2 = torch.tensor([[1,-.9],
                       [-.9,1]])
sigma3 = torch.tensor([[1, .1],
                       [.1, 1]])
sigmas = [sigma1, sigma2, sigma3]
dist3 = MultiModalNormal(mus,sigmas)
original_samples3 = dist3.sample(N)
# true density plot
sns.kdeplot(original_samples3[:,0], original_samples3[:,1])

# SGLD
x = torch.tensor([3., 0.], requires_grad=True)
sgld = SGLD([x], eta=1e-1, log_density=dist3.loss)
samples_SGLD = []

for epoch in tqdm.tqdm(range(N)):
    x = sgld.sample(x)
    samples_SGLD.append(x.data.clone().detach().T)

samples_SGLD = np.array(np.vstack(samples_SGLD))

# SGRLD
x = torch.tensor([3., 0.], requires_grad=True)
sgrld = SGRLD([x], eta=1e-1, log_density=dist3.loss)
samples_SGRLD = []

for epoch in tqdm.tqdm(range(N)):
    x = sgrld.sample(x)
    samples_SGRLD.append(x.data.clone().detach().T)

samples_SGRLD = np.array(np.vstack(samples_SGRLD))

# SGHMC
x = torch.tensor([3., 0.], requires_grad=True)
sghmc = SGHMC([x], alpha=0.01, eta=1e-1, log_density=dist3.loss)
samples_SGHMC = []

for epoch in tqdm.tqdm(range(N)):
```

```
    x = sghmc.sample(x)
    samples_SGHMC.append(x.detach().clone().T)

samples_SGHMC = np.array(np.vstack(samples_SGHMC))

fig = plt.figure()

# true density plot
ax1 = fig.add_subplot(2,2,1)
#plt.subplot(221)
ax1.scatter(original_samples3[:,0], original_samples3[:,1],
            c=np.arange(N)[::-1], cmap='Reds',
            alpha=.2,label='True samples')
sns.kdeplot(original_samples3[:,0], original_samples3[:,1])
ax1.legend()

#niter=1000
#plt.subplot(222)
# SGHMC plot
ax2 = fig.add_subplot(2,2,2)
sns.kdeplot(original_samples3[:, 0], original_samples3[:, 1])
ax2.scatter(samples_SGHMC[:, 0], samples_SGHMC[:, 1],
            c=np.arange(N)[::-1], cmap='Reds',
            alpha=.2, label='SGHMC')
ax2.legend()

# SGLD
ax3 = fig.add_subplot(2,2,3)
#plt.subplot(223)
sns.kdeplot(original_samples3[:, 0], original_samples3[:, 1])
ax3.scatter(samples_SGLD[:, 0], samples_SGLD[:, 1],
            c=np.arange(N)[::-1], cmap='Reds',
            alpha=.2, label='SGLD')
ax3.legend()

# SGRLD
ax4 = fig.add_subplot(2,2,4)
#plt.subplot(224)
sns.kdeplot(original_samples3[:, 0], original_samples3[:, 1])
ax4.scatter(samples_SGRLD[:, 0], samples_SGRLD[:, 1],
            c=np.arange(N)[::-1], cmap='Reds',
            alpha=.2, label='SGRLD')
ax4.legend()
```

```
fig.savefig('trimodal Gaussian.png')

"""### QuadraModal Gaussian"""

# true data generation process (QudraModal)
N=10000
mu1 = torch.zeros(2)
mu2 = torch.tensor([5,0])
mu3 = torch.tensor([5,5])
mu4 = torch.tensor([0,5])
mus = [mu1,mu2,mu3,mu4]
sigma1 = torch.tensor([[1, 0.],
                       [0., 1]])
sigma2 = torch.tensor([[1,0.],
                       [0.,1]])
sigma3 = torch.tensor([[1, 0.],
                       [0., 1]])
sigma4 = torch.tensor([[1,0.],
                       [.0,1]])
sigmas = [sigma1, sigma2, sigma3, sigma4]
dist4 = MultiModalNormal(mus,sigmas)
original_samples4 = dist4.sample(N)
# true density plot
sns.kdeplot(original_samples4[:,0], original_samples4[:,1])

# SGLD
x = torch.tensor([-10., 0.], requires_grad=True)
sgld = SGLD([x], eta=1e-1, log_density=dist4.loss)
samples_SGLD = []

for epoch in tqdm.tqdm(range(N)):
    x = sgld.sample(x)
    samples_SGLD.append(x.data.clone().detach().T)

samples_SGLD = np.array(np.vstack(samples_SGLD))

# SGRLD
x = torch.tensor([-10., 0.], requires_grad=True)
sgrld = SGRLD([x], eta=1e-1, log_density=dist4.loss)
samples_SGRLD = []

for epoch in tqdm.tqdm(range(N)):
    x = sgrld.sample(x)
```

```
    samples_SGRLD.append(x.data.clone().detach().T)

samples_SGRLD = np.array(np.vstack(samples_SGRLD))

# SGHMC
x = torch.tensor([-10., 0.], requires_grad=True)
sghmc = SGHMC([x], alpha=0.01, eta=1e-1, log_density=dist4.loss)
samples_SGHMC = []

for epoch in tqdm.tqdm(range(N)):
    x = sghmc.sample(x)
    samples_SGHMC.append(x.detach().clone().T)

samples_SGHMC = np.array(np.vstack(samples_SGHMC))

fig = plt.figure()

# true density plot
ax1 = fig.add_subplot(2,2,1)
#plt.subplot(221)
ax1.scatter(original_samples4[:,0], original_samples4[:,1],
                c=np.arange(N)[::-1], cmap='Reds',
                alpha=.2,label='True samples')
sns.kdeplot(original_samples4[:,0], original_samples4[:,1])
ax1.legend()

#niter=1000
#plt.subplot(222)
# SGHMC plot
ax2 = fig.add_subplot(2,2,2)
sns.kdeplot(original_samples4[:, 0], original_samples4[:, 1])
ax2.scatter(samples_SGHMC[:, 0], samples_SGHMC[:, 1],
            c=np.arange(N)[::-1], cmap='Reds',
            alpha=.2, label='SGHMC')
ax2.legend()

# SGLD
ax3 = fig.add_subplot(2,2,3)
#plt.subplot(223)
sns.kdeplot(original_samples4[:, 0], original_samples4[:, 1])
ax3.scatter(samples_SGLD[:, 0], samples_SGLD[:, 1],
            c=np.arange(N)[::-1], cmap='Reds',
            alpha=.2, label='SGLD')
ax3.legend()
```

```
# SGRLD
ax4 = fig.add_subplot(2,2,4)
#plt.subplot(224)
sns.kdeplot(original_samples4[:, 0], original_samples4[:, 1])
ax4.scatter(samples_SGRLD[:, 0], samples_SGRLD[:, 1],
                c=np.arange(N)[::-1], cmap='Reds',
                alpha=.2, label='SGRLD')
ax4.legend()


fig.savefig('quadramodal Gaussian.png')

"""### Writing """
```