Reproducing the paper: *Stochastic Gradient Hamiltonian Monte Carlo* by Tianqi Chen, Emily B. Fox and Carlos Guestrin

Sam Adam-Day, Alexander Goodall, Theo Lewy and Fanqi Xu

Abstract

We reproduce the experiments contained in **sghmc** [**sghmc**] by **sghmc**.

FiXme: Give more details in abstract

List of Corrections

1 Introduction

- Overview of paper and its context.
- Which experiments replicated, and rationale for this choice.
- Target questions of paper.
- Experimental methodology.
- Implementation details.
  - Integration with Pyro.
  - Which parts are new, and which are from publicly available code?
  - Details about how key aspects were implemented.
- Link to repository.
- New aspects?

2 Background

Hamiltonian Monte Carlo (HMC) ([**duane-hmc**; **neal-hmc**]) is a Markov Chain Monte Carlo (MCMC) sampling algorithm. Given a target probability distribution — in our case the posterior distribution of a set of variables θ given independent observations $x \in \mathcal{D}$ — it produces samples by carrying out a random walk over the parameter space using Hamiltonian dynamics.

To begin with prior distribution $p(\theta)$ and likelihood $p(x \mid \theta)$. Using these we define the *potential energy* function U:

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

Note that, using Bayes' rule, we have that the posterior $p(\theta \mid \mathscr{D}) \propto \exp(-U)$. Hamiltonian dynamics introduces an auxiliary set of momentum variables r. These dynamics have a physical interpretation in which an object moves about a landscape determined by U. We let this object have *mass matrix* M. Then $U(\theta)$ represents the potential energy of the object, and its kinetic energy is given by $\frac{1}{2}r^{\mathsf{T}}M^{-1}r$. The total energy of the system is a quantity known as the *Hamiltonian function*:

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

The development of the system is governed by the following equations.

$$\begin{aligned} \mathrm{d}\theta &= M^{-1}r\,\mathrm{d}t \\ \mathrm{d}r &= -\nabla U(\theta)\,\mathrm{d}t \end{aligned}$$

To simulate these continuous dynamics in practice, we must use a discretised version of these equations. To correct for the inaccuracies introduced by doing so, it is necessary to make a *Metropolis-Hastings correction step*. A simple algorithm is given in Algorithm 1.

Algorithm 1 A simple HMC algorithm

```
for $t = 1, 2, \dots$ do
    $r \sim \mathcal{N}(0, 1)$                                  ▷ Resample momentum
    $(\theta_0, r_0) = (\theta, r)$
    for $i = 1$ to $m$ do
        $\theta \leftarrow \theta + \epsilon M^{-1} r$
        $r \leftarrow r - \epsilon \nabla U(\theta)$
    end for
    $u \sim$ Uniform$[0, 1]$
    $\rho = \exp(H(\theta, r) - H(\theta_0, r_0))$              ▷ Acceptance probability
    if $u > \min(1, \rho)$ then                  ▷ Only accept new state with probability $\rho$
        $\theta = \theta_0$
    end if
end for
```

In practice, the dataset $\mathscr{D}$ may be large, and so running Algorithm 1 may be computationally expensive. One idea to combat this is to simulate the Hamiltonian system using only a subset of the data at a time, in analogy with stochastic gradient descent. Unfortunately, such a dynamical system can diverge quite rapidly from the true posterior distribution [**neal-hmc**], which necessitates frequent Metropolis-Hastings steps. Such steps are costly since they must be carried out using the whole dataset. The method 'Stochastic Gradient Hamiltonian Monte Carlo' (SGHMC) proposed in [**sghmc**] addresses this shortcoming. The idea is to incorporate friction into the dynamical system, which works to counteract the noise introduced by selecting a subset of the data.

To specify the SGHMC method, consider sampling a minibatch $\widetilde{\mathscr{D}} \subset \mathscr{D}$ uniformly at random. We estimate the gradient $\nabla U(\theta)$ using this minibatch as follows:

$$\nabla \widetilde{U}(\theta) = -\frac{|\mathscr{D}|}{|\widetilde{\mathscr{D}}|} \sum_{x \in \widetilde{\mathscr{D}}} \nabla \log p(x \mid \theta) - \nabla \log p(\theta)$$

Appealing to the Central Limit Theorem, we can take the noisy estimate $\nabla \widetilde{U}(\theta)$ to be normally distributed about $\nabla U(\theta)$, with some covariance matrix $V(\theta)$.

The naïve adaptation of HMC to this stochastic scenario simply replaces $\nabla U(\theta)$ with $\nabla \widetilde{U}(\theta)$ in Algorithm 1. The corresponding discrete system is then the ϵ-discretisation of the following dynamics:

$$\begin{aligned} \mathrm{d}\theta &= M^{-1} r \,\mathrm{d}t \\ \mathrm{d}r &= -\nabla U(\theta)\,\mathrm{d}t + \mathcal{N}(0, 2B(\theta)\,\mathrm{d}t) \end{aligned}$$

where $B(\theta) = \frac{1}{2}\epsilon V(\theta)$.

The SGHMC method adds a 'friction term' $BM^{-1}r$ to the momentum update. Since we are unlikely to know the noise model B in practice, we instead take an estimate $\widehat{B}$ of B, together with a user-specified friction term $C \succeq \widehat{B}$ and simulate the following dynamics.

$$\begin{aligned} \mathrm{d}\theta &= M^{-1} r \,\mathrm{d}t \\ \mathrm{d}r &= -\nabla U(\theta)\,\mathrm{d}t - CM^{-1} r \,\mathrm{d}t + \mathcal{N}(0, 2(C - \widehat{B}(\theta))\,\mathrm{d}t) + \mathcal{N}(0, 2B(\theta)\,\mathrm{d}t) \end{aligned}$$

The algorithm is given in Algorithm 2. In the case where $\widehat{B} = B$, these dynamics accurately traverse from the posterior distribution. In practice, we must rely on an inaccurate estimate $\widehat{B}$. The simplest choice is $\widehat{B} = 0$. A better but more costly estimate is $\widehat{B}(\theta) = \frac{1}{2}\epsilon\widehat{V}(\theta)$, where $\widehat{V}$ is the observed (empirical Fisher) information [**sgld-fisher**]. The friction term C can then be taken as a hyperparameter, and set so as to counteract the inaccuracies of the estimate $\widehat{B}$.

Algorithm 2 The SGHMC algorithm

```
for $t = 1, 2, \dots$ do
    $r \sim \mathcal{N}(0, 1)$                                      ▷ Resample momentum
    for $i = 1$ to $m$ do
        $\theta \leftarrow \theta + \epsilon M^{-1} r$
        $r \leftarrow r - \epsilon \nabla \widetilde{U}(\theta) - \epsilon C M^{-1} r + \mathcal{N}(0, 2(C - \widehat{B}(\theta))\epsilon)$
    end for
end for
```

3 Implementation Details

We implemented the following algorithms from scratch: HMC, SGHMC, SGLD, SGD, SGD with Nesterov momentum and SGNUTS - a novel extension to SGHMC that uses ideas from the popular No U-Turn Sampler [**nuts**]. All of our implementations subclass Pyro's `MCMCKernel` and are designed to be used directly with Pyro [**pyro**] - a universal probabilistic programming language (PPL) written in Python. In Pyro the user specifies a model which is a probabilistic program (PP) that describes a posterior distribution $p(\theta \mid D)$ that we want to sample from; θ corresponds to the sampled parameters or latent parameters of the model and D corresponds to the observed parameters of the model.

Pyro already comes with the following MCMC samplers: HMC, MH and NUTS. So while the algorithms we implemented are not novel they are in fact innovative since Pyro doesn't come with any of them already built. The main reason for choosing to implement our algorithms on top of Pyro is because Pyro comes with a method `initialize_model` that given a Pyro PP or model P transforms it into a potential function U. Once we have U we can pass the latent and observed parameters (θ, D) to U, which computes the negative log joint $-\log p(\theta, D)$. Bayes' rule tells us that $p(\theta \mid D) \propto p(\theta, D)$ which is typically all we need for MCMC samplers and even simpler ones such as Importance and Rejection samplers. Pyro also comes with the method `potential_grad`, which given (θ, D) computes the gradient of U with respect to the parameters θ. The result is that we can specify any arbitrary PP and apply our algorithms to them - letting Pyro handle the transformation from PP to potential function and the gradient computations.

In traditional MCMC samplers the observed dataset D is constant and so once a Pyro PP has been transformed into a potential function $U(\theta) = -\log p(\theta, D)$ we need not change it. Unfortunately this is less straight forward for stochastic gradient samplers such as SGHMC and SGLD since we subsample the full dataset D by sampling minibatches $\tilde{D}$, where $\tilde{D} \subset D$. In both SGHMC and SGLD we require that the potential function has the form,

$$\tilde{U}(\theta) = -\frac{|D|}{|\tilde{D}|} \log p(\tilde{D} \mid \theta) - \log p(\theta)$$

Unfortunately when we subsample the dataset and call `initialize_model` Pyro doesn't explicitly supply us with the likelihood term $p(\tilde{D} \mid \theta)$ - it only gives us the negative log joint $-\log p(\theta, \tilde{D})$, so we had to modify Pyro's source code to get the desired behaviour above. As a result every time we generate a new sample using SGHMC or SGLD we have to call `initialize_model` so that it gives us the correct $\tilde{U}(\theta)$ for some given minibatch $\tilde{D}$, although this is a small price to pay for much quicker gradient computations.

4 Experiments

- Describe experiments and compare with results in the paper.

4.1 Simulated examples

4.2 Bayesian Neural Networks for Classification

For the Bayesian neural network (BNN) MNIST classification we actually used the reparameterisation of SGHMC described in the section "Connection to SGD with Momentum" of [**sghmc**]. The SGHMC algorithm is reframed in terms of learning rate and momentum decay, and simulates the following dynamics instead,

$$\begin{cases} \nabla\theta = v \\ \nabla v = -\eta \nabla \tilde{U}(\theta) - \alpha v + \mathcal{N}(0, 2(\alpha - \hat{\beta})\eta) \end{cases}$$

Where $\eta = \epsilon^2 M^{-1}, \alpha = \epsilon M^{-1} C, \hat{\beta} = \eta M^{-1} \hat{B}$. In all our experiments in this section we set the mass matrix M to the identity, and the noise model $\hat{\beta} = \hat{B} = 0$. Other than the architecture of the BNN there are now only 3 hyperparameters for SGHMC, the learning rate η, the momentum decay α and the batch size $|\tilde{D}|$, in all our experiments we fixed $|\tilde{D}| = 500$ which follows from [**sghmc**].

To build on top of the work in [**sghmc**] we implemented learning rate annealing for SGLD and following [**sgld**] we weighted the samples by the learning rate as follows,

$$\mathbb{E}[f(\theta)] \approx \frac{\sum_{t=1}^{T} \epsilon_t f(\theta_t)}{\sum_{t=1}^{T} \epsilon_t}$$

Although since f is a classifier we can ignore the denominator. Our BNN followed the same architecture as in [**sghmc**], that is one linear with 100 hidden units followed by ReLU activation followed by another linear layer and a log softmax for multi-class classification. The weights and biases for both linear layers are sampled from univariate standard normal distributions, but the Pyro method `to_event()` declares dependence between the parameters.

Our implementations of SGD and SGD with momentum are meant to be used directly with Pyro, and so Gaussian priors on the weights and biases is equivalent to L2 regularization in the non-Bayesian paradigm. We experimented with regularization strengths of $\lambda \in \{0.1, 1.0, 10.0\}$ and found $\lambda = 1.0$ to be the most effective. Additionally we implemented weight decay for both SGD and SGD with momentum but found that this didn't improve anything in this setting.

For the momentum based algorithms, SGHMC and SGD with momentum, we tried $\eta \in \{1.0, 2.0, 4.0, 8.0\} \times 10^{-6}$, and $\alpha \in \{0.1, 0.01, 0.001\}$. For SGHMC the best configuration was $\eta = 2.0^{-6}, \alpha = 0.01$, and for SGD with momentum the best configuration was $\eta = 1.0^{-6}, \alpha = 0.01$.

For SGLD and SGD, we tried $\eta \in \{1.0, 2.0, 4.0, 6.0\} \times 10^{-5}$, for SGLD we also tried learning rate annealing but it proved not to make much of a difference in this setting so we ignored it in the end. The best configuration for SGLD was $\eta = 4.0^{-5}$, and for SGD the best configuration was $\eta = 1.0 \times 10^{-5}$.

For MNIST we ran each of the algorithms for 800 epochs with 50 warmup epochs. For the sampling algorithms we do Bayesian averaging over all the sampled parameterisations of the BNN after warmup as is described in Section II of [**hands-on-bnn**] and report the test error. For the optimization algorithms we take most recent sample / set of parameters as a point estimate and report the test error. Figure 1 presents our results.

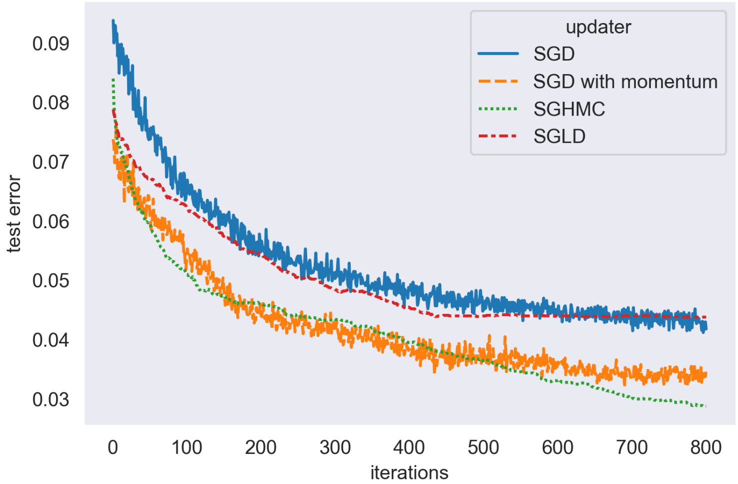
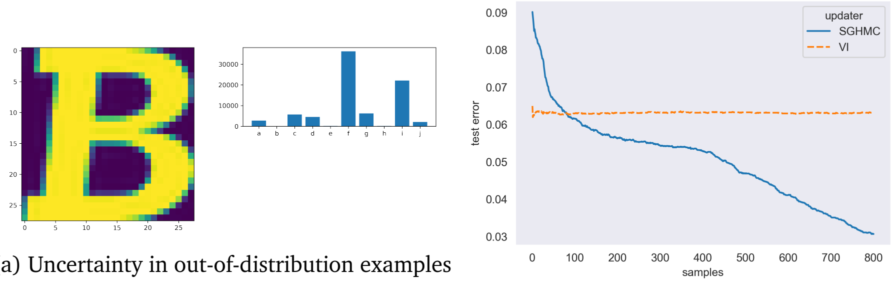


Figure 1: Reproducing the MNIST classification experiment from [**sghmc**]; SGHMC ($\eta = 2.0^{-6}, \alpha = 0.01,$ `resample_n` $= 0$), SGLD ($\eta = 4.0^{-5}$), SGD ($\eta = 1.0^{-5}$), SGD with momentum ($\eta = 1.0^{-6}, \alpha = 0.01$)

The results we get from MNIST classification align very closely with those in [**sghmc**] and so we come to the same conclusion; the need for scalable and efficient Bayesian inference algorithms. The key benefit of BNNs is that we are not overconfident on out-of-distribution examples, Figure 2a illustrates that we still maintain this property when using SGHMC to approximately sample from the posterior distribution. We additionally conducted a brief comparison



(a) Uncertainty in out-of-distribution examples

(b) VI and SGHMC on MNIST

Figure 2: **Left** (a) illustrates that we get uncertainty estimates on out-of-distribution examples. **Right** (b) compares VI (Renyi ELBO, $\alpha = 0.01$, $\texttt{num\_particles} = 2$) and SGHMC ($\eta = 2.0^{-6}, \alpha = 0.01, \texttt{resample\_n} = 0$) on MNIST. For VI we draw 80000 samples from the variational posterior q_ϕ and report the test error by Bayesian averaging. For SGHMC we do the same, except we are approximately sampling from the true posterior $p(\theta\ |D\ )$.

between Variational Inference (VI) and SGHMC in this setting, Figure 2b outlines our findings.

The initial results suggest that SGHMC performs better than VI in this setting, although this is not the full picture. Once VI fits the variational posterior distribution q_ϕ as closely as possible to the true posterior it takes only hundreds of samples to characterise q_ϕ, whereas SGHMC requires several more samples to characterise the true posterior. In practice storing thousands of parameterisations of the same NN is very costly and so this is probably why VI is a more popular choice for Bayesian inference.

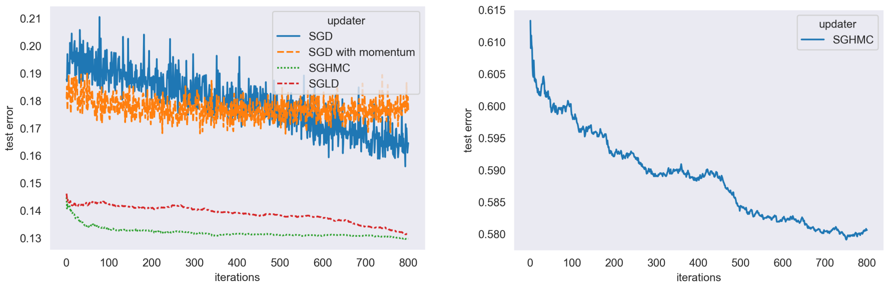
We conclude this section by demonstrating our algorithms can be applied to other datasets and more complicated models, Figure 3a presents the results of running the same BNN architecture on FashionMNIST, and Figure 3b demonstrates that our implementation of SGHMC can be used with convolutional neural networks (CNNs).

5 NUTS

In our current description of the SGHMC algorithm we have the user-defined hyperparameter L, the number of steps iterated over before we take a sample. If this number is too small, our samples will be correlated, and hence successive samples would appear to follow a random walk, and we would get slow mixing times. We demonstrate this behaviour by training 3 versions of SGHMC with $L = 1, 3, 5$ (with ϵL fixed). The learning curves below show that increasing L increases the speed with which SGHMC reaches low error rates.

However, if L is too large we waste computational power, as we continue to step through time even though each sample is already independent of the last. We would like to set L to its optimal value.

Hoffman and Gelman introduce the algorithm NUTS in their paper "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo" [**nuts**]. In its basic form this algorithm removes the need for a user to input a value for L in the standard HMC implementation. We converted this NUTS algorithm from being based on HMC to being based on SGHMC, and



(a) Experiment on FashionMNIST

(b) Convolutional BNN on CIFAR10

Figure 3: **Left** (a) FashionMNIST classification; SGHMC ($\eta = 1.0^{-6}, \alpha = 0.01, \texttt{resample\_n} = 0$), SGLD ($\eta = 1.0^{-5}$), SGD ($\eta = 1.0^{-5}$), SGD with momentum ($\eta = 1.0^{-6}, \alpha = 0.01$); $\texttt{warmup\_epochs} = 400$. **Right** (b) Convolutional BNN with 2 convolutional layers, batch norm, max pooling and tanh activations followed by two Bayesian linear layers with tanh activation; SGHMC ($\eta = 1.0^{-6}, \alpha = 0.01, \texttt{resample\_n} = 0$) ; $\texttt{warmup\_epochs} = 150$.

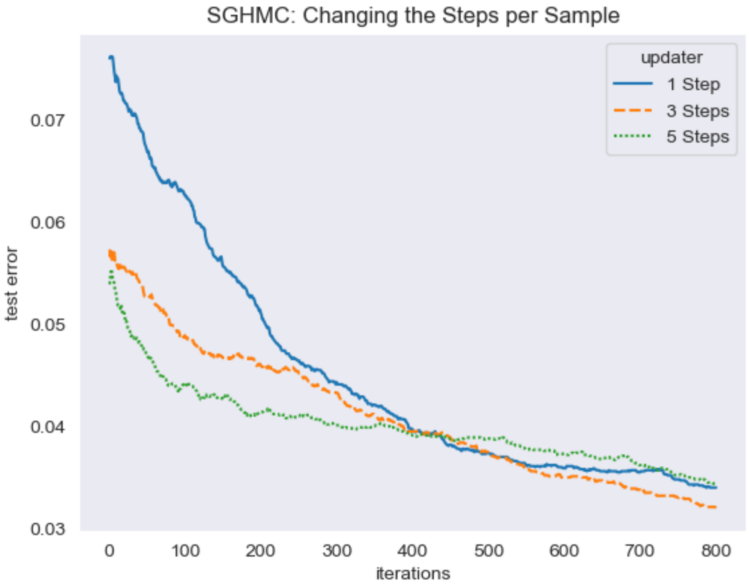


Figure 4: Changing the number of steps per sample. Each agent was run for 100 warmup epochs.

investigated its power. We will begin by presenting the high level overview of the original NUTS algorithm for HMC sampling.

The key idea is the concept of a ’U-Turn’. This is the point at which the samples of θ start to get closer to the initial value of θ, rather than away from it. This marks the point at which further steps will likely only waste computational power. Mathematically, for current position θ, initial position θ_0 and momentum r, this corresponds to the time at which

$$\frac{d}{dt}|\theta - \theta_0|^2 = 2(\theta - \theta_0) \cdot \frac{d\theta}{dt} = 2(\theta - \theta_0) \cdot r < 0$$

where we use the fact that in the dynamics of HMC (and also SGHMC) we have $\frac{d\theta}{dt} = r$. This simple fact suggests an algorithm in which we draw the sample θ once we have performed enough steps so that $(\theta - \theta_0) \cdot r < 0$. However this is too simplistic an approach - as HMC is an instance of the Metropolis Hastings algorithm we require the Markov Chain of θ to be reversible, which is not the case here.

To remedy this problem, NUTS requires keeping track of a set $\mathscr{B}$, which contains all values of (θ, r) as steps are taken both forward and backwards in time. The values at the earliest and latest times considered across a single trajectory are labelled (θ^-, r^-) and (θ^+, r^+) respectively. NUTS starts from a single (θ, r) and then steps either forward or backwards one step. It then steps forward or backwards 2 steps, then 4 steps, then 8 steps etc until a 'U-Turn' is seen. A description is as follows:

```
1. Start with an initial $(\theta_0, r_0)$
2. Set $n \leftarrow 1$, $\mathscr{B} \leftarrow \{(\theta_0, r_0)\}$, $(\theta^-, r^-) \leftarrow (\theta_0, r_0)$ and $(\theta^+, r^+) \leftarrow (\theta_0, r_0)$
3. While there is no U-Turn at $(\theta^-, r^-)$ nor at $(\theta^+, r^+)$
   (ie $(\theta^+ - \theta^-) \cdot r^- \geqslant 0$ and $(\theta^+ - \theta^-) \cdot r^+ \geqslant 0$):
   (a) Choose to go *forwards* or *backwards* in time, each with probability $\frac{1}{2}$
   (b) If *forwards*:
       i. $(\theta, r) \leftarrow (\theta^+, r^+)$
       ii. Repeat n times:
           A. $(\theta, r) \leftarrow$ Step forward in time from $(\theta, r)$
           B. $\mathscr{B} \leftarrow (\theta, r) \cup \mathscr{B}$
       iii. $(\theta^+, r^+) \leftarrow (\theta, r)$
   (c) If *backwards*:
       i. $(\theta, r) \leftarrow (\theta^-, r^-)$
       ii. Repeat n times:
           A. $(\theta, r) \leftarrow$ Step backwards in time from $(\theta, r)$
           B. $\mathscr{B} \leftarrow (\theta, r) \cup \mathscr{B}$
       iii. $(\theta^-, r^-) \leftarrow (\theta, r)$
   (d) $n \leftarrow 2n$
4. Carefully choose a subset $\mathscr{C} \subseteq \mathscr{B}$. This step is the key to the Markov Chain
   being reversible, however for clarity's sake we will not go into detail as to
   how this is done here.
5. Sample an element of $\mathscr{C}$
```

The benefit to this algorithm is that it removes the need to set the number of steps performed before we sample, as we keep stepping until a 'U-Turn' is seen. We should note here that in the original form of NUTS, the 'step' being referred to in $3biiA$ and $3ciiA$ is the step of the HMC algorithm. We edited the NUTS Pyro source code to make it perform SGHMC steps, and we named this SGNUTS.

There were some doubts as to whether the NUTS algorithm would work when using SGHMC steps instead of HMC steps. This was because NUTS requires the ability to step backwards in time, while an SGHMC step includes an injection of stochastic noise. As there is no action that undoes this injection, there was a worry that the backwards step through time in NUTS would become a problem. We explain how we attempted to solve this problem at the end of the next section.

5.1 Implementation of SGNUTS

To build SGNUTS we started with the Pyro source code for NUTS [**nuts_code**] which takes the Pyro HMC class as its parent. We altered this to instead inherit from our SGHMC class. This required removing step-size adaptation and mass matrix adaptation functionality from NUTS, as our implementation of SGHMC was not able to interface with this. It also required introducing some caching methods into the SGHMC class - to help keep things simple in our original SGHMC class we did this in a new class, named SGHMC_for_NUTS. Most importantly, we had to alter the (θ, v) step update rule which is hardcoded in NUTS. This meant that instead of being the HMC update step it was now the SGHMC update step of:

$$\theta \leftarrow \begin{cases} \theta + \epsilon M^{-1} r, & \textit{forwards step} \\ \theta - \epsilon M^{-1} r, & \textit{backwards step} \end{cases}$$

$$r \leftarrow \begin{cases} r - \epsilon \nabla \tilde{U}(\theta) - \epsilon C M^{-1} r + \mathcal{N}(0, 2(C - \hat{B})\epsilon), & \textit{forwards step} \\ r + \epsilon \nabla \tilde{U}(\theta) - \epsilon C M^{-1} r + \mathcal{N}(0, 2(C - \hat{B})\epsilon), & \textit{backwards step} \end{cases}$$

In particular, note that there is an injection of stochastic noise and momentum-reducing friction in both the forward steps and the backward steps. This is not ideal, as our backwards step in time does not undo a forward step, which likely breaks the reversibility of the Markov Chain being considered in NUTS. However we investigated this nonetheless.

5.2 Results

We tested the SGNUTS algorithm on the BNNs described earlier. We ran SGNUTS for only 20 warmup epoch and 50 epochs as the code was slower than SGHMC to run. The learning curves were as follows:

The final accuracies were 0.94 for MNIST and 0.85 for FashionMNIST which are similar to the accuracies obtained using SGNUTS (0.97 and 0.86 respectively). This accuracy demonstrates that there is convergence to the true posterior in SGNUTS. This suggests that, like SGHMC itself, the SGNUTS Markov Chain is likely reversible in some (non traditional) sense. Denoting the posterior as $\pi(\theta, r)$, and transition kernels as $P(\theta, r|\theta', r')$, it is shown in ([**sghmc**]) that SGHMC satisfies

$$\pi(\theta, r) P_{SGHMC}(\theta, r|\theta', r') = \pi(\theta', -r') P_{SGHMC}(\theta', -r'|\theta, -r)$$

and it is suggested that this is the reason why the SGHMC algorithm works, despite not being reversible in the traditional sense. It would be interesting to consider if a similar property holds for P_{SGNUTS}.

While accurate, SGNUTS is slower to run than SGHMC to produce a single sample. It does however reach relatively high accuracies in a short amount of time. We measured the accuracy after the first epoch of SGHMC and NUTS as we changed the number of warmup epochs, and we measured how long they took to train.

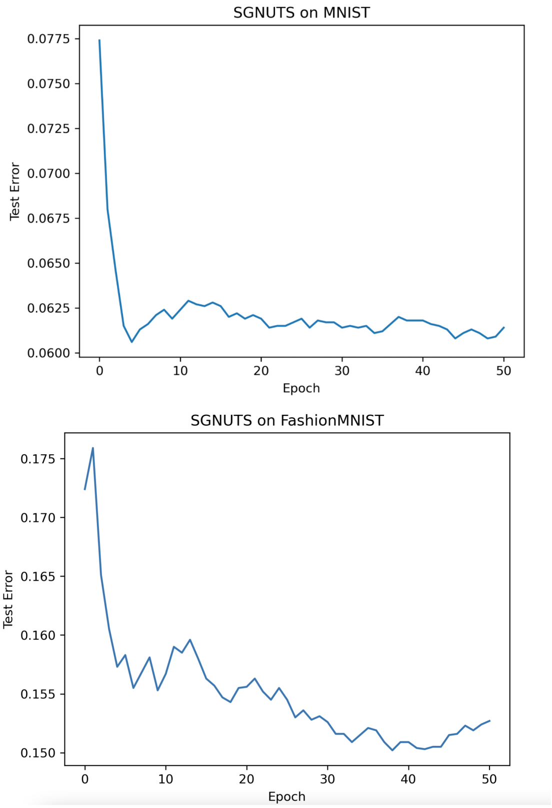


Figure 5: Classifying MNIST and FashionMNIST with SGNUTS

Number of Warmup-Epochs	SGHMC		SGNUTS	
	Accuracy	Time (s)	Accuracy	Time (s)
0	0.7747	3.2	0.8715	28.3
1	0.5764	4.5	0.9296	127.2
2	0.4141	6.2	0.9231	453.3
5	0.6471	11.4		
10	0.7100	20.3		
25	0.8866	50.2		

In particular, SGNUTS reached an accuracy of 0.87 in 28s, while SGHMC reached 0.89 in 50s. This suggests a new algorithm that uses both SGNUTS and SGHMC - we could run a single epoch of SGNUTS so that we quickly approach the posterior distribution, at which point we switch to running SGHMC. It should be noted the speed up suggested by the above results would only be 30s seconds, however maybe on more complex datasets this could be larger. It would be interesting to investigate this if we had more time.

6 Conclusion

- Analysis and discussion of findings.
- Suggest what could have been done with more time.