sgmcmc: An R Package for Stochastic Gradient Markov Chain Monte Carlo

Jack Baker[1*] Paul Fearnhead[1] Emily B. Fox[2] Christopher Nemeth[1]

[1] STOR-i Centre for Doctoral Training, Department of Mathematics and Statistics, Lancaster University, Lancaster, UK
[2] Department of Statistics, University of Washington, Seattle, WA

Abstract

This paper introduces the R package **sgmcmc**; which can be used for Bayesian inference on problems with large datasets using stochastic gradient Markov chain Monte Carlo (SGMCMC). Traditional Markov chain Monte Carlo (MCMC) methods, such as Metropolis-Hastings, are known to run prohibitively slowly as the dataset size increases. SGMCMC solves this issue by only using a subset of data at each iteration. SGMCMC requires calculating gradients of the log likelihood and log priors, which can be time consuming and error prone to perform by hand. The **sgmcmc** package calculates these gradients itself using automatic differentiation, making the implementation of these methods much easier. To do this, the package uses the software library **TensorFlow**, which has a variety of statistical distributions and mathematical operations as standard, meaning a wide class of models can be built using this framework. SGMCMC has become widely adopted in the machine learning literature, but less so in the statistics community. We believe this may be partly due to lack of software; this package aims to bridge this gap.

Keywords: R, stochastic gradient Markov chain Monte Carlo, big data, MCMC, stochastic gradient Langevin dynamics, stochastic gradient Hamiltonian Monte Carlo, stochastic gradient Nosé-Hoover thermostat

1 Introduction

This article introduces **sgmcmc**, an R package (R Development Core Team, 2008) for scalable Bayesian inference on a wide variety of models using stochastic gradient Markov chain Monte Carlo (SGMCMC). A disadvantage of most traditional MCMC methods are that they require calculations over the full dataset at each iteration; meaning the methods are prohibitively slow for large datasets. SGMCMC methods provide a solution to this issue. The methods use only a subset of the full dataset, known as a *minibatch*, at each MCMC iteration. While the methods no longer target the true posterior, they instead produce accurate approximations to the posterior at a reduced computational cost.

The **sgmcmc** package implements a number of popular SGMCMC samplers including stochastic gradient Langevin dynamics (SGLD) (Welling and Teh, 2011), stochastic gradient Hamiltonian Monte Carlo (SGHMC) (Chen et al., 2014) and stochastic gradient Nosé-Hoover thermostats (SGNHT) (Ding et al., 2014). Recent work has shown how control variates can be used to reduce the computational cost of SGMCMC algorithms (Baker et al., 2017; Nagapetyan et al., 2017). For each of the samplers implemented in the package, there is also a corresponding control variate sampler providing improved sampling efficiency.

Performing statistical inference on a model using SGMCMC requires calculating the gradient of the log likelihood and log priors. Calculating gradients by hand is often time consuming and error prone. One

*email: j.baker1@lancaster.ac.uk

of the major advantages of **sgmcmc** is that gradients are calculated within the package using automatic differentiation (Griewank and Walther, 2008). This means that users need only specify the log likelihood function and log prior for their model. The package calculates the gradients using **TensorFlow** (**TensorFlow** Development Team, 2015), which has recently been made available for R (Allaire et al., 2016). **TensorFlow** is an efficient library for numerical computation which can take advantage of a wide variety of architectures, as such, **sgmcmc** keeps much of this efficiency. Both **sgmcmc** and **TensorFlow** are available on CRAN, so **sgmcmc** can be installed by using the standard `install.packages` function. Though after the **TensorFlow** package has been installed, the extra `install_tensorflow()` function needs to be run, which installs the required Python implementation of **TensorFlow**.[1] The **sgmcmc** package also has a website with vignettes, tutorials and an API reference.[2]

SGMCMC methods have become popular in the machine learning literature but less so in the statistics community. We partly attribute this to the lack of available software. To the best of our knowledge, there are currently no R packages available for SGMCMC, probably the most popular programming language within the statistics community. The only package we are aware of which implements scalable MCMC is the Python package **edward** (Tran et al., 2016). This package implements both SGLD and SGHMC, but does not implement SGNHT or any of the control variate methods.

In Section 2 we briefly review the methodology behind the SGMCMC methods implemented in **sgmcmc**. Section 3 provides a brief introduction to **TensorFlow**. Section 4 overviews the structure of the package, as well as details of how the algorithms are implemented. Section 5 presents a variety of example simulations. Finally, Section 6 provides a discussion on benefits and drawbacks of the implementation, as well as how we plan to extend the package in the future.

2 Stochastic gradient MCMC

Suppose we have a dataset of size N, with data $\mathbf{x} = \{x_i\}_{i=1}^N$, where $x_i \in \mathcal{X}$ for some space $\mathcal{X}$. We denote the probability density of x_i as $p(x_i|\theta)$, where $\theta \in \Theta \subseteq \mathbb{R}^p$ are model parameters to be inferred. We assign a prior density $p(\theta)$ to the parameters and the resulting posterior is then $p(\theta|\mathbf{x}) \propto p(\theta)\prod_{i=1}^N p(x_i|\theta)$.

Often the posterior can only be calculated up to a constant of proportionality Z. In practice Z is rarely analytically tractable; so MCMC provides a way to construct a Markov chain using only the unnormalized posterior density $h(\theta) := p(\theta)\prod_{i=1}^N p(x_i|\theta)$. The Markov chain is designed so that its stationary distribution is the posterior $p(\theta|\mathbf{x})$. The result (once the chain has converged) is a sample $\{\theta_t\}_{t=1}^T$ from the posterior, though this sample is not independent. A downside of these traditional MCMC algorithms is that they require the evaluation of the unnormalized density $h(\theta)$ at every iteration. This results in an $O(N)$ cost per iteration. Thus MCMC becomes prohibitively slow on large datasets.

The Metropolis-Hastings algorithm is a type of MCMC algorithm. New proposed samples θ' are drawn from a proposal distribution $q(\theta'|\theta)$ and then accepted with probability,

$$\min\left\{1, \frac{p(\theta'|\mathbf{x})q(\theta|\theta')}{p(\theta|\mathbf{x})q(\theta'|\theta)}\right\}. \tag{1}$$

Notice that the normalising constant Z cancels in (1), so we can interchange the posterior $p(\theta|\mathbf{x})$ with $h(\theta)$. The efficiency of the Metropolis-Hastings algorithm is dependent on the choice of proposal distribution, q. There are a number of proposals for the Metropolis-Hastings algorithm which can have a very high acceptance rate. Some examples are the Metropolis-adjusted Langevin algorithm (MALA; see e.g., Roberts and Rosenthal (1998)) and Hamiltonian Monte Carlo (HMC; see Neal (2010)). The reason these proposals achieve such high acceptance rates is that they approximate a continuous diffusion process whose stationary distribution is $p(\theta|\mathbf{x})$. As an example, consider the Metropolis-adjusted Langevin algorithm (MALA). The MALA algorithm uses a Euler discretisation of the Langevin diffusion as the proposal,

$$q(\theta'|\theta) = \mathcal{N}\left(\theta' \,|\, \theta + \frac{\epsilon}{2}\nabla_\theta \log p(\theta|\mathbf{x}), \epsilon \mathrm{I}\right), \tag{2}$$

[1] More information on installing **TensorFlow** for R can be found at `https://tensorflow.rstudio.com/`.

[2] **sgmcmc** website at `https://stor-i.github.io/sgmcmc`

where $\mathcal{N}(\theta|\mu, \Sigma)$ denotes a multivariate Normal density evaluated at θ with mean μ and variance Σ; I is simply the identity matrix; ϵ is a tuning parameter referred to as the *stepsize*. Discretising the diffusion introduces an approximation error, which is corrected by the Metropolis-Hastings acceptance step (1). This means that as $\epsilon \to 0$, we tend back towards the exact, continuous diffusion and the acceptance rate is 1. However this would result in a Markov chain that never moves. Thus picking ϵ is a balance between a high acceptance rate and good mixing.

Many popular MCMC proposal distributions, including MALA and HMC, require the calculation of the gradient of the log posterior at each iteration, which is an $O(N)$ calculation. Stochastic gradient MCMC methods get around this by replacing the true gradient with the following unbiased estimate

$$\nabla_\theta \widehat{\log p(\theta_t|\mathbf{x})} := \nabla_\theta \log p(\theta_t) + \frac{N}{n} \sum_{i \in S_t} \nabla_\theta \log p(x_i|\theta_t), \tag{3}$$

calculated on some subset of the all observations $S_t \subset \{1, \ldots, N\}$, known as a *minibatch*, with $|S_t| = n$.

Calculating the Metropolis-Hastings acceptance step (1) is another $O(N)$ calculation. To get around this, SGMCMC methods set the tuning constants such that the acceptance rate will be high, and remove the acceptance step from the algorithm altogether. By ignoring the acceptance step, and estimating the log posterior gradient, the per iteration cost of SGMCMC algorithms is $O(n)$, where n is the minibatch size. However, the algorithm no longer targets the true posterior but an approximation. There has been a body of theory exploring how these methods perform in different settings. Similar to MALA, the algorithms rely on a stepsize parameter ϵ. Some of the algorithms have been shown to weakly converge as $\epsilon \to 0$.

2.1 Stochastic gradient Langevin dynamics

SGLD, first introduced by Welling and Teh (2011), is an SGMCMC approximation to the Metropolis-adjusted Langevin algorithm (MALA). By substituting (3) into the MALA proposal equation (2), we arrive at the following update for θ

$$\theta_{t+1} = \theta_t + \frac{\epsilon_t}{2} \nabla_\theta \widehat{\log p(\theta_t|\mathbf{x})} + \zeta_t, \tag{4}$$

where $\zeta_t \sim \mathcal{N}(0, \epsilon_t)$.

Welling and Teh (2011) noticed that as $\epsilon_t \to 0$ this update will sample from the true posterior. Although the algorithm mixes slower as ϵ gets closer to 0, so setting the stepsize is a trade-off between convergence speed and accuracy. This motivated Welling and Teh (2011) to suggest setting ϵ_t to decrease to 0 as t increases. There is a body of theory that investigates the SGLD approximation to the true posterior (see e.g., Teh et al., 2016; Sato and Nakagawa, 2014; Vollmer et al., 2016). In particular, SGLD is found to converge weakly to the true posterior distribution asymptotically as $\epsilon_t \to 0$. The mean squared error of the algorithm is found to decrease at best with rate $O(T^{-1/3})$. In practice, the algorithm tends to mix poorly when ϵ is set to decrease to 0 (Vollmer et al., 2016), so in our implementation we use a fixed stepsize which has been shown to mix better empirically. Theoretical analysis for this case is provided in Vollmer et al. (2016). The tuning constant ϵ, referred to as the *stepsize* is a required argument in the package. It affects the performance of the algorithm considerably.

2.2 Stochastic gradient Hamiltonian Monte Carlo

The stochastic gradient Hamiltonian Monte Carlo algorithm (SGHMC) (Chen et al., 2014) is similar to SGLD, but instead approximates the Hamiltonian Monte Carlo (HMC) algorithm (Neal, 2010). To ensure SGHMC is $O(n)$, the same unbiased estimate to the log posterior gradient is used (3). SGHMC augments the parameter space with momentum variables ν and samples approximately from a joint distribution $p(\theta, \nu|\mathbf{x})$, whose marginal distribution for θ is the posterior of interest. The algorithm performs the following updates at each iteration

$$\begin{aligned}
\theta_{t+1} &= \theta_t + \nu_t, \\
\nu_{t+1} &= (1 - \alpha)\nu_t + \epsilon \nabla_\theta \widehat{\log p(\theta_{t+1}|\mathbf{x})} + \zeta_t,
\end{aligned}$$

where $\zeta_t \sim \mathcal{N}(0, 2[\alpha - \hat{\beta}_t]\epsilon)$; ϵ and α are tuning constants and $\hat{\beta}_t$ is proportional to an estimate of the Fisher information matrix. In our current implementation, we simply set $\hat{\beta}_t := 0$, as in the experiments of the original paper by Chen et al. (2014). In future implementations, we aim to estimate $\hat{\beta}_t$ using a Fisher scoring estimate similar to Ahn et al. (2012). Often the dynamics are simulated repeatedly L times before the state is stored, at which point ν is resampled. The parameter L can be included in our implementation. The tuning constant ϵ is the stepsize and is a required argument in our implementation, as for SGLD its value affects performance considerably. The constant α tends to be fixed at a small value in practice. As a result, in our implementation it is an optional argument with default value 0.01.

2.3 Stochastic gradient Nosé-Hoover thermostat

Ding et al. (2014) suggest that the quantity $\hat{\beta}_t$ in SGHMC is difficult to estimate in practice. They appeal to analogues between these proposals and statistical physics in order to suggest a set of updates which do not need this estimation to be made. Once again Ding et al. (2014) augment the space with momentum parameters ν. They replace the tuning constant α with a dynamic parameter α_t known as the *thermostat* parameter. The algorithm performs the following updates at each iteration

$$\theta_{t+1} = \theta_t + \nu_t, \tag{5}$$

$$\nu_{t+1} = (1 - \alpha_t)\nu_t + \epsilon \nabla_\theta \widehat{\log p(\theta_{t+1}|\mathbf{x})} + \zeta_t, \tag{6}$$

$$\alpha_{t+1} = \alpha_t + \left[\frac{1}{p}(\nu_{t+1})^\top(\nu_{t+1}) - \epsilon\right]. \tag{7}$$

Here $\zeta_t \sim \mathcal{N}(0, 2a\epsilon)$; ϵ and a are tuning parameters to be chosen and p is the dimension of θ. The update for α in (7) requires a vector dot product, it is not obvious how to extend this when θ is higher order than a vector, such as a matrix or tensor. In our implementation, when θ is a matrix or tensor we use the standard inner product in those spaces (Abraham et al., 1988). The tuning constant ϵ is the stepsize and is a required argument in our implementation, as again its value affects performance considerably. The constant a, similarly to α in SGHMC, tends to be fixed at a small value in practice (Ding et al., 2014). As a result, in our implementation it is an optional argument with default value 0.01.

2.4 Stochastic gradient MCMC with control variates

A key feature of SGMCMC methods is replacing the log posterior gradient calculation with an unbiased estimate. The unbiased gradient estimate, which can be viewed as a noisy version of the true gradient, can have high variance when estimated using a small minibatch of the data. Increasing the minibatch size will reduce the variance of the gradient estimate, but increase the per iteration computational cost of the SGMCMC algorithm. Recently control variates (Ripley, 2009) have been used to reduce the variance in the gradient estimate of SGMCMC (Dubey et al., 2016; Nagapetyan et al., 2017; Baker et al., 2017). Using these improved gradient estimates have been shown to lead to improvements in the MSE of the algorithm (Dubey et al., 2016), as well as its computational cost (Nagapetyan et al., 2017; Baker et al., 2017).

We implement the formulation of Baker et al. (2017), who replace the gradient estimate $\nabla_\theta \widehat{\log p(\theta|\mathbf{x})}$ with

$$\nabla_\theta \widetilde{\log p(\theta|\mathbf{x})} := \nabla_\theta \log p(\hat{\theta}|\mathbf{x}) + \nabla_\theta \widehat{\log p(\theta|\mathbf{x})} - \nabla_\theta \widehat{\log p(\hat{\theta}|\mathbf{x})}, \tag{8}$$

where $\hat{\theta}$ is an estimate of the posterior mode. This method requires the burn-in phase of MCMC to be replaced by an optimisation step which finds $\hat{\theta} := \text{argmax}_\theta \log p(\theta|\mathbf{x})$. There is then an $O(N)$ preprocessing step to calculate $\nabla_\theta \log p(\hat{\theta}|\mathbf{x})$, after which the chain can be started from $\hat{\theta}$ resulting in a negligible mixing time. Baker et al. (2017) and Nagapetyan et al. (2017) have shown that there are considerable improvements to the computational cost of SGLD when (8) is used in place of (3). In particular they showed that standard SGLD requires setting the minibatch size n to be $O(N)$ for arbitrarily good performance; while using control variates requires an $O(N)$ preprocessing step, but after that a batch size of $O(1)$ can be used to reach the

desired performance. Baker et al. (2017) also showed empirically that this particular formulation can lead to a reduction in the burn-in time compared with standard SGLD and the formulation of (Dubey et al., 2016), which tended to get stuck in local stationary points. This is because in complex scenarios the optimisation step is often faster than the burn-in time of SGMCMC. The package **sgmcmc** includes control variate versions of all the SGMCMC methods implemented: SGLD, SGHMC and SGNHT.

3 Brief TensorFlow introduction

TensorFlow (**TensorFlow** Development Team, 2015) is a software library released by Google. The tool was initially designed to easily build deep learning models; but the efficient design and excellent implementation of automatic differentiation (Griewank and Walther, 2008) has made it useful more generally. This package is built on **TensorFlow**, and while we have tried to make the package as easy as possible to use, some knowledge of **TensorFlow** is unavoidable; especially when declaring the log likelihood and log prior functions, or for high dimensional chains which will not fit into memory. With this in mind, we provide a brief introduction to **TensorFlow** in this section. This should provide enough knowledge for the rest of the article. A more detailed introduction to **TensorFlow** for R can be found at Allaire et al. (2016).

Any procedure written in **TensorFlow** follows three main steps. The first step is to declare all the variables, constants and equations required to run the algorithm. In the background, these declarations enable **TensorFlow** to build a graph of the possible operations, and how they are related to other variables, constants and operations. Once everything has been declared, the **TensorFlow** session is begun and all the variables and operations are initialised. Then the previously declared operations can be run as required; typically these will be sequential and will be run in a for loop.

3.1 Declaring TensorFlow tensors

Everything in **TensorFlow**, including operations, are represented as a tensor; which is basically a multi-dimensional array. There are a number of ways of creating tensors:

- `tf$constant(value)` – create a constant tensor with the same shape and values as `value`. The input `value` is generally an R array, vector or scalar. The most common use for this in the context of the package is for declaring constant parameters when declaring log likelihood and log prior functions.
- `tf$Variable(value)` – create a tensor with the same shape and values as `value`. Unlike `tf$constant`, this type of tensor can be changed by a declared *operation*. The input `value` is generally an R array, vector or scalar.
- `tf$placeholder(datatype, shape)` – create an empty tensor of type `datatype` and dimensions `shape` which can be fed all or part of a dataset, this is useful when declaring operations which rely on data which can change. When you have storage constraints (see Section 4.2) you can use a placeholder to declare test functions that rely on a test dataset. The `datatype` should be a **TensorFlow** data type, such as `tf$float32`; the `shape` should be an R vector or scalar, such as `c(100,2)`.
- *operation* – an operation declares an operation on previously declared tensors. These use **TensorFlow** defined functions, such as those in its math library. This is essentially what you are declaring when coding the `logLik` and `logPrior` functions. The `params` input consists of a list of `tf$Variables`, representing the model parameters to be inferred. The `dataset` input consists of a list of `tf$placeholder` tensors, representing the minibatch of data fed at each iteration. Your job is to declare functions that return an operation on these tensors that define the log likelihood and log prior.

3.2 TensorFlow operations

TensorFlow operations take other tensors as input and manipulate them to reach the desired output. Once the **TensorFlow** session has begun, these operations can be run as needed, and will use the current values for

its input tensors. For example, we could declare a Normal density tensor which manipulates a `tf$Variable` tensor representing the parameters and a `tf$placeholder` tensor representing the current data point. The tensor could then use the `TensorFlow` `tf$contrib$distributions$Normal` object to return a tensor object of the current value for a Normal density given the current parameter value and the data point that's fed to the placeholder. We can break this example down into three steps. First we declare the tensors that we require:

```
library("tensorflow")

# Declare required tensors
loc = tf$Variable(rep(0, 2))
dataPoint = tf$placeholder(tf$float32, c(2))
scaleDiag = tf$constant(c(1, 1))
# Declare density operation tensor
distn = tf$contrib$distributions$MultivariateNormalDiag(loc, scaleDiag)
dens = distn$prob(dataPoint)
```

Here we have declared a `tf$Variable` tensor to hold the location parameter; a `tf$placeholder` tensor which will be fed the current data point; the scale parameter is fixed so we declare this as a `tf$constant` tensor. Next we declare the operation which takes the inputs we just declared and returns the Normal density value. The first line which is assigned to `distn` creates a `MultivariateNormalDiag` object, which is linked to the `loc` and `scaleDiag` tensors. Then `dens` evaluates the density of this distribution. The `dens` variable is now linked to the tensors `dataPoint` and `scaleDiag`, so if it is evaluated it will use the current values of those tensors to calculate the density estimate. Next we begin the **TensorFlow** session:

```
# Begin TensorFlow session, initialize variables
sess = tf$Session()
sess$run(tf$global_variables_initializer())
```

The two lines we just ran starts the **TensorFlow** session and initialises all the tensors we just declared. The **TensorFlow** graph has now been built and no new tensors can now be added. This means that all operations need to be declared before they can be evaluated. Now the session is started we can run the operation `dens` we declared given current values for `dataPoint` and `loc` as follows:

```
# Evaluate density, feeding random data point to placeholder
x = rnorm(2)
out = sess$run(dens, feed_dict = dict( dataPoint = x ) )
print(paste0("Density value for x is ", out))

# Get another random point and repeat!
x = rnorm(2)
out = sess$run(dens, feed_dict = dict( dataPoint = x ) )
print(paste0("Density value for x is ", out))
```

Since `dataPoint` is a `placeholder`, we need to feed it values each time. In the block of code above we feed `dataPoint` a random value simulated from a standard Normal. The `sess$run` expression then evaluates the current Normal density value given `loc` and `dataPoint`.

As mentioned earlier, this is essentially what is happening when you are writing the `logLik` and `logPrior` functions. These functions will be fed a list of `tf$Variable` objects to the `params` input, and a list of `tf$placeholder` objects to the `dataset` input. The output of the function will then be declared as a `TensorFlow` operation. This allows the gradient to be calculated automatically, while maintaining the efficiencies of **TensorFlow**.

Function	Algorithm
`sgld`	Stochastic gradient Langevin dynamics
`sghmc`	Stochastic gradient Hamiltonian Monte Carlo
`sgnht`	Stochastic gradient Nosé-Hoover thermostat
`sgldcv`	Stochastic gradient Langevin dynamics with control variates
`sghmccv`	Stochastic gradient Hamiltonian Monte Carlo with control variates
`sgnhtcv`	Stochastic gradient Nosé-Hoover thermostat with control variates

Table 1: Outline of 6 main functions implemented in **sgmcmc**.

Function inputs	Definition
`logLik`	Log-likelihood function taking `dataset` and `params` as inputs
`dataset`	R list containing data
`params`	R list containing model parameters
`stepsize`	R list of stepsizes for each parameter
`optStepsize`	R numeric stepsize for control variate optimisation step
`logPrior`	Function of the parameters (on the log-scale); default $p(\theta) \propto 1$
`minibatchSize`	Size of minibatch per iteration as integer or proportion; default 0.01.
`nIters`	Number of MCMC iterations; default is 10,000.

Table 2: Outline of the key arguments required by the functions in Table 1.

TensorFlow implements a lot of useful functions to make building these operations easier. For example a number of distributions are implemented at `tf$contrib$distributions`,[3] (e.g., `tf$contrib$distributions$Normal` and `tf$contrib$distributions$Gamma`). **TensorFlow** also has a comprehensive math library which provides a variety of useful tensor operations.[4] For examples of writing **TensorFlow** operations see the worked examples in Section 4 or the **sgmcmc** vignettes.

4 Package structure and implementation

The package has 6 main functions. The first three: `sgld`, `sghmc` and `sgnht` will implement SGLD, SGHMC and SGNHT, respectively. The other three: `sgldcv`, `sghmccv` and `sgnhtcv` implement the control variate versions of SGLD, SGHMC and SGNHT, respectively. All of these are built on the **TensorFlow** library for R, which enables gradients to be automatically calculated and efficient computations to be performed in high dimensions. The usage for these functions is very similar, with the only differences in input being tuning parameters. These main functions are outlined in Table 1

The functions `sgld`, `sghmc` and `sgnht` have the same required inputs: `logLik`, `dataset`, `params` and `stepsize`. These determine respectively: the log likelihood function for the model; the data for the model; the parameters of the model; and the stepsize tuning constants for the SGMCMC algorithm. The input `logLik` is a function taking `dataset` and `params` as input, while the rest are defined as lists. Using lists in this way provides a lot of flexibility: allowing multiple parameters to be defined; use multiple datasets; and use different stepsizes for each parameter, which is vital if the scalings are different. It also allows users to easily reference parameters and datasets in the `logLik` function by simply referring to the relevant names in the list.

The functions also have a couple of optional parameters that are particularly important, `logPrior` and `minibatchSize`. These respectively define the log prior for the model; and the minibatch size, as it was defined in Section 2. By default, the `logPrior` is set to an uninformative uniform prior, which is fine to

[3]For full API details see `https://www.tensorflow.org/api_docs/python/tf/contrib/distributions` though note this is for Python, so the `.` object notation needs to be replaced by `$`, for example `tf.contrib.distributions.Normal` would be replaced by `tf$contrib$distributions$Normal`.

[4]See `https://www.tensorflow.org/api_guides/python/math_ops`.

use for quick checks but will need to be set properly for more complex models. The `logPrior` is a function similar to `logLik`, but only takes `params` as input. The `minibatchSize` is a numeric, and can either be a proportion of the dataset size, if it is set between 0 and 1, or an integer. The default value is 0.01, meaning that 1% of the full dataset is used at each iteration.

The control variate functions have the same inputs as the non-control variate functions, with one more required input. The `optStepsize` input is a numeric that specifies the stepsize for the initial optimisation step to find the $\hat{\theta}$ maximising the posterior as defined in Section 2.4. For a full outline of the key inputs, see Table 2.

Often large datasets and high dimensional problems go hand in hand. In these high dimensional settings storing the full MCMC chain in memory can become an issue. For this situation we provide functionality to run the chain one iteration at a time in a user defined loop. This enables the user to deal with the output at each iteration how they see fit. For example, they may wish to calculate a test function on the output to reduce the dimensionality of the chain; or they might calculate the required Monte Carlo estimates on the fly. We aim to extend this functionality to enable the user to define their own Gibbs updates alongside the SGMCMC procedure. This functionality is more involved, and requires more knowledge of **TensorFlow**, so we leave the details to the example in Section 4.2.

For the rest of this section we go into more detail about the usage of the functions using a worked example. The package is used to infer the bias and coefficient parameters in a logistic regression model. Section 4.1 demonstrates standard usage by performing inference on the model using the `sgld` and `sgldcv` functions. Section 4.2 demonstrates usage in problems where the full MCMC chain cannot be fit into memory. The same logistic regression model is used throughout.

4.1 Example usage

In this example we use the functions `sgld` and `sgldcv` to infer the bias (or intercept) and coefficients of a logistic regression model. The `sgldcv` case is also available as a vignette. Suppose we have data $\mathbf{x}_1, \dots, \mathbf{x}_N$ of dimension d taking values in $\mathbb{R}^d$; and response variables $y_1, \dots, y_N$ taking values in $\{0, 1\}$. Then a logistic regression model with coefficients β and bias β_0 will have likelihood

$$p(\mathbf{X}, \mathbf{y}|\beta, \beta_0) = \prod_{i=1}^{N} \left[\frac{1}{1 + e^{-\beta_0 - \mathbf{x}_i\beta}}\right]^{y_i} \left[1 - \frac{1}{1 + e^{-\beta_0 - \mathbf{x}_i\beta}}\right]^{1-y_i} \tag{9}$$

We will use the `covertype` dataset (Blackard and Dean, 1999) which can be downloaded and loaded using the **sgmcmc** function `getDataset`, which downloads example datasets for the package. The `covertype` dataset uses mapping data to predict the type of forest cover. Our particular dataset is taken from the LIBSVM library (Chang and Lin, 2011), which converts the data to a binary problem, rather than multiclass. The dataset consists of a matrix of dimension 581012×55. The first column contains the labels $\mathbf{y}$, taking values in $\{0, 1\}$. The remaining columns are the explanatory variables $\mathbf{X}$, which have been scaled to take values in $[0, 1]$.

```
library("sgmcmc")
covertype = getDataset("covertype")
# Split the data into predictors and response
X = covertype[,2:ncol(covertype)]
y = covertype[,1]
# Create dataset list for input
dataset = list( "X" = X, "y" = y )
```

In the last line we defined the dataset as a list object which will be input to the relevant **sgmcmc** function. The list names can be arbitrary, but must be consistent with the variables declared in the `logLik` function (see below).

When accessing the data, it is assumed that observations are split along the first axis. In other words, `dataset$X` is a 2-dimensional matrix, and we assume that observation $\mathbf{x}_i$ is accessed at `dataset$X[i,]`. Similarly, suppose `Z` was a 3-dimensional array, we would assume that observation i would be accessed at `Z[i,,]`. Parameters are declared in a similar way, except the list entries are the desired parameter starting points. There are two parameters for this model, the bias β_0 and the coefficients β, which can be arbitrarily initialised to 0.

```
# Get the dimension of X, needed to set shape of params
d = ncol(dataset$X)
params = list( "bias" = 0, "beta" = matrix( rep( 0, d ), nrow = d ) )
```

The log likelihood is specified as a function of the `dataset` and `params`, which are lists with the same names we declared earlier. The only difference is that the objects inside the lists will have automatically been converted to **TensorFlow** objects. The `dataset` list will contain **TensorFlow** placeholders. The `params` list will contain **TensorFlow** variables. The `logLik` function should be a function that takes these lists as input and returns the log likelihood value given the current parameters and data. This is done using **TensorFlow** operations, as this allows the gradient to be automatically calculated.

For the logistic regression model (9), the log likelihood is

$$\log p(\mathbf{X}, \mathbf{y}|\beta, \beta_0) = \sum_{i=1}^{N} y_i \log y_{\text{est}} + (1 - y_i) \log(1 - y_{\text{est}}),$$

where $y_{\text{est}} = [1 + e^{-\beta_0 - \mathbf{x}_i \beta}]^{-1}$, which coded as a `logLik` function is defined as follows

```
logLik = function(params, dataset) {
    yEst = 1 / (1 + tf$exp( - tf$squeeze(params$bias +
            tf$matmul(dataset$X, params$beta))))
    logLik = tf$reduce_sum(dataset$y * tf$log(yEst) +
            (1 - dataset$y) * tf$log(1 - yEst))
    return(logLik)
}
```

For more information about the usage of these **TensorFlow** functions, please see the **TensorFlow** documentation.[5]

Next, the log prior density, where we assume each β_j, for $j = 0, \ldots, d$, has an independent Laplace prior with location 0 and scale 1, so $\log p(\beta) \propto -\sum_{j=0}^{d} |\beta_j|$. Similar to `logLik`, this is defined as a function, but with only `params` as input

```
logPrior = function(params) {
    logPrior = - (tf$reduce_sum(tf$abs(params$beta)) +
            tf$reduce_sum(tf$abs(params$bias)))
    return(logPrior)
}
```

The final input that needs to be set is the `stepsize` for tuning the methods, this can be set as a list

```
stepsize = list("beta" = 2e-5, "bias" = 2e-5)
```

Setting the same stepsize for all parameters is done as `stepsize = 2e-5`. This shorthand can also be used for any of the optional tuning parameters which need to specified as lists. The stepsize parameter

[5] Documentation for **TensorFlow** for R available at `https://tensorflow.rstudio.com/tensorflow/`

will generally require a bit of tuning in order to get good performance, for this we recommend using cross validation (see e.g., Friedman et al., 2001, Chapter 7).

Everything is now ready to run a standard SGLD algorithm, with `minibatchSize` set to 500. To keep things reproducible we'll set the seed to 13.

```
output = sgld( logLik, dataset, params, stepsize,
       logPrior = logPrior, minibatchSize = 500, nIters = 10000, seed = 13 )
```

The output of the function is also a list with the same names as the `params` list. Suppose a given parameter has shape $(d_1, \ldots, d_l)$, then the output will be an array of shape $(\text{nIters}, d_1, \ldots, d_l)$. So in our case, `output$beta[i,,]` is the i^{th} MCMC sample from the parameter β; and `dim(output$beta)` is `c(10000, 54, 1)`.

In order to run a control variate algorithm such as `sgldcv` we need one additional argument, which is the stepsize for the initial optimisation step. The optimisation uses the **TensorFlow** `GradientDescentOptimizer`. The stepsize should be quite similar to the stepsize for SGLD, though is often slightly larger. First, so that we can measure the performance of the chain, we shall set a seed and randomly remove some observations from the full `dataset` to form a `testset`. We also set a short burn-in of 1000.

```
set.seed(13)
testSample = sample(nrow(dataset$X), 10^4)
testset = list( "X" = dataset$X[testSample,], "y" = dataset$y[testSample] )
dataset = list( "X" = dataset$X[-testSample,], "y" = dataset$y[-testSample] )
output = sgldcv( logLik, dataset, params, 5e-6, 5e-6,
       logPrior = logPrior, minibatchSize = 500, nIters = 11000, seed = 13 )
# Remove burn-in
output$beta = output$beta[-c(1:1000),,]
output$bias = output$bias[-c(1:1000)]
```

A common performance measure for a classifier is the *log loss*. Given an observation with data $\mathbf{x}_i$ and response y_i, logistic regression predicts that $y_i = 1$ with probability

$$\pi(\mathbf{x}_i, \beta, \beta_0) = \frac{1}{1 + e^{-\beta_0 - \mathbf{x}_i \beta}}.$$

Given a test set T of data response pairs $(\mathbf{x}, y)$, the log loss $s(\cdot)$, of a binary chain, is defined as

$$s(\beta, \beta_0, T) = -\frac{1}{|T|} \sum_{(\mathbf{x},y) \in T} \left[y \log \pi(\mathbf{x}, \beta, \beta_0) + (1 - y) \log(1 - \pi(\mathbf{x}, \beta, \beta_0)) \right]. \tag{10}$$

To check convergence of `sgldcv` we'll plot the log loss every 10 iterations, using the `testset` we removed earlier. Results are given in Figure 1. The plot shows the `sgldcv` algorithm converging to a stationary after a short burn-in period. The burn-in period is so short due to the initial optimisation step.

```
iterations = seq(from = 1, to = 10^4, by = 10)
logLoss = rep(0, length(iterations))
# Calculate log loss every 10 iterations
for ( iter in 1:length(iterations) ) {
    j = iterations[iter]
    beta0_j = output$bias[j]
    beta_j = output$beta[j,]
    for ( i in 1:length(testset$y) ) {
        piCurr = 1 / (1 + exp(- beta0_j - sum(testset$X[i,] * beta_j)))
```

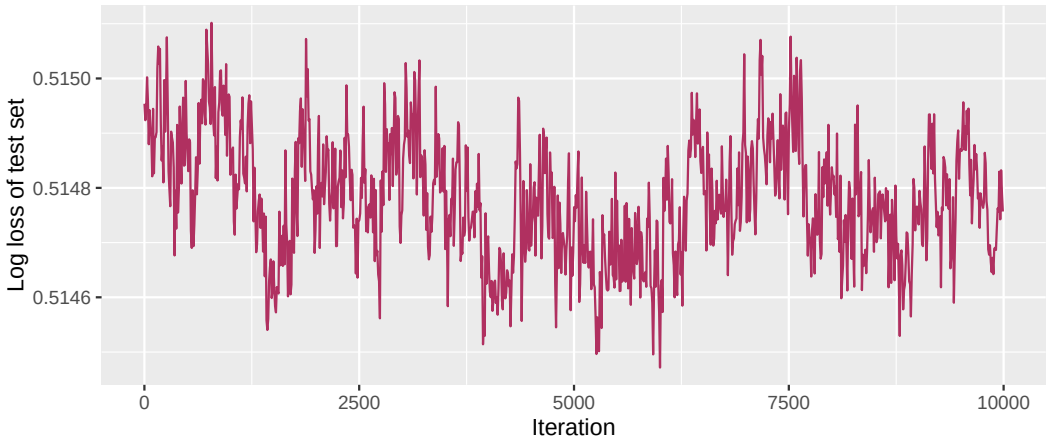

Figure 1: Log loss on a test set for parameters simulated using the `sgldcv` algorithm after 1000 iterations of burn-in. Logistic regression problem with the covertype dataset.

```
        y_i = testset$y[i]
        logLossCurr = - ( (y_i * log(piCurr) + (1 - y_i) * log(1 - piCurr)) )
        logLoss[iter] = logLoss[iter] + 1 / length(testset$y) * logLossCurr
    }
}
# Plot output
plotFrame = data.frame("Iteration" = iterations, "logLoss" = logLoss)
ggplot(plotFrame, aes(x = Iteration, y = logLoss)) +
    geom_line(color = "maroon") +
    ylab("Log loss of test set")
```

4.2 Example usage: Storage constraints

Often large datasets and high dimensionality go hand in hand. Sometimes the dimensionality is so high that storage of the full MCMC chain in memory becomes an issue. There are a number of ways around this, including: calculating estimates of the desired posterior quantity on the fly; reducing the dimensionality of the chain using a test function; or just periodically saving a the chain to the hard disk and starting from scratch. To deal with high storage costs **sgmcmc** provides functionality to run SGMCMC algorithms step by step. This allows users to deal with the output as they see fit at each iteration.

In this section, we detail how to run SGMCMC chains step by step. To do this requires more knowledge of **TensorFlow**, including using **TensorFlow** sessions and building custom placeholders and tensors. For more details see the **TensorFlow** for R documentation (Allaire et al., 2016). The step by step procedure works similarly to a standard **TensorFlow** procedure: **TensorFlow** variables, tensors and placeholders are declared; then the **TensorFlow** session is started and all the required tensors are initialised; finally the SGMCMC algorithm is run step by step in a user defined loop, and the user evaluates tensors as required.

To demonstrate this concept we keep things simple and use the logistic regression example introduced in the previous section. While this example can fit into memory, it allows us to demonstrate the concepts without getting bogged down in a complicated model. For a more realistic example, where the output does not fit into memory, see the Bayesian neural network model in Section 5.3.

We start by assuming the `dataset`, `params`, `logLik`, `logPrior` and `stepsize` objects have been created in exactly the same way as in the previous example (Section 4.1). Now suppose we want to make inference

using stochastic gradient Langevin dynamics (SGLD), but we want to run it step by step. The first step is to initialise an `sgld` object using the function `sgldSetup`. For every function in Table 1 there is a corresponding `*Setup` function, such as `sghmccvSetup` or `sgnhtSetup`. This function will create all the **TensorFlow** objects required, as well as declare the dynamics of the SGMCMC algorithm. For our example we can run the following

```
sgld = sgldSetup(logLik, dataset, params, stepsize,
        logPrior = logPrior, minibatchSize = 500, seed = 13)
```

This `sgld` object is a type of `sgmcmc` object, it is an R `S3` object, which is essentially a list with a number of entries. The most important of these entries for building SGMCMC algorithms is called `params`, which holds a list, with the same names as in the `params` that were fed to `sgldSetup`, but this list contains `tf$Variable` objects. This is how the tensors are accessed which hold the current parameter values in the chain. For more details on the attributes of these objects, see the documentation for `sgldSetup`, `sgldcvSetup`, etc.

Now that we have created the `sgld` object, we want to initialise the **TensorFlow** variables and the `sgmcmc` algorithm chosen. For a standard algorithm, this will initialise the **TensorFlow** graph and all the tensors that were created. For an algorithm with control variates (e.g., `sgldcv`), this will also find the $\hat{\theta}$ estimates of the parameters and calculate the full log posterior gradient at that point; as detailed in Section 2.4. The function used to do this is `initSess`,

```
sess = initSess(sgld)
```

The `sess` returned by `initSess` is the current **TensorFlow** session, which is needed to run the SGMCMC algorithm of choice, and to access any of the tensors needed, such as `sgld$params`.

Now we have everything to run an SGLD algorithm step by step as follows

```
for (i in 1:10^4) {
    sgmcmcStep(sgld, sess)
    currentState = getParams(sgld, sess)
}
```

Here the function `sgmcmcStep` will update `sgld$params` using a single update of SGLD, or whichever SGMCMC algorithm is given. The function `getParams` will return a list of the current parameters as R objects rather than as tensors.

This simple example of running SGLD step by step only stores the most recent value in the chain, which is useless for a Monte Carlo method. Also, for large scale examples, it is often useful to reduce the dimension of the chain by calculating some test function $g(\cdot)$ of θ at each iteration rather than the parameters themselves. This example will demonstrate how to store a test function at each iteration, as well as calculating the estimated posterior mean on the fly. We assume that a new R session has been started and the `sgld` object has just been created using `sgldSetup` with the same properties as in the example above. We assume that no **TensorFlow** session has been created (i.e., `initSess` has not been run yet).

Before the **TensorFlow** session has been declared, the user is able to create their own custom tensors. This is useful, as test functions can be declared beforehand using the `sgld$params` variables, which allows the test functions to be quickly calculated by just evaluating the tensors in the current session. The test function used here is once again the log loss of a test set, as defined in (10).

Suppose we input `sgld$params` and the `testset` T to the `logLik` function. Then the log loss is actually $-\frac{1}{|T|}$ times this value. This means we can easily create a tensor that calculates the log loss by creating a list of placeholders that hold the test set, then using the `logLik` function with the `testset` list and `sgld$params` as input. We can do this as follows

```
testPlaceholder = list()
testPlaceholder[["X"]] = tf$placeholder(tf$float32, dim(testset[["X"]]))
testPlaceholder[["y"]] = tf$placeholder(tf$float32, dim(testset[["y"]]))
testSize = as.double(nrow(testset[["X"]]))
logLoss = - logLik(sgld$params, testPlaceholder) / testSize
```

This placeholder is then fed the full `testset` each time the log loss is calculated. Now we will declare the **TensorFlow** session, and run the chain step by step. At each iteration we will calculate the current Monte Carlo estimate of the parameters. The log loss will be stored every 100 iterations. We omit a plot of the log loss as it is similar to Figure 1.

```
sess = initSess(sgld)
# Fill a feed dict with full test set (used to calculate log loss)
feedDict = dict()
feedDict[[testPlaceholder[["X"]]]] = testset[["X"]]
feedDict[[testPlaceholder[["y"]]]] = testset[["y"]]
# Burn-in chain
message("Burning-in chain...")
message("iteration\tlog loss")
for (i in 1:10^4) {
# Print progress
    if (i %% 100 == 0) {
        progress = sess$run(logLoss, feed_dict = feedDict)
        message(paste0(i, "\t", progress))
    }
    sgmcmcStep(sgld, sess)
}
# Initialise posterior mean estimate using value after burn-in
postMean = getParams(sgld, sess)
logLossOut = rep(0, 10^4 / 100)
# Run chain
message("Running SGMCMC...")
for (i in 1:10^4) {
    sgmcmcStep(sgld, sess)
    # Update posterior mean estimate
    currentState = getParams(sgld, sess)
    for (paramName in names(postMean)) {
        postMean[[paramName]] = (postMean[[paramName]] * i +
                currentState[[paramName]]) / (i + 1)
    }
    # Print and store log loss
    if (i %% 100 == 0) {
        logLossOut[i/100] = sess$run(logLoss, feed_dict = feedDict)
        message(paste0(i, "\t", logLossOut[i/100]))
    }
}
```

5 Simulations

In this section we demonstrate the algorithms and performance by simulating from a variety of models using all the implemented methods and commenting on the performance of each. These simulations are reproducible and available in the supplementary material and on Github.[6] For more usage tutorials similar to Sections 4.1 and 4.2, please see the vignettes on the package website.[7]

5.1 Gaussian mixture

In this model we assume our dataset $x_1, \ldots, x_N$ is drawn i.i.d from

$$X_i \,|\, \theta_1, \theta_2 \sim \frac{1}{2}\mathcal{N}(\theta_1, \mathrm{I}_2) + \frac{1}{2}\mathcal{N}(\theta_2, \mathrm{I}_2), \quad i = 1, \ldots, N; \tag{11}$$

where θ_1, θ_2 are parameters to be inferred and I_2 is the 2×2 identity matrix. We assume the prior $\theta_1, \theta_2 \sim \mathcal{N}(0, 10\mathrm{I}_2)$. To generate the synthetic dataset, we simulate 10^3 observations from $\frac{1}{2}\mathcal{N}\left([0,0]^\top, \mathrm{I}_2\right) + \frac{1}{2}\mathcal{N}\left([0.1, 0.1]^\top, \mathrm{I}_2\right)$. While this is a small number of observations, it allows us to compare the results to a full Hamiltonian Monte Carlo (HMC) scheme using the R implementation of **STAN** (Carpenter et al., 2017). The full HMC scheme should sample from close to the true posterior distribution, so acts as a good surrogate for the truth. We compare each **sgmcmc** algorithm implemented to the HMC sample to compare performance. Larger scale examples are given in Sections 5.2 and 5.3. We ran all methods for 10^4 iterations, except SGHMC, since the computational cost is greater for this method due to the trajectory parameter L. We ran SGHMC for 2,000 iterations, using default trajectory $L = 5$, as this ensures the overall computational cost of the method is similar to the other methods. We used a burn-in step of 10^4 iterations, except for the control variate methods, where we used 10^4 iterations in the initial optimisation step, with no burn-in. Again this ensures comparable computational cost across different methods.

The `logLik` and `logPrior` functions used for this model are as follows

```
logLik = function( params, dataset ) {
    # Declare Sigma (assumed known)
    SigmaDiag = c(1, 1)
    # Declare distribution of each component
    component1 = tf$contrib$distributions$MultivariateNormalDiag(
            params$theta1, SigmaDiag )
    component2 = tf$contrib$distributions$MultivariateNormalDiag(
            params$theta2, SigmaDiag )
    # Declare allocation probabilities of each component
    probs = tf$contrib$distributions$Categorical(c(0.5,0.5))
    # Declare full mixture distribution given components and probs
    distn = tf$contrib$distributions$Mixture(
            probs, list(component1, component2))
    # Declare log likelihood
    logLik = tf$reduce_sum( distn$log_prob(dataset$X) )
    return( logLik )
}

logPrior = function( params ) {
    # Declare hyperparameters mu0 and Sigma0
    mu0 = c( 0, 0 )
    Sigma0Diag = c(10, 10)
```

[6] https://github.com/jbaker92/sgmcmc-simulations
[7] https://stor-i.github.io/sgmcmc

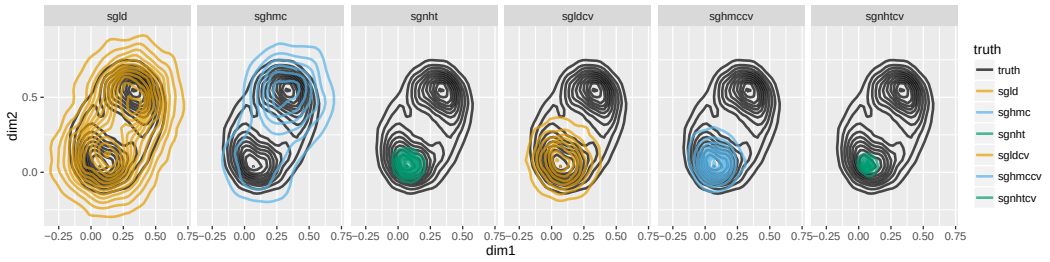

Figure 2: Plots of the approximate posterior for θ_1 simulated using each of the methods implemented by **sgmcmc**, compared with a full HMC run, treated as the truth, for the Gaussian mixture model (11).

```
    # Declare prior distribution
    priorDistn = tf$contrib$distributions$MultivariateNormalDiag(
            mu0, Sigma0Diag )
    # Declare log prior density and return
    logPrior = priorDistn$log_prob( params$theta1 ) +
            priorDistn$log_prob( params$theta2 )
    return( logPrior )
}
```

The following list determines the stepsizes used for each method, the `optStepsize` used for control variate methods was `5e-5`.

```
stepsizeList = list("sgld" = 5e-3, "sghmc" = 5e-4, "sgnht" = 3e-4,
        "sgldcv" = 1e-2, "sghmccv" = 1.5e-3, "sgnhtcv" = 3e-3)
```

We set the seed to be 2 using the optional `seed` argument and use a minibatch size of 100. We also used a seed of 2 when generating the data (see the supplementary material for full details). Starting points were sampled from a standard Normal.

The results are plotted in Figure 2. The black contours represent the best guess at the true posterior, which was found using the standard HMC procedure in STAN. The coloured contours that overlay the black contours are the approximations of each of the SGMCMC methods implemented by **sgmcmc**. This allows us to compare the SGMCMC estimates with the 'truth' by eye.

In the simulation, we obtain two chains, one approximating θ_1 and the other approximating θ_2. In order to examine how well the methods explore both modes, we take just θ_1 and compare this to the HMC run for θ_1. The results are quite variable, and it demonstrates a point nicely: there seems to be a trade-off between predictive accuracy and exploration. Many methods have demonstrated good performance using predictive accuracy; where a test set is removed from the full dataset to assess how well the fitted model performs on the test set. This is a useful technique for complex models, which are high dimensional and have a large number of data points, as they cannot be plotted, and an MCMC run to act as the 'truth' cannot be fitted.

However, this example shows that it does not give the full picture. A lot of the methods which show improved predictive performance (e.g., control variate methods and especially `sgnht`) over `sgld` appear here to perform worse at exploring the full space. In this example, `sgld` performs the best at exploring both components, though it over-estimates posterior uncertainty. The algorithm `sghmc` also explores both components but somewhat unevenly. We find that `sgnht`, while being shown to have better predictive performance in the original paper (Ding et al., 2014), does not do nearly as well as the other algorithms at exploring the space and appears to collapse to the posterior mode. The control variate methods, shown in the following sections, and in Baker et al. (2017), appear to have better predictive performance than `sgld`, but do not explore both components either. For example, `sgldcv` explores the space the best but over-estimates

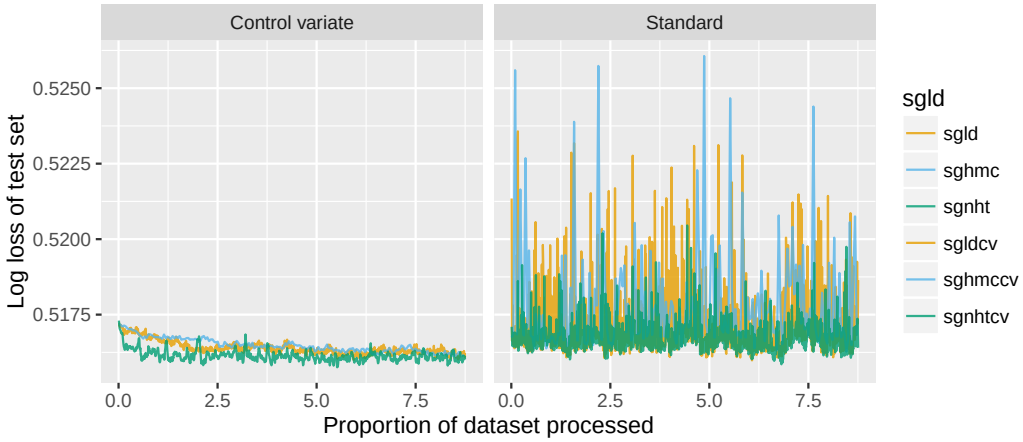

Figure 3: Plots of the log loss of a test set for β_0 and β simulated using each of the methods implemented by **sgmcmc**. Logistic regression problem with the covertype dataset.

uncertainty of the first component, since it relies on SGLD updates which also overestimates uncertainty. In contrast, `sgnhtcv` collapses to a posterior mode since it relies on the SGNHT updates which also collapse.

5.2 Bayesian logistic regression

In this section, we apply all the methods to the logistic regression example in Section 4.1. We compare the performance of the methods by calculating the log loss of a test set every 10 iterations, again as detailed in Section 4.1. The standard methods (`sgld`, `sghmc`, `sgnht`) were run for 10^4 iterations with an additional 10^4 iterations of burn-in; except for `sghmc` which has $5\times$ the computational cost, so is ran for 2,000 iterations with 2,000 iterations of burn-in. The control variate methods were run for 10^4 iterations with an additional 10^4 iterations for the initial optimisation step, and no burn-in; again except for `sghmccv` which was run for 2,000 iterations. This means that all the methods should be somewhat comparable in terms of computation time.

The following list determines the stepsizes used for each method, the `optStepsize` used was `1e-6`.

```
stepsizes = list("sgld" = 5e-6, "sghmc" = 1e-7, "sgnht" = 1e-7, "sgldcv" = 1e-5,
        "sghmccv" = 1e-6, "sgnhtcv" = 5e-7)
```

We set the seed to be 1 for each of the simulations, and when generating the test data (see the supplementary material for reproducible code) and use a minibatch size of 500. Starting points are sampled from a standard Normal.

Results are plotted in Figure 3. All of the algorithms show decent performance. Methods which use control variates have significantly better predictive performance; and result in chains with lower variance. `sghmc` has lower variance than `sgld` and `sgnht`, though this could be related to the high computational cost. One might envisage setting a lower trajectory L would result in a chain with higher variance. `sgldcv` takes longer to burn-in than the other control variate methods. The algorithm `sgld` has the highest variance by far; this could be related to our discussion in Section 5.1 on exploration versus accuracy.

5.3 Bayesian neural network

In this simulation we demonstrate a very high dimensional chain. This gives a more realistic example of when we would want to run the chain step by step. The model is a two layer Bayesian neural network which is fit to the MNIST dataset (LeCun and Cortes, 2010). The MNIST dataset consists of 28×28 pixel images

of handwritten digits from zero to nine. The images are flattened to be a vector of length 784. The dataset is available as a standard dataset from the **TensorFlow** library, with a matrix of 55,000 training vectors and 10,000 test vectors, each with their corresponding labels. The dataset can be constructed in a similar way to the logistic regression example of Section 4.1, using the standard dataset in the package `mnist`.

```
library("sgmcmc")
mnist = getDataset("mnist")
dataset = list("X" = mnist$train$images, "y" = mnist$train$labels)
testset = list("X" = mnist$test$images, "y" = mnist$test$labels)
```

We build the same neural network model as in the original SGHMC paper by Chen et al. (2014). Suppose Y_i takes values in $\{0, \dots, 9\}$, so is the output label of a digit, and $\mathbf{x}_i$ is the input vector, with $\mathbf{X}$ the full $N \times 784$ dataset, where N is the number of observations. The model is then as follows

$$Y_i \,|\, \theta, \mathbf{x}_i \sim \text{Categorical}(\beta(\theta, \mathbf{x}_i)), \tag{12}$$

$$\beta(\theta, \mathbf{x}_i) = \sigma\left(\sigma\left(\mathbf{x}_i^\top B + b\right) A + a\right). \tag{13}$$

Here A, B, a, b are parameters to be inferred with $\theta = (A, B, a, b)$; $\sigma(\cdot)$ is the softmax function (a generalisation of the logistic link function). A, B, a and b are matrices with dimensions: 100×10, 784×100, 1×10 and 1×100 respectively. Each element of these parameters is assigned a Normal prior

$$A_{kl}|\lambda_A \sim \mathcal{N}(0, \lambda_A^{-1}), \quad B_{jk}|\lambda_B \sim \mathcal{N}(0, \lambda_B^{-1}),$$
$$a_l|\lambda_a \sim \mathcal{N}(0, \lambda_a^{-1}), \quad b_k|\lambda_b \sim \mathcal{N}(0, \lambda_b^{-1}),$$
$$j = 1, \dots, 784; \quad k = 1, \dots, 100; \quad l = 1, \dots, 10;$$

where λ_A, λ_B, λ_a and λ_b are hyperparameters. Finally, we assume

$$\lambda_A, \lambda_B, \lambda_a, \lambda_b \sim \text{Gamma}(1, 1).$$

The model contains a large number of high dimensional parameters, and unless there is sufficient RAM available, a standard chain of length 10^4 will not fit into memory. First, we shall create the `params` dictionary, and then code the `logLik` and `logPrior` functions. We can sample the initial λ parameters from a standard Gamma distribution, and the remaining parameters from a standard Normal as follows

```
# Sample initial weights from standard Normal
d = ncol(dataset$X)
params = list()
params$A = matrix(rnorm(10*100), ncol = 10)
params$B = matrix(rnorm(d*100), ncol = 100)
# Sample initial bias parameters from standard Normal
params$a = rnorm(10)
params$b = rnorm(100)
# Sample initial precision parameters from standard Gamma
params$lambdaA = rgamma(1, 1)
params$lambdaB = rgamma(1, 1)
params$lambdaa = rgamma(1, 1)
params$lambdab = rgamma(1, 1)
```

```
logLik = function(params, dataset) {
    # Calculate estimated probabilities
    beta = tf$nn$softmax(tf$matmul(dataset$X, params$B) + params$b)
```

```
    beta = tf$nn$softmax(tf$matmul(beta, params$A) + params$a)
    # Calculate log likelihood of categorical distribution with prob. beta
    logLik = tf$reduce_sum(dataset$y * tf$log(beta))
    return(logLik)
}
```

```
logPrior = function(params) {
    distLambda = tf$contrib$distributions$Gamma(1, 1)
    distA = tf$contrib$distributions$Normal(0, tf$rsqrt(params$lambdaA))
    logPriorA = tf$reduce_sum(distA$log_prob(params$A)) +
           distLambda$log_prob(params$lambdaA)
    distB = tf$contrib$distributions$Normal(0, tf$rsqrt(params$lambdaB))
    logPriorB = tf$reduce_sum(distB$log_prob(params$B)) +
           distLambda$log_prob(params$lambdaB)
    dista = tf$contrib$distributions$Normal(0, tf$rsqrt(params$lambdaa))
    logPriora = tf$reduce_sum(dista$log_prob(params$a)) +
           distLambda$log_prob(params$lambdaa)
    distb = tf$contrib$distributions$Normal(0, tf$rsqrt(params$lambdab))
    logPriorb = tf$reduce_sum(distb$log_prob(params$b)) +
           distLambda$log_prob(params$lambdab)
    logPrior = logPriorA + logPriorB + logPriora + logPriorb
    return(logPrior)
}
```

Similar to Section 4.2, we use the log loss as a test function. This time though it is necessary to update the definition, as the logistic regression example was a binary problem whereas now we have a multiclass problem. Given a test set T of pairs $(\mathbf{x}, y)$, now y can take values in $\{0, \ldots, K\}$, rather than just binary values. To account for this we redefine the definition of log loss to be

$$s(\theta, T) = -\frac{1}{|T|} \sum_{\mathbf{x},y \in T} \sum_{k=1}^{K} \mathbf{1}_{y=k} \log \beta_k(\theta, \mathbf{x}),$$

where $\mathbf{1}_A$ is the indicator function, and $\beta_k(\theta, \mathbf{x})$ is the k^{th} element of $\beta(\theta, \mathbf{x})$ as defined in (13).

As in Section 4.2, the log loss is simply $-\frac{1}{|T|}$ times the `logLik` function, if we feed it the `testset` rather than the `dataset`. This means the `logLoss` tensor can be declared in a similar way to Section 4.2

```
testPlaceholder = list()
testPlaceholder[["X"]] = tf$placeholder(tf$float32, dim(testset[["X"]]))
testPlaceholder[["y"]] = tf$placeholder(tf$float32, dim(testset[["y"]]))
testSize = as.double(nrow(testset[["X"]]))
logLoss = - logLik(sgld$params, testPlaceholder) / testSize
```

We can run the chain in exactly the same way as Section 4.2, and so omit the code for this. We ran 10^4 iterations of each of the algorithms in Table 1, calculating the log loss for each every 10 iterations. The standard algorithms have 10^4 iterations of burn-in while the control variate algorithms have no burn-in, but 10^4 iterations in the initial optimisation step. Note that due to the trajectory parameter L, `sghmc` and `sghmccv` will have 5 times the computational cost of the other algorithms. Therefore, we ran these algorithms for 2,000 iterations instead, to make the computational cost comparable. We used the following list of stepsizes

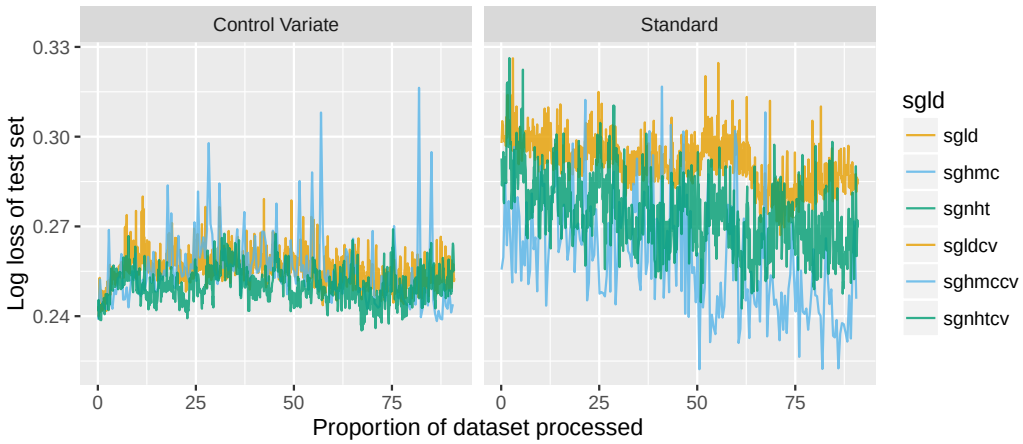

Figure 4: Plots of the log loss of a test set for β_0 and β simulated using each of the methods implemented by **sgmcmc**. Logistic regression problem with the covertype dataset.

```
list("sgld" = 1e-4, "sghmc" = 1e-5, "sgnht" = 5e-6, "sgldcv" = 5e-5,
        "sghmccv" = 1e-5, "sgnhtcv" = 5e-7)
```

Generally these are the stepsizes which produce the smallest log loss; except when these chains did not seem to explore the space fully, in which case we increased the stepsize slightly. We set the seed to be 1 for each of the simulations, and when generating the test data (see the supplementary material for reproducible code).

The results are plotted in Figure 4. Again we see improvements in the predictive performance of the control variate methods. Among the standard methods, `sghmc` and `sgnht` have the best predictive performance; which is to be expected given the apparent trade-off between accuracy and exploration.

6 Discussion

We presented the R package **sgmcmc**, which enables Bayesian inference with large datasets using stochastic gradient Markov chain Monte Carlo. The package only requires the user to specify the log likelihood and log prior functions; and any differentiation required can be performed automatically. The package is based on **TensorFlow**, an efficient library for numerical computation that can take advantage of many different architectures, including GPUs. The **sgmcmc** package keeps much of this efficiency. The package provides functionality to deal with cases where the full MCMC chain is too large to fit into memory. As the chain can be run step by step at each iteration, there is flexibility for these cases.

We implemented the methods on a variety of statistical models, many on realistic datasets. One of these statistical models was a neural network, for which the full MCMC chain would not fit into memory. In this case we demonstrated building test functions and calculating the Monte Carlo estimates on the fly. We empirically demonstrated the predictive performance of the algorithms and the trade-off that appears to occur between predictive performance and exploration.

Many complex models for which SGMCMC methods have been found to perform well require Gibbs updates to be performed periodically (Patterson and Teh, 2013; Li et al., 2016). In the future we would like to build functionality for user defined Gibbs steps that can be updated step by step alongside the **sgmcmc** algorithms. SGHMC has been implemented by setting the value $\hat{\beta}_t = 0$, as in the experiments of the original paper Chen et al. (2014). In the future, we would like to implement a more sophisticated approach to set this value, such as using a similar estimate to Ahn et al. (2012).

7 Acknowledgements

The first author gratefully acknowledges the support of the EPSRC funded EP/L015692/1 STOR-i Centre for Doctoral Training. This work was supported by EPSRC grant EP/K014463/1, ONR Grant N00014-15-1-2380 and NSF CAREER Award IIS-1350133.

References

Abraham, R., Marsden, J. E., and Ratiu, R. (1988). *Manifolds, Tensor Analysis, and Applications*, volume 2. Springer-Verlag.

Ahn, S., Korattikara, A., and Welling, M. (2012). Bayesian posterior sampling via stochastic gradient Fisher scoring. In *Proceedings of the 29th International Conference on Machine Learning*, pages 1591–1598. PMLR.

Allaire, J., Eddelbuettel, D., Golding, N., and Tang, Y. (2016). **TensorFlow***: R Interface to TensorFlow*.

Baker, J., Fearnhead, P., Fox, E. B., and Nemeth, C. (2017). Control variates for stochastic gradient MCMC.

Blackard, J. A. and Dean, D. J. (1999). Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and Electronics in Agriculture*, 24(3):131–151.

Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. (2017). **Stan**: A probabilistic programming language. *Journal of Statistical Software*, 76(1).

Chang, C.-C. and Lin, C.-J. (2011). **LIBSVM**: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):1–27.

Chen, T., Fox, E. B., and Guestrin, C. (2014). Stochastic gradient Hamiltonian Monte Carlo. In *Proceedings of the 31st International Conference on Machine Learning*, pages 1683–1691. PMLR.

Ding, N., Fang, Y., Babbush, R., Chen, C., Skeel, R. D., and Neven, H. (2014). Bayesian sampling using stochastic gradient thermostats. In *Advances in Neural Information Processing Systems 27*, pages 3203–3211. Curran Associates, Inc.

Dubey, K. A., Reddi, S. J., Williamson, S. A., Poczos, B., Smola, A. J., and Xing, E. P. (2016). Variance reduction in stochastic gradient Langevin dynamics. In *Advances in Neural Information Processing Systems 29*, pages 1154–1162. Curran Associates, Inc.

Friedman, J., Hastie, T., and Tibshirani, R. (2001). *The Elements of Statistical Learning*, volume 1. Springer-Verlag.

Griewank, A. and Walther, A. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, volume 2. SIAM.

LeCun, Y. and Cortes, C. (2010). *MNIST Handwritten Digit Database*.

Li, W., Ahn, S., and Welling, M. (2016). Scalable MCMC for mixed membership stochastic blockmodels. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, pages 723–731. PMLR.

Nagapetyan, T., Duncan, A., Hasenclever, L., Vollmer, S. J., Szpruch, L., and Zygalakis, K. (2017). The true cost of stochastic gradient Langevin dynamics.

Neal, R. M. (2010). MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*, volume 1. Chapman & Hall.

Patterson, S. and Teh, Y. W. (2013). Stochastic gradient Riemannian Langevin dynamics on the probability simplex. In *Advances in Neural Information Processing Systems 26*, pages 3102–3110. Curran Associates, Inc.

TensorFlow Development Team (2015). **TensorFlow***: Large-Scale Machine Learning on Heterogeneous Systems*.

R Development Core Team (2008). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing.

Ripley, B. D. (2009). *Stochastic Simulation*, volume 1. John Wiley & Sons.

Roberts, G. O. and Rosenthal, J. S. (1998). Optimal scaling of discrete approximations to Langevin diffusions. *Journal of the Royal Statistical Society B*, 60(1):255–268.

Sato, I. and Nakagawa, H. (2014). Approximation analysis of stochastic gradient Langevin dynamics by using Fokker-Planck equation and Ito process. In *Proceedings of the 31st International Conference on Machine Learning*, pages 982–990. PMLR.

Teh, Y. W., Thiéry, A. H., and Vollmer, S. J. (2016). Consistency and fluctuations for stochastic gradient Langevin dynamics. *Journal of Machine Learning Research*, 17(7):1–33.

Tran, D., Kucukelbir, A., Dieng, A. B., Rudolph, M., Liang, D., and Blei, D. M. (2016). **Edward**: A Library for Probabilistic Modeling, Inference, and Criticism.

Vollmer, S. J., Zygalakis, K. C., and Teh, Y. W. (2016). Exploration of the (non-)asymptotic bias and variance of stochastic gradient Langevin dynamics. *Journal of Machine Learning Research*, 17(159):1–48.

Welling, M. and Teh, Y. W. (2011). Bayesian learning via stochastic gradient Langevin dynamics. In *Proceedings of the 28th International Conference on Machine Learning*, pages 681–688. PMLR.