IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 4 April 2024, accepted 30 April 2024, date of publication 15 May 2024, date of current version 24 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3401234

TUTORIAL

Bayesian Neural Networks via MCMC: A Python-Based Tutorial

ROHITASH CHANDRA[1,2], (Senior Member, IEEE), AND JOSHUA SIMMONS[3]
[1]Transitional Artificial Intelligence Research Group, School of Mathematics and Statistics, UNSW Sydney, Sydney, NSW 2052, Australia
[2]Centre for Artificial Intelligence and Innovation, Pingala Institute, Nausori, Fiji
[3]ARC Training Centre in Data Analytics for Resources and Environments, The University of Sydney, Camperdown, NSW 2050, Australia

Corresponding author: Rohitash Chandra (rohitash.chandra@unsw.edu.au)

The work of Joshua Simmons was supported by Australian Research Council Industrial Transformation Training Centre (ITTC) in Data Analytics for Resources and Environments under Grant IC19010031.

ABSTRACT Bayesian inference provides a methodology for parameter estimation and uncertainty quantification in machine learning and deep learning methods. Variational inference and Markov Chain Monte-Carlo (MCMC) sampling methods are used to implement Bayesian inference. In the past few decades, MCMC sampling methods have faced challenges in being adapted to larger models (such as deep learning models) and big data problems. Advanced proposal distributions that incorporate gradients, such as a Langevin proposal distribution, provide a means to address some of the limitations of MCMC sampling for Bayesian neural networks. Furthermore, MCMC methods have typically been constrained to statisticians, and hence not well-known among deep learning researchers. We present a tutorial for MCMC methods that covers simple Bayesian linear and logistic models, and Bayesian neural networks. The aim of this tutorial is to bridge the gap between theory and implementation via Python code, given a general sparsity of libraries and tutorials. This tutorial provides code in Python with data and instructions that enable their use and extension. We provide results for selected benchmark problems showing the strengths and weaknesses of implementing the respective Bayesian models via MCMC. We highlight the challenges in sampling multi-modal posterior distributions for the case of Bayesian neural networks and the need for further improvement of convergence diagnosis methods.

INDEX TERMS MCMC, Bayesian deep learning, Bayesian neural networks, Bayesian linear regression, Bayesian inference.

I. INTRODUCTION

Bayesian inference provides a probabilistic approach for parameter estimation in a wide range of models used across the fields of machine learning, econometrics, environmental and Earth sciences [1], [2], [3], [4], [5]. The term 'probabilistic' refers to the representation of unknown parameters as probability distributions rather than using fixed point estimates as in conventional machine learning models where gradient-based optimisation methods are prominent [6]. A probabilistic representation of unknown parameters requires a different approach to optimisation, which is known as *sampling* from a computational statistics point-of-view [7].

Markov Chain Monte Carlo (MCMC) sampling methods have been prominent for inference (estimation) of model parameters via the posterior probability distribution. In other words, Bayesian methods attempt to quantify the uncertainty in model parameters by marginalising over the predictive posterior distribution. Hence, in the case of neural networks, MCMC methods can be used to implement Bayesian neural networks that represent weights and biases as probability distributions [8], [9], [10], [11], [12]. Probabilistic machine learning provides natural way of providing uncertainty quantification in predictions [13], since the uncertainties can be obtained by probabilistic representation of parameters. A probabilistic representation (Bayesian approach) enables

The associate editor coordinating the review of this manuscript and approving it for publication was Derek Abbott.

one to obtain a set of predictions from the trained model, rather than having a single prediction from single-point estimate (Frequentist approach) using optimisation methods. The inference procedure in the Bayesian approach can be seen as a form of learning (optimisation) applied to the model parameters [9]. In this tutorial, we employ linear models and simple neural networks to implement MCMC sampling methods. The probabilistic representation of weights and biases in the respective models allows uncertainty quantification on model predictions.

We note that MCMC refers to a family of algorithms for implementing Bayesian inference for parameter and uncertainty estimation in models. Bayesian inference applications include statistical, graphical, and machine learning models. The differences in the model complexity from different domains have led to the existence of a wide range of MCMC sampling algorithms. Some of the prominent ones are Metropolis-Hastings algorithm [14], [15], [16], Gibbs sampling [17], [18], [19], Langevin MCMC [20], [21], [22], rejection sampling [23], [24], importance sampling [25], [26], sequential MCMC [27], adaptive MCMC [28], parallel tempering (tempered) MCMC [29], [30], [31], [32], reversible-jump MCMC [33], [34], specialised MCMC methods for discrete time series models [35], [36], [37], constrained parameter and model settings [38], [39], and likelihood free MCMC [40]. MCMC sampling methods have also been used for data augmentation [41], [42], model fusion [43], model selection [44], [45], and interpolation [46]. Apart from this, we note that MCMC methods have been prominent in a wide range of applications that include geophysical inversions [47], [48], [49], geoscientific models [5], [50], [51], environmental and hydrological modelling [52], [53], bio-systems modelling [54], [55], [56], and quantitative genetics [57], [58].

In the case of Bayesian neural networks, the number of model parameters that emerge from large neural network architectures and deep learning models pose challenges for MCMC sampling methods. Hence, progress in the application of Bayesian approaches to big data and deep neural networks has been slow. Research in this space has included a number of methods that have been fused with MCMC such as gradient-based methods [22], [59], [60], [61], [62], and evolutionary (meta-heuristic) algorithms which include differential evolution, genetic algorithms, and particle swarm optimisation [63], [64], [65], [66].

This use of gradients in MCMC was initially known as Metropolis-adjusted Langevin dynamics [22] and has shown promising performance for linear models [61] and has also been extended to Bayesian neural networks [62]. Hamiltonian Monte Carlo (HMC) sampling also employ gradient-based proposal distributions [60] and has been effectively applied to Bayesian neural networks [67]. In similar way, Langevin dynamics can be used to incorporate gradient-based stepping with Gaussian noise into the proposal distribution [61]. HMC avoids random walk behaviour using an auxiliary momentum vector and implementing Hamiltonian dynamics where the momentum samples are discarded later. The samples are hence less correlated, which tend to converge to the target distribution more rapidly. Another direction has been the use of better exploration features in MCMC sampling, such as parallel tempering MCMC with Langevin proposal distribution and parallel computing [62]. These have the ability to provide a competitive alternative to stochastic gradient-descent [68] and Adam optimizers [6], with the addition of uncertainty quantification in predictions. These methods have also been applied to Bayesian deep learning models such as Bayesian autoencoders [69] and Bayesian graph convolutional neural networks (CNNs) [70], which require millions of trainable parameters represented as posterior distributions. Recently, Kapoor et al. [66] combined tempered MCMC with particle swarm optimisation-based proposal distribution in a parallelized environment that showed more effective sampling when compared with the conventional approach. However, we note that large deep learning models can feature hundreds of millions to billions of parameters, which brings further challenges to sampling strategies, and hence the road is less travelled.

Variational inference provides an alternative approach to MCMC methods to approximate Bayesian posterior distribution [71], [72]. *Bayes by backpropagation* is a variational inference method that showed competitive results when compared to stochastic gradient descent and *dropout* methods used as approximate Bayesian methods [73]. Dropout is a regularisation technique that involves randomly dropping selected weights in forward-pass operation of backpropagation. This improves the generalization performance of neural networks and has been widely adopted [74]. Gal and Ghahramani [75] presented an approximate Bayesian methodology using dropout-based regularisation, which has been used for other deep learning models such as CNNs [76]. Later, Gal and Ghahramani [77] presented variational inference-based dropout technique for recurrent neural networks (RNNs); particularly, long-short term memory (LSTM) and gated recurrent unit (GRU) models for language modelling and sentiment analysis tasks.

We argue that the use of dropouts for Bayesian inference [76] cannot be seen as an alternative to MCMC sampling, as dropout-based inference does not sample directly from the posterior distribution. In the case of dropouts for Bayesian inference, we do not know the priors nor much about the posterior distribution. Furthermore, there is only a means to provide computational efficacy for capturing uncertainty during model training with weak theoretical foundations. Furthermore, in the Bayesian methodology, a probabilistic representation using priors is needed, which is questionable in the dropout methodology for Bayesian computation. Given that variational inference methods are seen as approximate Bayesian methods, we need to invest more effort in directly sampling from the posterior distribution for Bayesian deep learning models. This can only be possible if both communities (i.e., statistics and machine learning) are aware about the strengths and weaknesses of MCMC methods for sampling Bayesian neural networks. We note that such models can span hundreds to thousands of parameters, and go orders of magnitude higher when looking at Bayesian deep learning models. The progress of MCMC for deep learning has been slow, due to the lack of implementation details, libraries and tutorials that provide the balance of theory and implementation.

In this paper, we present a Python-based MCMC sampling tutorial for simple Bayesian linear models and Bayesian neural networks. We provide code in Python with data and instructions that enable their use and extension. We provide detailed instructions for sample code in a related Github repository which is easy to clone and run. Our code implementation is simple and relies on basic Python libraries such as *numpy* as the goal of this tutorial is to serve as a go-to document for beginners who have basic knowledge of machine learning models, and need to get hands-on experience with MCMC sampling. Hence, this is a code-based computational tutorial with a theoretical background. We provide results for selected benchmark problems showing the strengths and weaknesses of implementing the respective Bayesian models. Finally, we highlight the challenges in sampling multi-modal posterior distributions in the case of Bayesian neural networks and shed light on the use of convergence diagnostics.

The rest of the paper is organised as follows. In Section B, we present a background of related methods, including Bayesian inference and probability distributions. Section C presents the basic module of the core tutorial that includes the MCMC sampling code implementation in Python and Section D extends this for Bayesian linear models using regression problems. Section E presents Bayesian neural networks with MCMC using classification problems, and Section F presents experiments and results for benchmark problems. Section G provides an overview of convergence diagnosis, and Section I presents the discussion that concludes the paper with limitations and directions for future work.

II. BACKGROUND

A. BAYESIAN INFERENCE

We recall that Bayesian methods account for the uncertainty in prediction and decision-making via the posterior distribution [78]. Note that the posterior is the conditional probability determined after taking into account the prior distribution and the relevant evidence or data via sampling methods. Thomas Bayes (1702 – 1761) presented and proved a special case of the Bayes' theorem [79], [80] which is the foundation of Bayesian inference. However, Pierre-Simon Laplace (1749 – 1827) introduced a general version of the theorem and used it to approach problems [81]. Figure 1 gives an overview of the Bayesian inference framework that uses data with a prior and likelihood to estimate (by sampling from) the posterior distribution. This is the building block of the rest of the lessons that will feature Bayesian logistic regression and Bayesian neural networks.

Bayesian inference estimates unknown parameters using prior information or belief about the variable. Prior information is captured in the form of a distribution. A simple example of a prior belief is a distribution that has a positive real-valued number in some range. This essentially would imply a belief that our result or posterior distribution would likely be a distribution of positive numbers in some range which would be similar to the prior but not the same. If the posterior and prior both follow the same type of distribution, this is known as a *conjugate prior* [82]. If the prior provides useful information about the variable, rather than very loose constraints, it is known as an *informative prior*. The prior distribution is based on expert knowledge (opinion) and also dependent on the domain for different types of models [83], [84].

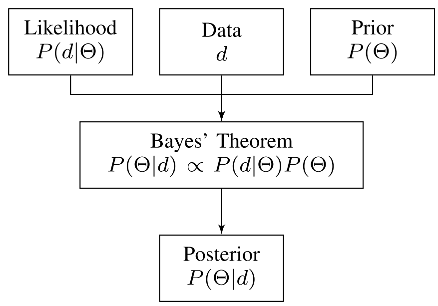


FIGURE 1. We show the relationship of the likelihood with data and the prior distribution for sampling the posterior distribution.

The need for efficient sampling methods to implement Bayesian inference has been a significant focus of research in computational statistics. This is especially true in the case of multimodal and irregular posterior distributions [32], [85], [86], which tend to dominate Bayesian neural networks [11], [87]. MCMC sampling methods are used to update the probability for a hypothesis (proposal Θ) as more information becomes available. The hypothesis is given by a prior probability distribution that expresses one's belief about a quantity (or free parameter in a model) before some data (d) are observed. MCMC sampling methods construct the posterior distribution ($P(\Theta|d)$) iteratively by using a proposal distribution, prior distribution $P(\Theta)$, and a likelihood function ($P(d|\Theta)$). As expressed in Equation 1

$$P(\Theta|d) = \frac{P(d|\Theta) \times P(\Theta)}{P(d)}. \tag{1}$$

$P(d)$ is the marginal distribution of the data and is often seen as a normalising constant and ignored. Hence, ignoring it, we can express Equation 1 as Equation 2

$$P(\Theta|d) \propto P(d|\Theta) \times P(\Theta). \tag{2}$$

The likelihood is a function of the parameters of a given model with observed data [88], which can be seen as a measure of fitness. Therefore, from an optimisation perspective, the likelihood function can be seen as a fitness or error function. The posterior distribution is constructed after taking into account the relevant evidence (data) and prior distribution, with the likelihood that consider the proposal (proposed parameters) and the model with given data. MCMC methods essentially implement Bayesian inference via an iterative numerical approach that marginalizes (integrates) over the posterior distribution [89]. Note that *probability* and *likelihood* are not the same in the field of statistics, while in everyday language they are used as if they are the same. The term "probability" refers to the possibility of something happening, in relation to a given distribution of data. The likelihood refers to the likelihood function that provides a measure of fitness in relation to a distribution. The likelihood function indicates which parameter (data) values are more likely than others in relation to a given probability distribution. Further details regarding Bayesian inference and MCMC sampling have been given in the literatures [90] and [91].

B. PROBABILITY DISTRIBUTIONS

1) GAUSSIAN (NORMAL) DISTRIBUTION

A normal probability density or distribution, also known as the Gaussian distribution, is described by two parameters, mean (μ) around which the distribution is centered, and the standard deviation (σ) which represents the spread (sometimes described by the variance, σ^2). We can fit a probability (normal) distribution to data from some source using the mean and variance. In a similar way, given a probability distribution, we can generate data and this process is known as sampling from the distribution. In sampling the distribution, we simply present random data points (uniform) to the distribution and obtain data that are transformed by the distribution. These parameters determine the shape of the probability distribution, e.g., if it is peaked or spread. Note that the normal distribution is symmetrical and caters for negative and positive numbers of real data.

Equation 3 presents the Gaussian distribution probability density function (PDF) for parameters μ and σ.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) \tag{3}$$

We will sample from this distribution in Python via the NumPy library [92] which covers the various distributions discussed in this tutorial. The SciPy library [93] is used to get a representation of the probability distribution function (PDF). The associated Github repository contains the code to generate Figures 2 to 4 using the Seaborn and Matplotlib Python libraries. Listing 1 shows an example of drawing samples from a Gaussian distribution. The SciPy and NumPy libraries feature all the following distributions.

We note that the mean and standard deviation are purely based on the given data and change depending on the problem. Let us visualise what happens to the distribution when the mean and standard deviation change, as shown in Figure 2.

```
import numpy as np
from numpy import random
from scipy import stats

# 10 random draws from a standard Gaussian
    distribution - N(0,1)
samples = random.randn(10)
# Gaussian distribution PDF with mean of 4 and
    standard deviation of 0.5
x = np.linspace(0, 8, 100)
pdf = stats.norm.pdf(x, loc=4, scale=0.5)
```

Listing 1. Random number generation for a Gaussian distribution.

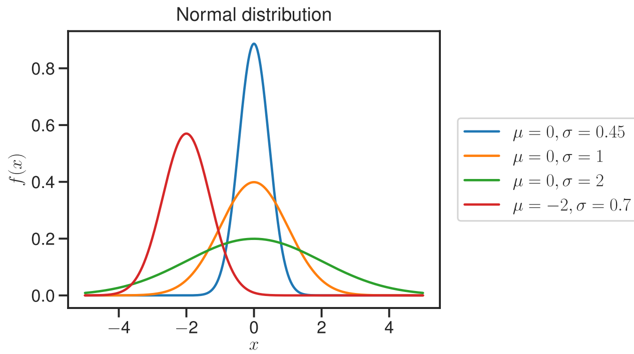


FIGURE 2. Normal distributions with different parameters, i.e., mean and the standard deviation.

2) MULTIVARIATE NORMAL DISTRIBUTION

The multivariate normal distribution or joint normal distribution generalises univariate normal distribution to more variables or higher dimensions, as shown in the PDF in Equation 4.

$$\begin{aligned} f(x_1, \ldots, x_M) \\ = \frac{1}{\sqrt{(2\pi)^M|\Sigma|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^{\mathrm{T}}\Sigma^{-1}(\mathbf{x}-\mu)\right) \end{aligned} \tag{4}$$

where $\mathbf{x}$ is a real M-dimensional column vector and $|\Sigma|$ is the determinant of the *symmetric covariance matrix*, which is *positive definite*.

3) GAMMA DISTRIBUTION

A gamma distribution is defined by the parameters shape (α) and rate (β), as shown in Equation 5

$$f(x; \alpha, \beta) = \frac{\beta^\alpha x^{\alpha-1} e^{-\beta x}}{\Gamma(\alpha)} \tag{5}$$

for $x > 0$ $\alpha, \beta > 0$; where $\Gamma(n) = (n-1)!$. Figure 3 presents the Gamma distribution for various parameter combinations, with the corresponding code in the accompanying Github repository. The corresponding inverse-Gamma (IG) distribution takes the same parameters with examples given in Figure 4 and is more appropriate for real positive numbers.

https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial/blob/main/01-Distributions.ipynb

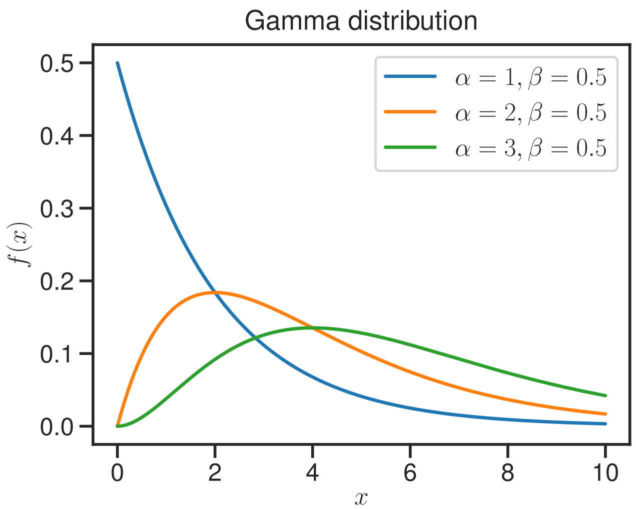


FIGURE 3. **Gamma distributions with different shape and rate parameters (α and β).**

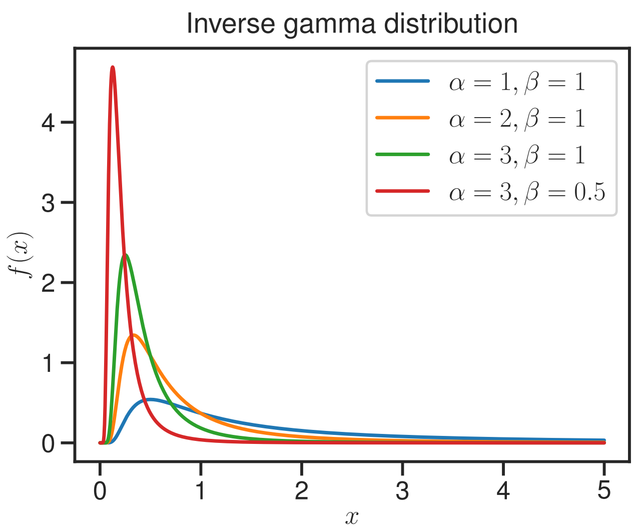


FIGURE 4. **Inverse gamma distributions with different shape and rate parameters (α and β).**

4) BINOMIAL DISTRIBUTION

We have only addressed real numbers with respective probability distributions so far; however, we also need to consider discrete numbers. The binomial distribution is a discrete probability distribution typically used for modelling binary classification problems. We begin with an example where a variable x takes the value 1 with probability p and the value 0 with probability $q = 1 - p$. We give the probability mass function for this distribution over the possible outcomes (x) in Equation 6.

$$f(x; p) = p^x (1-p)^{1-x} \tag{6}$$

for $x \in 0, 1$. The probability of getting exactly k successes ($x = 1$) in n independent trials ($f(k, n, p)$) is given as $\Pr(k; n, p)$ in Equation 7

$$\Pr(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k} \tag{7}$$

for $k = 0, 1, 2, \ldots, n$, where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

The Bernoulli distribution is a special case of the binomial distribution where a single trial is conducted, i.e. $n = 1$.

5) MULTINOMIAL DISTRIBUTION

In the case of binomial distribution, we catered for the case of two outcomes; however, we can consider the case of more than two outcomes. Suppose a single trial can result in k ($k \geq 2$) possible outcomes numbered $1, 2, \ldots, k$ and let $p_i = \mathbb{P}$(a single trial results in outcome i) ($\sum_{i=1}^{k} p_i = 1$). In the case of n independent trials, let X_i denote the number of trials resulting in outcome i (then $\sum_{i=1}^{k} X_i = n$). Then, we can state that the distribution of $(X_1, X_2, \ldots, X_k) \sim \text{Multinomial}(n; p_1, p_2, \ldots, p_k)$, and it holds

$$\mathbb{P}(X_1 = x_1, X_2 = x_2, \ldots, X_k = x_k)$$
$$= \frac{n!}{x_1! x_2! \ldots x_k!} p_1^{x_1} p_2^{x_2} \ldots p_k^{x_k}, \ 0 < p_i < 1, \ \sum_{i=1}^{k} p_i = 1. \tag{8}$$

III. MCMC

A Markov process is a random process with the property that the future is dependent on the present state and is independent of the past history. We note that a Markov process is uniquely defined by its transition probabilities $P(x'|x)$, which defines the probability of transitioning from any given state x to another given state x'. The Markov process has a unique stationary distribution $\pi(x)$ given the following two conditions are met.

1) There must exist a stationary distribution π which solves the detailed balance equations, and therefore requires that each transition $x \to x'$ is reversible. This implies that for every pair of states x, x', the probability of being in state x and moving to state x', must be equal to the probability of being in state x' and moving to state x; hence, $\pi(x)P(x' \mid x) = \pi(x')P(x \mid x')$.
2) The stationary distribution must be unique, which is guaranteed by ergodicity of the Markov process [94], [95], [96], [97]. Ergodicity is guaranteed when every state is aperiodic (i.e., the system does not return to the same state at fixed intervals) and positive recurrent (i.e., the expected number of steps for returning to the same state is finite). An ergodic system is one that mixes well; in other words, you get the same result whether you average its values over a given timeframe.

Given that $\pi(x)$ is chosen to be $P(x)$, the detailed balance condition becomes $P(x' \mid x)P(x) = P(x \mid x')P(x')$, which is re-written as shown in Equation 9.

$$\frac{P(x' \mid x)}{P(x \mid x')} = \frac{P(x')}{P(x)} \tag{9}$$

Algorithm 1 presents a basic MCMC sampler with random-walk proposal distribution that runs until a maximum number of samples (N_{max}) has been reached for training data, **d**.

Algorithm 1 A Basic MCMC Sampler Leveraging the Metropolis-Hastings Algorithm

Data: Training data, **d**
Result: N_{max} samples from the posterior distribution
- Initialise x_0 ;
for $i = 1$ *until* N_{max} **do**
  1. Propose a value $x'|x_i \sim q(x_i)$, where $q(.)$ is the proposal distribution;
  2. Given x', execute the model $f(x', \mathbf{d})$ to compute the predictions (output y) and the likelihood;
  3. Calculate the acceptance probability $\alpha = \min\left(1, \frac{P(x')}{P(x_i)} \frac{q(x_i|x')}{q(x'|x_i)}\right)$
  4. Generate a random value from a uniform distribution $u \sim U(0, 1)$;
  5. Accept or reject proposed value x';
  if $u < \alpha$ **then**
    accept the sample, $x_i = x'$
  else
    reject current and retain previous sample, $x_i = x_{i-1}$
  end
end

Algorithm 1 proceeds by proposing new values of the parameter x (Step 1) from the selected proposal distribution $q(.)$; in this case, a uniform distribution between 0 and 1. Conditional on these proposed values, the model $f(x', \mathbf{d})$ computes (predicts) an output using proposal x' and data **d** (Step 2). We compute the likelihood using the prediction and employ a Metropolis-Hasting criterion (Step 3) to determine whether to accept or reject the proposal (Step 5). We compare the acceptance ratio α with $u \sim U(0, 1)$, this enforces that the proposal is accepted with probability α. If the proposal is accepted, the chain moves to this proposed value. If rejected, the chain stays at the current value. The process is repeated until the convergence criterion is met, which is the maximum number of samples (N_{max}).

A. PRIORS

The prior distribution is generally based on belief, expert opinion or other information without viewing the data [9], [98]. Information to construct the prior can be based on past experiments or the posterior distribution of the model for related datasets. There are no hard rules for how much information should be encoded in the prior distribution; hence, we can take multiple approaches.

An *informative prior* gives specific and definite information about a variable. If we consider the prior distribution for the temperature tomorrow evening, it would be reasonable to use a normal distribution with an expected value (as mean) of today's evening temperature with a standard deviation of the temperature each evening for the entire season. A *weakly informative prior* expresses partial information about a variable. In the case of the prior distribution of evening temperature, a weakly informative prior would consider day time temperature of the day (as mean) with a standard deviation of day time temperature for the whole year. An *uninformative prior* or *diffuse prior* expresses vague information about a variable, such as the variable is positive or has some limit range.

A number of studies have been done regarding priors for linear models [99], [100] and Bayesian neural networks and deep learning models [101]. Hobbs et al. [102] presented a study for Bayesian priors in generalised linear models for clinical trials. We note that the insertion of prior knowledge in deep learning models [103] is different from defining priors in Bayesian deep learning models. Due to the similarity of terms, we caution the readers that these can be often mixed up.

In the case of Bayesian neural networks, the prior distribution can be based on the distribution of the weights and biases from similar neural network models. This can be seen as an example of expert knowledge and implemented in previous studies [69], [104]. Another example of expert knowledge is the concept of *weight decay* [105] regularisation (L2 or Ridge regression [106]) which restricts large weights and can be incorporated when defining the prior distribution (priors) [8], [9].

B. MCMC SAMPLER IN PYTHON

We begin with a deliberately simple example where we sample one parameter from a binomial distribution to demonstrate a simple MCMC implementation in Python. Looking at a simple binomial (e.g., coin flipping) likelihood (we will explore the likelihood later), given the data of k successes in n trials, we calculate the posterior probability of the parameter p that defines the chance of success for any given trial. MCMC sampling requires a prior distribution and a likelihood function to evaluate a set of parameters (proposed) for the given data and model. In other words, the likelihood is the measure of the quality of proposals obtained from a proposal distribution.

Listing 2 presents an implementation of this simple MCMC sampling exercise in Python of Algorithm 1.

In this example, we adopt a uniform distribution as an uninformative prior, only constraining the p to be between the values of 0 and 1 ($p \in [0, 1]$).

In MCMC sampling, a certain portion of the initial samples are discarded that is known as the burn-in (warmup) period. The burn-in period depends on the sampling problem (complexity of the model) and could be seen as an optimisation stage. In this simple case, we will use 25 % burn-in and in the case of neural network models, 50 % burn-in will likely be required. Essentially, during burn-in, we are discarding

https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial/blob/main/02-Basic-MCMC.ipynb

```python
# First define our likelihood function which will be dependent on provided 'data'
# in this case we will choose k = 50, n = 100
def likelihood(query_prob):
    '''
    Given the data of k successes in n trials, return a likelihood function which
    evaluates the probability that a single success (p) is query_prob for any given
    query_prob (between 0 and 1).
    '''
    k = 50
    n = 100
    return stats.binom.pmf(k, n, query_prob)

## MCMC Settings and Setup
n_samples = 10000 # number of samples to draw from the posterior
burn_in = 2500 # number of samples to discard before recording draws from the posterior

x = random.uniform(0, 1) # initialise a value of x0

# create an array of NaNs to fill with our samples
p_posterior = np.full(n_samples, np.nan)

print('Generating {} MCMC samples from the posterior:'.format(n_samples))

# now we can start the MCMC sampling loop
for ii in np.arange(n_samples):
    # Sample a value uniformly from 0 to 1 as a proposal
    x_new = random.uniform(0, 1)

    # Calculate the Metrpolis-Hastings acceptance probability based on the prior
    # (can be ignored in this case) and likelihood
    prior_ratio = 1 # for this simple example as discussed above
    likelihood_ratio = likelihood_function(x_new) / likelihood_function(x)
    alpha = np.min([1, likelihood_ratio * prior_ratio])

    # Here we use a random draw from a uniform distribution between 0 and 1 as a
    # method of accepting the new proposal with a probability of alpha
    # (i.e., accept if u < alpha)
    u = random.uniform(0, 1)
    if u < alpha:
        x = x_new # then update the current sample to the propoal for the next iteration

    # Store the current sample
    p_posterior[ii] = x
```

Listing 2. Python implementation of Algorithm 1.

material that is not part of the posterior distribution, since the posterior should feature good proposals (defined by the model accuracy and captured by the likelihood) which we get once the sampler goes towards convergence.

Typically, histograms of the posterior distribution and the trace plot are used to visualise the MCMC sampling performance. The histogram of the posterior distribution allows us to examine the mean and variance visually, while the trace plot shows the value of samples at each iteration, allowing us to examine the behavior and convergence of the MCMC.

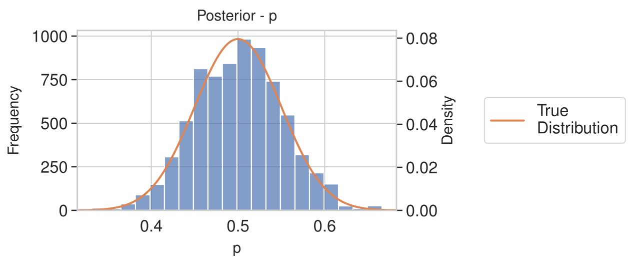
Although it is necessary to exclude the burn-in samples in the posterior distribution, it can be helpful to include them in the trace plots, in order to examine where the model started and how well it converged. We provide a visualization of the results that features a normal distribution obtained by a simple MCMC sampling from executing code Listing 2. The histogram of the posterior shows a normally distributed shape (Figure 5 - Panel (a), and the trace plot (Figure 5 - Panel (b) shows that the samples are distributed around the convergence value, as well as the burn-in samples which are in red. We also note that the value of the posterior is usually taken as the mean of the distribution, and in this case, the mean value is 0.502. The median can also be taken as a measure which would be more useful in irregular distributions.

IV. BAYESIAN LINEAR MODELS VIA MCMC

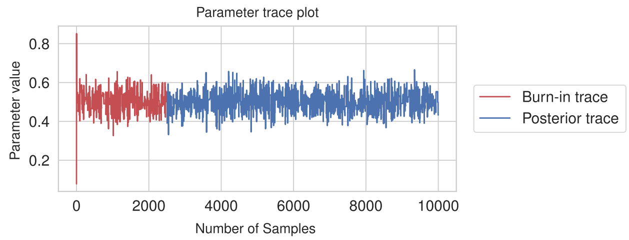
We provide details of implementing *Bayesian linear regression* that employs MCMC sampling with random-walk proposal distribution. We wish to model a dataset consisting of input (features or covariates) $\mathbf{x} = (x_1, \ldots, x_S)'$ and corresponding outputs $\mathbf{y} = (y_1, \ldots, y_S)'$ for S instances in data. This approach models the response observations as being composed of a regression component (the linear regression denoted by $f(\mathbf{x}, \theta)$) and a noise term (Gaussian distribution with a mean of zero ($\mu = 0$) and variance τ^2 (Equation 10).

$$\bar{\mathbf{y}} = f(\mathbf{x}, \theta) + e \qquad e \sim \mathcal{N}(0, \tau^2) \tag{10}$$

In the Bayesian linear regression, we treat the parameters (θ and τ^2) as random variables to be estimated (sampled) based on the data and likelihood. Therefore, the linear



(a) Posterior distribution



(b) Trace plot

FIGURE 5. Posterior and trace plot for the basic MCMC sampler given in Listing 2.

regression model in Equation 10 can be expressed as a Bayesian linear regression model, as given by Equation 11

$$p(\mathbf{y}|\mathbf{x}, \theta, \tau^2) \sim \mathcal{N}\left(f(\mathbf{x}, \theta), \tau^2\right). \tag{11}$$

Equation 12 expresses the general case of a linear model using a vector of input data $\mathbf{x}$ to obtain prediction $\mathbf{y}$.

$$f(\mathbf{x}, \theta) = \theta \mathbf{x}^T \tag{12}$$

In the case of Bayesian linear regression, θ is a set of distributions (typically Gaussian) rather than a fixed point estimate in conventional linear models. Therefore, we estimate the parameters (θ and τ^2) using MCMC sampling to obtain their posterior distributions. As mentioned earlier, the case of sampling can be seen as a form of optimisation, e.g., using gradient-based methods for learning the parameters of linear models and neural networks in the machine learning and neural networks literatures [107] and [108]. Furthermore, the key feature of a MCMC sampler is the ability to sample a posterior probability distribution that represents the parameters of a model rather than a fixed point estimate (frequentest approach) given by optimisation methods.

A. LIKELIHOOD

Our Bayesian approach for the problem requires sampling (estimating) the posterior distribution $p(\theta \mid \mathbf{y})$, that requires the definition of both a likelihood function $p(\theta \mid \mathbf{x})$ and prior distribution $p(\theta)$. We begin by defining the likelihood function, i.e. probability of the data given the model, which is given by the product of the likelihood for every data point in the dataset of S instances, as shown in Equation 13

$$p(\mathbf{y} \mid \mathbf{x}, \theta, \tau^2) = \prod_{t=1}^{S} p(\mathbf{y}_t \mid \mathbf{x}_t, \theta, \tau^2). \tag{13}$$

We note that for our MCMC sampler, we use the log-likelihood (i.e., taking the log of the likelihood function) to eliminate numerical instabilities, which can occur since we multiply probabilities that grow with the size of the data. It is also more convenient to maximize the log of the likelihood function since the logarithm is a monotonically increasing function of its argument, i.e., maximization of the log of a function is equivalent to maximization of the function itself. In order to transform a likelihood function into a log-likelihood, we will use the log product rule as given below

$$log_b(x \times y) = log_b(x) + log_b(y). \tag{14}$$

The log-likelihood simplifies the subsequent mathematical analysis and also helps avoid numerical instabilities due to the product of a large number of small probabilities. In the log-likelihood, Equation 13 is much simplified by computing the sum of the log probabilities as given in Equation 15

$$\ln p(y \mid \mathbf{x}, \theta, \tau^2) = \sum_{t=1}^{S} \ln p(y_t \mid \mathbf{x}_t, \theta, \tau^2). \tag{15}$$

In order to construct the likelihood function, we use our definition of the probability for each data point given the model as shown in Equation 11, and the form of the Gaussian distribution as defined in Equation 3. We use a set of weights and biases as the model parameters θ in our model $f(x, \theta)$, for S training data instances and variance τ^2. Our assumption of normally distributed errors leads to a likelihood given in Equation 16

$$p(\mathbf{y} \mid \mathbf{x}, \theta, \tau^2) = \frac{1}{(2\pi\tau^2)^{S/2}} \exp\left(-\frac{1}{2\tau^2}\sum_{t=1}^{S}(\mathbf{y}_t - f(\mathbf{x}_t, \theta))^2\right). \tag{16}$$

B. PRIOR

We note that a conventional linear model transforms into a Bayesian linear model with the use of a prior distribution and a likelihood function to sample the posterior distribution via the MCMC sampler. In Section III-B, we discussed the need to define a prior distribution for our model parameters θ and τ^2. In the case where the prior distribution comes from the same family as the posterior distribution, the prior and posterior are seen as conjugate distributions [109], [110]. The prior is called a *conjugate prior* for the likelihood function of the Bayesian model.

To implement conjugate priors in our linear model, we will assume a multivariate Gaussian prior for θ (Equation 17) and an inverse Gamma distribution (IG) for τ^2 (Equation 18).

$$\theta \sim \mathcal{N}(0, \sigma^2) \tag{17}$$

$$\tau^2 \sim IG(\nu_1, \nu_2) \tag{18}$$

The noise is defined by the spread (variance) of a normal distribution. We need to define the variance that is represented by τ^2 which cannot be a negative number, and hence we use IG in Equation 18. We do not know the appropriate value for τ^2, and hence we sample this parameter in a similar fashion as θ during the MCMC sampling process. We note that the prior for τ^2 must represent a distribution that can only sample positive real values, and we use the conjugate inverse Gamma prior with hyperparameters ν_1 and ν_2,representing the shape and scale parameters (see Section II-B).

We use the multivariate Gaussian distribution to represent the prior for parameters θ such as weights and bias of the linear model, which features negative and positive real numbers. Our model features more than one parameter, hence the multivariate Gaussian distribution is most appropriate for the prior. In this example, we adopt uninformative priors with hyperparameter values of $\sigma = 5$, $\nu_1 = 0$, and $\nu_2 = 0$ (Listing 6: Lines 15-17), but these values are user-defined and could be refined using trial experimental runs. These values are based on expert opinion from analysis of related trained models.

First, we revisit the multivariate normal distribution from Equation 4 to define the prior distribution for our linear model's parameters (weights and biases). Suppose that θ is our set of M parameters given by ($\theta = \theta_1, \ldots, \theta_M$). Since our prior is based on the normal distribution, we select the mean ($\mu = 0$) for each parameter to ensure we sample both positive and negative real numbers. Therefore, the mean μ is a vector of zeros and we get the prior using Equation 19

$$f(\theta) = \frac{1}{\sqrt{(2\pi)^M |\Sigma|}} \exp\left(-\frac{1}{2}(\theta)^{\mathrm{T}}\Sigma^{-1}(\theta)\right). \tag{19}$$

The covariance matrix Σ is a diagonal matrix with all values equal to σ^2 (scalar). Note that Σ^{-1} becomes I/σ^2 where I is an identity matrix (diagonal elements which are all ones). Hence, we take the numerator from Equation 19, i.e.

$$(\theta)^{\mathrm{T}}\Sigma^{-1}(\theta) \tag{20}$$

becomes

$$\frac{(\theta)^{\mathrm{T}}\mathbf{I}(\theta)}{\sigma^2}. \tag{21}$$

We note that multiplying the identity matrix with any other matrix is the matrix itself; hence, we get θ^2 in the numerator. We can now move to the inverse-Gamma distribution used to define the prior for our model's variance (τ^2) and sampled (just as θ) and given by Equation 22

$$f(\tau^2) = \frac{\nu_1^{\nu_2}}{\Gamma(\nu_1)}\left(\frac{1}{\tau^2}\right)^{\nu_1+1} \exp\left(\frac{-\nu_2}{\tau^2}\right). \tag{22}$$

We note that $\nu_1^{\nu_2}/\Gamma(\nu_1)$ is a constant which can be dropped considering proportionality. We take into account the product of all our sampled parameters (θ and τ^2) to define the combined prior, as given by Equation 23

$$p(\theta) \propto \frac{1}{(2\pi\sigma^2)^{M/2}} \times \exp\left\{-\frac{1}{2\sigma^2}\left(\sum_{i=1}^{M}\theta^2\right)\right\} \times \tau^{-2(1+\nu_1)} \exp\left(\frac{-\nu_2}{\tau^2}\right). \tag{23}$$

C. PYTHON IMPLEMENTATION

The Python code presented in Listing 3 begins the implementation of a Bayesian linear model using MCMC sampling. First, we define our simple linear model as given in Equation 12 using class LinearModel (Line 1) of Listing 3 and define functions to evaluate the proposal (line 13). We encode the parameters proposed from the MCMC sampler class into the linear model (Line 30) to get the prediction (Line 25).

Now that we have a class for our linear model, we can define the functions that will allow us to carry out MCMC sampling for the model parameters. We define our log-likelihood function using Equation 16, which becomes Equation 24

$$logp(\mathbf{y} \mid \mathbf{x}, \theta, \tau^2) = -log((2\pi\tau^2)^{S/2}) \quad -\frac{1}{2\tau^2}\sum_{t=1}^{S}(\mathbf{y}_t - f(\mathbf{x}_t, \theta))^2. \tag{24}$$

Furthermore, we define our log-prior using Equation 23 which becomes Equation

$$\log p(\theta) \propto -\frac{M}{2}\log 2\pi\sigma^2 - \frac{1}{2\sigma^2}\left(\sum_{i=1}^{M}\theta^2\right) - (1+\nu_1)\log\tau^2 - \frac{\nu_2}{\tau^2}. \tag{25}$$

We present Python implementation of the log-likelihood (Lines 2 - 18) and the prior (Lines 21 - 37) in Listing 4.

Before running the MCMC sampler, we need to set up the sampler hyperparameters, such as the maximum sampling time and burn-in period (Listing 7). We also need to assign hyperparameters that define the priors such as Gaussian prior variance (σ^2) and the IG prior parameters, ν_1 and ν_2 (Listing 6: Lines 15, 16 and 17). First, we need to generate an initial sample for our parameters and initialise arrays to capture the samples that form the posterior distribution, the accuracy, and the model predictions as shown in code Listing 5 (Lines 5-23). Then we proceed with sampling as per the MCMC sampling algorithm detailed in Algorithm 1 and code Listing 5. This algorithm uses a Gaussian random-walk distribution for the parameter proposals (θ_p and τ_p^2). We perturb the current value with Gaussian noise as shown in

https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial/blob/main/03-Linear-Model.ipynb

```python
class LinearModel:
    '''
    Simple linear model with a single output (y) given the covariates x_1...x_M of the form:
    y = w_1 * x_1 + ... + w_M * x_M + b
    where M = number of features, w are the weights, and b is the bias.
    '''
    # Initialise values of model parameters
    def __init__(self):
        self.w = None
        self.b = None

    # Function to take in data and parameter sample and return the prediction
    def evaluate_proposal(self, data, theta):
        '''
        Encode the proposed parameters and then use the model to predict
        Input:
            data: (N x M) array of data
            theta: (M + 1) vector of parameters. The last element of theta consitutes the bias term (giving M + 1 elements)
        '''
        self.encode(theta)  # method to encode w and b
        prediction = self.predict(data) # predict and return
        return prediction

    # Linear model prediction
    def predict(self, x_in):
        y_out = x_in.dot(self.w) + self.b
        return y_out

    # Helper function to split the parameter vector into w and band store in the model
    def encode(self, theta):
        self.w =  theta[0:-1]
        self.b = theta[-1]
```

Listing 3. Python implementation of a simple linear regression model.

Equations 26 and 27, respectively.

$$\theta_p \sim \theta_{p-1} + \mathcal{N}(0, \Delta_\theta) \tag{26}$$

$$\eta_p \sim \eta_{p-1} + \mathcal{N}(0, \Delta_\eta) \tag{27}$$

We implement the MCMC sampler with a Gaussian random-walk proposal for $\eta_p = \log \tau_p^2$, where we use η to represent τ^2 in log-space (Listing 5: Line 29). The step sizes for the proposals are determined by the hyperparameters Δ_θ and Δ_η which define the variance for the proposal of θ_p and η_p respectively. Once we sample η, we take the exponential to convert it back to the original form (see Line 30) and obtain τ^2.

After getting the proposal for the parameters, i.e. θ and τ^2, we call the likelihood and prior functions to obtain their respective values, as shown in Lines 32 - 35 of Listing 5. Note that the log-likelihood is used and hence the ratio of previous and current likelihood will need to consider log laws (rules), i.e., we note the log product rule in Equation 28 and the quotient rule in Equation 29. We use these rules in Lines 37 and 38 of Listing 5. Based on Equation 9 and taking the quotient rule into account since we are in the log space, we then accept/reject the proposed value according to the Metropolis-Hastings acceptance ratio (Line 42) as shown in Lines 41 - 54.

$$log_b(x \times y) = log_b(x) + log_b(y) \tag{28}$$

$$log_b(x/y) = log_b(x) - log_b(y) \tag{29}$$

Now that we have the sampler code (Listing 5), we can create an MCMC class that brings together the model, data, hyperparameters and sampling algorithm as shown in Listing 6.

We can then run the MCMC sampler function as shown in Listing 7, Line 12. After the code runs, we get the results (Line 14) and can generate the predictions from model posterior draws (Line 16) of the trained Bayesian linear model.

V. BAYESIAN NEURAL NETWORKS VIA MCMC

A. NEURAL NETWORKS

We utilise a simple neural network, also known as a multilayer perceptron to demonstrate the process of training a Bayesian neural network via the MCMC sampler. A neural network model $f(x)$ is made up of a series of computations, that transform inputs to their corresponding outputs $\{\bar{x}_t, y_t\}$. Neural networks feature layers of neurons whose value is determined based on a linear combination of inputs from the previous layer, with an activation function.

We consider a simple neural network with one hidden layer with four input neurons, five hidden neurons and one output neuron, as shown in Figure 6. As an example, we can calculate the output value of the jth neuron in the first hidden layer of a network (h, j) using a weighted combination of the m inputs $(\bar{x}_t)$, as shown in Equation 30.

$$g\left(\delta_{h,j} + \sum_{i=1}^{m} w_{i,j}\bar{x}_{t,i}\right) \tag{30}$$

```
# Define the log-likelihood function
def likelihood_function(self, theta, tausq):
    '''
    Calculate the likelihood of the data given the parameters
    Input:
        theta: (M + 1) vector of parameters. The last element of theta consitutes the bias term (giving M + 1 elements)
        tausq: variance of the error term
    Output:
        log_likelihood: log likelihood of the data given the parameters
        model_prediction: prediction of the model given the parameters
        accuracy: accuracy (RMSE) of the model given the parameters
    '''
    # first make a prediction with parameters theta
    model_prediction = self.model.evaluate_proposal(self.x_data, theta)
    accuracy = self.rmse(model_prediction, self.y_data) #RMSE error metric
    # now calculate the log likelihood
    log_likelihood = np.sum(-0.5 * np.log(2 * np.pi * tausq) - 0.5 * np.square(self.y_data - model_prediction) / tausq)
    return [log_likelihood, model_prediction, accuracy]

# Define the prior
def prior(self, sigma_squared, nu_1, nu_2, theta, tausq):
    '''
    Calculate the prior of the parameters
    Input:
        sigma_squared: variance of normal prior for theta
        nu_1: parameter nu_1 of the inverse gamma prior for tau^2
        nu_2: parameter nu_2 of the inverse gamma prior for tau^2
        theta: (M + 1) vector of parameters. The last element of theta consitutes the bias term (giving M + 1 elements)
        tausq: variance of the error term
    Output:
        log_prior: log prior
    '''
    n_params = self.theta_size # number of parameters in model
    part1 = -1 * (n_params / 2) * np.log(sigma_squared)
    part2 = 1 / (2 * sigma_squared) * (sum(np.square(theta)))
    log_prior = part1 - part2 - (1 + nu_1) * np.log(tausq) - (nu_2 / tausq)
    return log_prior
```

Listing 4. **Python implementation of likelihood and prior functions for linear regression model to be incorporated into the MCMC sampling class.**

where, the bias ($\delta_{h,j}$) and weights (w_i for each of the m inputs) are parameters to be sampled (trained or estimated), and $g(.)$ is the activation function that is used to perform a nonlinear transformation. In our case, the function $g(.)$ is the *sigmoid activation function*, used for the hidden and output layers as shown in Figure 6.

We train the model to approximate the function f such that $f(\mathbf{x}) = \bar{y}$ for all input-output pairs of instances from the training dataset. We extend the calculation of outputs in the hidden layer to calculate the output $f(\bar{x}_t)$ as shown in Equation 31.

$$f(\mathbf{x}) = g\left(\delta_o + \sum_{j=1}^{H} v_h \times g\left(\delta_h + \sum_{i=1}^{m} w_{i,j}\mathbf{x}_i\right)\right) \tag{31}$$

where H is the number of neurons in the hidden layer, δ_o is the bias for the output, and v_h are the weights from the hidden layer to the output neuron. The complete set of parameters for the neural network model (Figure 6) is made up of $\theta = (\tilde{\mathbf{w}}, \tilde{\mathbf{v}}, \delta)$, where $\delta = (\delta_o, \delta_h)$. $\tilde{\mathbf{w}}$ are the weights transforming the input to hidden layer. $\tilde{\mathbf{v}}$ are the weights transforming the hidden to output layer. δ_h is the bias for the hidden layer, and δ_o is the bias for the output layer.

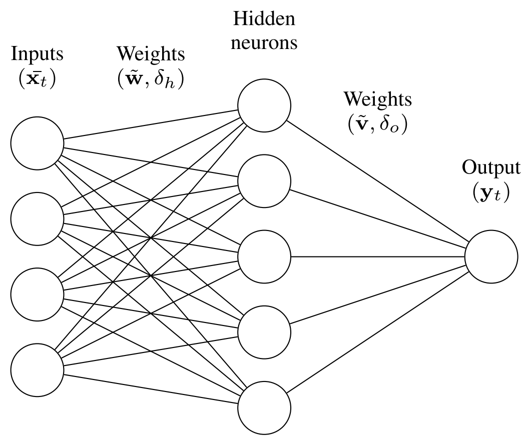


FIGURE 6. **Simple neural network with a single hidden layer. The information is passed and processed from the input to hidden and then finally to the output layer.**

B. BAYESIAN NEURAL NETWORKS

A Bayesian neural network is a probabilistic implementation of a standard neural network with the key difference being that the weights and biases are represented via the

```
# MCMC sampler
def sampler(self):
    # Run the sampler for a defined linear model
    # Define empty arrays to store the sampled posterior values
    pos_theta = np.ones((self.n_samples, self.theta_size))
    # posterior defining the variance of the noise in predictions
    pos_tau = np.ones((self.n_samples, 1))
    # record output f(x) over all samples
    pred_y = np.ones((self.n_samples, self.x_data.shape[0]))
    # record the RMSE of each sample
    rmse_data = np.zeros(self.n_samples)

    ## Initialisation:initialise theta - the model parameters
    theta = np.random.randn(self.theta_size)
    # make initial prediction
    pred_y[0,] = self.model.evaluate_proposal(self.x_data, theta)
    # initialise eta - we sample eta as a gaussian random walk in the log space of tau^2
    eta = np.log(np.var(pred_y[0,] - self.y_data))
    tausq_proposal = np.exp(eta)
    # calculate the prior
    prior_val = self.prior(self.sigma_squared, self.nu_1, self.nu_2, theta, tausq_proposal)
    # calculate the likelihood considering observations
    [likelihood, pred_y[0,], _, rmse_data[0]] = self.likelihood_function(theta, tausq_proposal)

    ## Run the MCMC sample for n_samples
    for ii in np.arange(1,self.n_samples):
        # Sample new values for theta and tau using a Gaussian random walk
        theta_proposal = theta + np.random.normal(0, self.step_theta, self.theta_size)
        eta_proposal = eta + np.random.normal(0, self.step_eta, 1) # sample tau^2 in log space
        tausq_proposal = np.exp(eta_proposal)
        # calculate the prior
        prior_proposal = self.prior(
        self.sigma_squared, self.nu_1, self.nu_2, theta_proposal, tausq_proposal)
        # calculate the log-likelihood considering observations
        [likelihood_proposal, pred_y[ii,], _, rmse_data[ii]] = self.likelihood_function(theta_proposal,
    tausq_proposal)
        # Noting that likelihood_function and prior_val return log likelihoods, we can calculate the
    acceptance probability
        diff_likelihood = likelihood_proposal - likelihood
        diff_priorlikelihood = prior_proposal - prior_val
        mh_prob = min(1, np.exp(diff_likelihood + diff_priorlikelihood))
        # sample to accept or reject the proposal according to the acceptance probability
        u = np.random.uniform(0, 1)
        if u < mh_prob:
            # accept and update the values
            likelihood = likelihood_proposal
            prior_val = prior_proposal
            theta = theta_proposal
            eta = eta_proposal
            # store to make up the posterior
            pos_theta[ii,] = theta_proposal
            pos_tau[ii,] = tausq_proposal
        else:
            # reject move and store the old values
            pos_theta[ii,] = pos_theta[ii-1,]
            pos_tau[ii,] = pos_tau[ii-1,]
    # store the posterior (samples after burn in) in a pandas dataframe and return
    self.pos_theta = pos_theta[self.n_burnin:, ]
    self.pos_tau = pos_tau[self.n_burnin:, ]
    self.rmse_data = rmse_data[self.n_burnin:]
    # split theta into w and b
    results_dict = {'w{}'.format(_): self.pos_theta[:, _].squeeze() for _ in range(self.theta_size-1)}
    results_dict['b'] = self.pos_theta[:, -1].squeeze()
    results_dict['tau'] = self.pos_tau.squeeze()
    results_dict['rmse'] = self.rmse_data.squeeze()
    results_df = pd.DataFrame.from_dict(results_dict)
    return results_df
```

Listing 5. **Python implementation of an MCMC sampler for the linear model.**

posterior probability distributions rather than single point values as shown in Figure 7. Similar to canonical neural networks [111], Bayesian neural networks also have universal continuous function approximation capabilities. However, the posterior distribution of the network parameters allows uncertainty quantification on the predictions.

The task for MCMC sampling is to estimate (sample) the posterior distributions representing the weights and biases

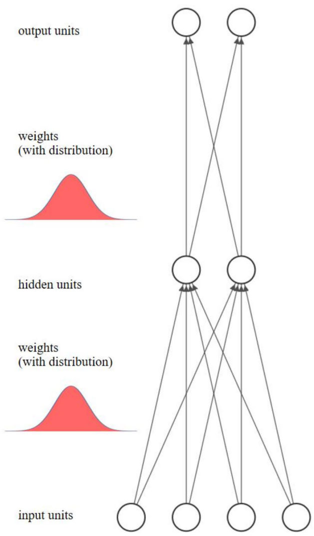
```python
class MCMC:
    def __init__(self, n_samples, n_burnin, x_data, y_data):
        self.n_samples = n_samples # number of MCMC samples
        self.n_burnin = n_burnin # number of burn-in samples
        self.x_data = x_data # (N x M)
        self.y_data = y_data # (N x 1)
        self.theta_size = x_data.shape[1] + 1 # weights for each feature and a bias term (M+1)

        # MCMC sampler hyperparameters - defines the variance term in our Gaussian random walk
        self.step_theta = 0.02;
        self.step_eta = 0.01; # note eta is used as tau in the sampler to consider log scale.

        # model hyperparameters
        # considered by looking at distribution of  similar trained  models - i.e distribution of weights and bias
        self.sigma_squared = 5
        self.nu_1 = 0
        self.nu_2 = 0

        # initisalise the linear model class
        self.model = LinearModel()

        # store output
        self.pos_theta = None
        self.pos_tau = None
        self.rmse_data = None

        # functions defined above - this is poor practice, but done for readability
        # and clarity
        self.likelihood_function = MethodType(likelihood_function, self)
        self.prior_val = MethodType(prior_val, self)
        self.sampler = MethodType(sampler, self)

    def model_draws(self, num_samples = 10):
        '''
        Simulate new model predictions (mu) under the assumption that our posteriors are
        Gaussian.
        '''
        # num_samples x num_data_points
        pred_y = np.zeros((num_samples,self.x_data.shape[0]))
        sim_y = np.zeros((num_samples,self.x_data.shape[0]))

        for ii in range(num_samples):
            theta_drawn = np.random.normal(self.pos_theta.mean(axis=0), self.pos_theta.std(axis=0), self.theta_size)
            tausq_drawn = np.random.normal(self.pos_tau.mean(), self.pos_tau.std())

            [_, pred_y[ii,:], sim_y[ii,:],_] = self.likelihood_function(
                theta_drawn, tausq_drawn
            )
        return pred_y, sim_y
```

Listing 6. Python implementation of an MCMC class for Bayesian linear model.

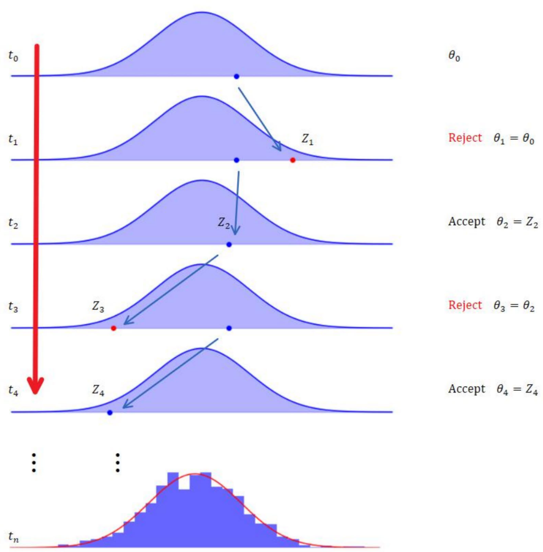
```python
## MCMC Settings and Setup
n_samples = 20000 # number of samples to draw from the posterior
burn_in = int(n_samples* 0.25) # number of samples to discard before recording draws from the posterior

# Generate toy data
n_data = 100
n_features = 1
x_data = np.repeat(np.expand_dims(np.linspace(0, 1, n_data),axis=-1),n_features,axis=1)
y_data = 3 * x_data[:,0] + 4 + np.random.randn(n_data) * 0.5

# Initialise the MCMC class
mcmc = MCMC(n_samples, burn_in, x_data, y_data)
# Run the sampler
results = mcmc.sampler()
# Draw sample models from the posterior
pred_y, _ = mcmc.model_draws(num_samples=100)
```

Listing 7. Code to call the MCMC sampling class and fit a model to some toy data.

of the neural network that best fit the data. Perhaps, it can be argued that the method should be called an estimator, but we will stick to the sampler as given in the literature. As in the previous examples, we begin inference with prior



(a) Bayesian neural network



(b) MCMC sampling

FIGURE 7. Bayesian neural network and MCMC sampling adapted from [104].

distributions over the weights and biases of the network and use a sampling scheme to find the posterior distributions given training data. Since non-linear activation functions exist in the network, the *conjugacy* of prior and posterior is lost. Therefore, we must employ an MCMC sampling scheme and make assumptions about the distribution of errors.

We specify the model similar to the Bayesian linear regression, assuming a Gaussian error as given in Equation 32.

$$y = f(\mathbf{x}, \theta) + e \qquad e \sim \mathcal{N}(0, \tau^2) \tag{32}$$

This leads to the same likelihood function as presented in logarithmic form in Equation 24. As in Section IV-B, we adopt Gaussian priors for all parameters of the model (θ), with zero mean and a user-defined variance (σ^2), and an IG distribution for the variance of the error model (τ^2), with parameters ν_1 and ν_2. The *likelihood function* and *prior* function remain unchanged from their definition in Listing 4.

C. MULTINOMIAL LIKELIHOOD FOR CLASSIFICATION PROBLEMS

We note that neural network models are also prominent for classification problems apart from regression and prediction problems. Bayesian neural networks via the MCMC sampler require an appropriate likelihood function, suitable for discrete outcomes, to capture classification problems. Hence, we use the multinomial likelihood, which is applicable to both binary and multi-class classification problems. We define the multinomial log-likelihood function for the classification problems using Equation 33

$$\log\left(p(y|\theta)\right) = \sum_{t \in N} \sum_{k=1}^{K} z_{t,k} \log \pi_k \tag{33}$$

for classes $k = 1, \ldots, K$, where π_k is the output of the neural network model after applying the transfer function, and N is the number of instances in the training data. In this case, we utilize the *softmax function* [112] as the transfer function:

$$\pi_k = \frac{\exp(f(x_p))}{\sum_{k=1}^{K} \exp(f(x_k))} \tag{34}$$

for $k = 1, \ldots, K$. $z_{t,k}$ is an indicator variable for the given instance of data t. We define class k in the data by

$$z_{t,k} = \begin{cases} 1, & \text{if } y_t = k \\ 0, & \text{otherwise.} \end{cases} \tag{35}$$

We note that we do not use the noise parameter (i.e., τ^2) as in the case of the inverse gamma distribution for the Gaussian likelihood for the regression case (Equations 16 and 24); hence, we do not need a prior distribution for the noise. We will only use a Gaussian prior for weights and biases of the neural network model. Therefore, in the case of classification, our prior distribution from Equation 25 simplifies to Equation 36

$$p(\theta) \propto \frac{1}{(2\pi\sigma^2)^{M/2}} \times \exp\left\{ -\frac{1}{2\sigma^2}\left(\sum_{i=1}^{M} \theta^2\right)\right\} \tag{36}$$

and finally, the log-prior can be expressed as Equation 37

$$\log p(\theta) \propto -\frac{M}{2}\log 2\pi\sigma^2 \times -\frac{1}{2\sigma^2}\left(\sum_{i=1}^{M} \theta^2\right). \tag{37}$$

D. TRAINING NEURAL NETWORKS VIA BACKPROPAGATION

We note that typically random-walk proposal distributions are used for small scale-models such as linear models; however, neural network models feature a large number of parameters. The choice of a proposal distribution is essential for models with large number of parameters. We need to incorporate gradients into our proposal distribution for better sampling, and we will begin by examining how gradients are incorporated in conventional neural networks and deep learning models.

Gradient-based optimization has been the backbone for the backpropagation training algorithm [113] and widely used in machine learning. A prominent implementation is *stochastic gradient descent* (SGD) which involves stepping through the parameter space iteratively in a stochastic manner using gradients, to optimize a differentiable objective function. The method has been prominently featured in the backpropagation algorithm for training various neural network architectures, including deep learning models [114], [115], [116]. Backpropagation involves a forward pass which propagates information forward to get the prediction (decision) at the output layer, and a backward pass to compute the local gradients for each of the parameters (weights and biases). These gradients are then used to inform the update of the model parameters in an iterative process, where the parameters are updated at each step. The training of neural networks is also considered as solving a non-convex optimization problem $\operatorname{argmin} L(\theta)$; where, $\theta \in R^n$ is the set of parameters and L is the loss function. We give the parameter (weight) update for an iteration (epoch) of SGD in Equation 38

$$\theta_k = \theta_{k-1} - a_{k-1}\nabla L(\theta_{k-1}) \tag{38}$$

where, θ_k denotes the k^{th} iteration, a_k is the learning rate, and $\nabla L(\theta_k)$ denotes the gradient.

We note that the *learning rate* is a user-defined hyperparameter, which depends on the problem and data at hand. It is typically determined through tuning using cross-validation or trial and error. Extensions of the backgropagation algorithm employing SGD were proposed to address limitations. These include the use of *weight decay* regularization during training to improve generalization ability [105], a momentum mechanism for faster training [117], [118], adaptive learning rate [118], and second-order gradient methods [119] which, although efficient, have problems in scaling up computationally with larger models. In the last decade, the extensions made were not only to improve the training accuracy, but to scale up better computationally for large deep learning models. These led to the development of methods such as the adaptive gradient algorithm (AdaGrad) [120], AdaDelta [121], and Adam (adaptive moment estimation) [122]. These algorithms adapt the learning rate automatically during the training, taking into account the recent history of the optimization process.

1) LANGEVIN PROPOSAL DISTRIBUTION

We mentioned earlier that random-walk proposal distributions are suited for small-scale models, and better proposal distributions would be required for neural network models. Although simple neural networks have a much lower number of parameters, when compared to deep learning models, training simple neural networks with MCMC sampling is a challenge with random-walk proposal distribution. We need to utilize the properties of backpropagation algorithm and the mechanism of weight update using gradients. Hence, we utilize *stochastic gradient Langevin dynamics* [61] for the proposal distribution, which features the addition of noise to the stochastic gradients. The method has shown to be effective for linear models [61] which motivated its use in Bayesian neural networks. In the literature, Langevin MCMC has been very promising for simple and deep neural networks [62], [69], [70]. Hence, we draw the proposed values for the parameters (θ^p) according to a one-step (epoch) gradient, as shown in Equation 39.

$$\theta^p \sim \mathcal{N}(\bar{\theta}^{[s]}, \Sigma_\theta) \tag{39}$$

A Gaussian distribution with a standard deviation of Σ_θ, and mean ($\bar{\theta}^{[s]}$) calculated using a gradient based update (Equation 40 of the parameter values from the previous step ($\theta^{[s]}$).

$$\bar{\theta}^{[s]} = \theta^{[s]} + r \times \nabla E(\theta^{[\mathbf{s}]}) \tag{40}$$

with learning rate r and gradient update ($\nabla E(\theta^{[s]})$) according to the model residuals.

$$\begin{aligned} E(\theta^{[s]}) &= \sum_{t \in \mathcal{T}} (y_t - F(x_i, \theta^{[\mathbf{s}]}))^2 \\ \nabla E(\theta^{[s]}) &= \left(\frac{\partial E}{\partial \theta_1}, \ldots, \frac{\partial E}{\partial \theta_L}\right). \end{aligned} \tag{41}$$

Hence, the Langevin proposal distribution (also referred as Langevin-gradient) consists of 2 parts:

1) Gradient descent-based weight update
2) Addition of Gaussian noise from $\mathcal{N}(0, \Sigma_\theta)$

We need to ensure that the detailed balance is maintained while sampling, since the Langevin proposals are not symmetric. We note that MCMC implementations with relaxed detailed balance conditions for some applications also exist [123]. Therefore, we use a combined update in the Metropolis-Hastings step, which accepts the proposal θ^p for a position s with the probability α, as shown in Equation 42

$$\alpha = \min\left\{1, \frac{p(\theta^p|\mathbf{y})q(\theta^{[s]}|\theta^p)}{p(\theta^{[s]}|\mathbf{y})q(\theta^p|\theta^{[s]})}\right\} \tag{42}$$

where $p(\theta^p|\mathbf{y})$ and $p(\theta^{[s]}|\mathbf{y})$ can be computed using the likelihood and prior (Equations (16) and (23)). We give the ratio of the proposed and the current $q(\theta^p|\theta^{[s]})$ in Equation 43

$$q(\theta^{[s]}|\theta^p) \sim N(\bar{\theta}^{[s]}, \Sigma_\theta) \tag{43}$$

which is based on a one-step (epoch) gradient $\nabla E_y[\theta^{[s]}$ and learning rate r, as given in Equation 44

$$\bar{\theta}^{[s]} = \theta^{[s]} + r \times \nabla E_y[\theta^{[s]}]. \tag{44}$$

Thus, this ensures that the detailed balance condition holds, and the sequence $\theta^{[s]}$ converges to draw from the posterior

$p(\theta|y)$. Since our implementation is in the log-scale, we give the log-posterior in Equation 45

$$\log(p(\theta|y)) = \log(p(\theta)) + \log(p(y|\theta)) + \log(q(\theta|\theta^*)) \tag{45}$$

Algorithm 2 gives a full description of the Langevin MCMC sampling scheme with user-defined parameters that include the maximum number of samples (S_{max}), rate of Langevin-gradient proposals (L_{prob}), and learning rate r used for the Langevin-gradient proposals. We note that in a standard Langevin MCMC approach, $L_{prob} = 1$ and Gaussian noise is already part of Langevin-gradient distribution. However, in our implementation, we use a combination of random-walk proposal distribution with Langevin-gradients, as this is computationally more efficient. Langevin-gradients require more computational time due to gradient computation when compared to random-walk proposals, especially in larger models.

We begin by drawing initial values for the θ from the prior distribution given in Equation (23) (Stage 1.1). We draw a new proposal for θ^p (which incorporates the model weights and biases and τ^2 from either a Langevin-gradient or random-walk proposal distribution (Stage 1.2). We evaluate the proposal using the Bayesian neural network (BNN) model with the log-likelihood function in Equation 16 (Stage 1.4) and the prior given in Equation (23) (Stage 1.3). We can then check if the proposal should be accepted using Metropolis-Hastings condition (Stage 1.5 and 1.6). If accepted, the proposal becomes part of the chain, else we retain the last accepted state as the current state of the chain. We repeat the procedure until the maximum samples are reached (S_{max}). Finally, we execute the post-sampling stage, where we obtain the posterior distribution by concatenating the history of the samples in the chain.

Algorithm 2 Bayesian Neural Network via Langevin MCMC Sampling

Data: Dataset
Result: Posterior distribution of model parameters (weights and biases)
- Stage 1.0: Metropolis Transition
- 1.1 Draw initial values θ_0 from the prior
for *each s until* S_{max} **do**
  1.2 Draw κ from a Uniform-distribution [0,1]
  if $\kappa \leq L_{prob}$ **then**
    Use Langevin-gradient proposal distribution: $\theta^p \sim \mathcal{N}(\bar{\theta}^{[s]}, \Sigma_\theta)$
  end
  else
    Use random-walk proposal distribution: $\theta^p \sim \mathcal{N}(\theta^{[s]}, \Sigma_\theta)$
  end
  1.3 Evaluate prior given in Equation 23
  1.4 Evaluate log-likelihood given in Equation 16
  1.5 Compute the posterior probability for Metropolis-Hastings condition - Equation 45
  1.6 Draw u from a Uniform-distribution [0,1]
  if $\log(u) \leq log(p(y|\theta))$ **then**
    Accept replica state: $\theta^{[s+1]} \leftarrow \theta^p$
  end
  else
    Reject and retain previous state: $\theta^{[s+1]} \leftarrow \theta^{[s]}$
  end
end

E. PYTHON IMPLEMENTATION

We first define and implement the simple neural network module (class), and implement methods (functions) for the forward and backward pass to calculate the output of the network, given a set of inputs. We need to compute the gradients and update the model parameters given a model prediction and observations, respectively. Listings 8 - 13 present the implementation of the Bayesian Neural Network and associated Langevin MCMC sampling scheme. Note that we implement the Bayesian Neural Network via the MCMC sampler class to sample (train) the weights and biases of the Neural Network class.

Next, we implement the model for a single hidden layer neural network with multiple input neurons and multiple output neurons (for binary and multi-class classification and multi-output regression). Listing 8 defines the Neural Network class with the *constructor* function (*init*) which defines the network topology, in terms of the number of input, hidden and output neurons along with the learning rate. These values are passed by the calling function. Next, we compute the total number of parameters by Line 17. In Line 22, we initialize the network by calling the function (*initialise_network*) where we initialize (create) matrices for the weights from the input-hidden, and hidden-output layer, along with the vectors for their biases (Lines 33-49). Line 51 gives the *evaluate_proposal* function that takes the input data and proposed parameters (*x_data* and *theta*) and *returns* the prediction (*fx*) in Line 62. We feature the sigmoid transfer (activation) function in Line 64.

Listing 9 lists the rest of the functions from the Neural Network class, where *forward_pass* propagates the information forward from input - hidden layer and then hidden to output layer using a *dot product* (Lines 12 and 16) and the *returns* the output layer (Line 17). The *backward_pass* function begins by computing the gradients (delta) at the output layer (Line 27) and hidden layer (Line 33). Lines 40–44 update the weights in the hidden and output later using their respective gradients.

In a conventional backpropagation algorithm implementation, basically the forward and backward pass functions will be called in an iterative loop that will call these functions until the maximum number of epochs, or a given training

https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial/blob/main/04-Bayesian-Neural-Network.ipynb

```python
class NeuralNetwork:
    '''
    Neural Network model with a single hidden layer and a single output (y)
    '''
    def __init__(self, layer_sizes,learning_rate=0.01):
        '''  Initialize the model
        Input:
            - layer_sizes (input, hidden, output): array specifying the number of
            nodes in each layer
            - learning_rate: learning rate for the gradient update
        '''
        # Initial values of model parameters
        self.input_num = layer_sizes[0]
        self.hidden_num = layer_sizes[1]
        self.output_num = layer_sizes[2]
        # total number of parameters from weights and biases
        self.n_params = (self.input_num * self.hidden_num) + (self.hidden_num * self.output_num) +\
        self.hidden_num + self.output_num
        # learning params
        self.lrate = learning_rate
        # Initialize network structure
        self.initialise_network()
        # functions defined above - this is poor practice, but done for readability
        # and clarity
        self.forward_pass = MethodType(forward_pass, self)
        self.backward_pass = MethodType(backward_pass, self)

    def initialise_network(self):
        '''
        Initialize network structure - weights and biases for the hidden layer and output layer
        '''
        # hidden layer
        self.l1_weights = np.random.normal(
            loc=0, scale=1/np.sqrt(self.input_num),
            size=(self.input_num, self.hidden_num))
        self.l1_biases = np.random.normal(
            loc=0, scale=1/np.sqrt(self.hidden_num),
            size=(self.hidden_num,))
        # placeholder for storing the hidden layer values
        self.l1_output = np.zeros((1, self.hidden_num))
        # output layer
        self.l2_weights = np.random.normal(
            loc=0, scale=1/np.sqrt(self.hidden_num),
            size=(self.hidden_num, self.output_num))
        self.l2_biases = np.random.normal(
            loc=0, scale=1/np.sqrt(self.hidden_num),
            size=(self.output_num,))
        # placeholder for storing the model outputs
        self.l2_output = np.zeros((1, self.output_num))

    def evaluate_proposal(self, x_data, theta):
        '''
        A helper function to take the input data and proposed parameter sample and return the prediction
        Input:     data: (N x num_features) array of data
            theta: (w,v,b_h,b_o) vector of parameters with weights and biases
        '''
        self.decode(theta)  # method to decode w into W1, W2, B1, B2.
        size = x_data.shape[0]
        fx = np.zeros(size)
        for i in range(0, size):  # to see what fx is produced by your current weight update
            fx[i] = self.forward_pass(x_data[i,])
        return fx

    def sigmoid(self, x):
        # sigmoid activation function
        return 1 / (1 + np.exp(-x))

    def softmax(self, x):
        #softmax function
        prob = np.exp(x) / np.sum(np.exp(x))
        return prob
```

Listing 8. Python implementation of the Neural Network class.

(validation) error, has been reached. However, in our case, we are using the Langevin MCMC sampler to train the neural network model; hence, we have additional helper functions (Listing 10) to ensure that the MCMC sampler class gets the information as needed. Essentially, from the MCMC class (Listing 11), the sampler function (Listing 13) calls

```
# NN prediction
def forward_pass(self, X):
    '''
    Take an input X and return the output of the network
    Input:
        - X: (N x num_features) array of input data
    Output:
        - self.l2_output: (N) array of output data f(x) which can be
        compared to observations (Y)
    '''
    # Hidden layer
    l1_z = np.dot(X, self.l1_weights) + self.l1_biases
    self.l1_output = self.sigmoid(l1_z) # activation function g(.)
    # Output layer
    l2_z = np.dot(self.l1_output, self.l2_weights) + self.l2_biases
    self.l2_output = self.sigmoid(l2_z)
    return self.l2_output

def backward_pass(self, X, Y):
    '''
    Compute the gradients using a backward pass and undertake Langevin-gradient updating of parameters
    Input:
        - X: (N x num_features) array of input data
        - Y: (N) array of target data
    '''
    # dE/dtheta
    l2_delta = (Y - self.l2_output) * (self.l2_output * (1 - self.l2_output))
    l2_weights_delta = np.outer(
        self.l1_output,
        l2_delta
    )
    # backprop of l2_delta and same as above
    l1_delta = np.dot(l2_delta,self.l2_weights.T) * (self.l1_output * (1 - self.l1_output))
    l1_weights_delta = np.outer(
        X,
        l1_delta
    )

    # update for output layer
    self.l2_weights += self.lrate * l2_weights_delta
    self.l2_biases += self.lrate * l2_delta
    # update for hidden layer
    self.l1_weights += self.lrate * l1_weights_delta
    self.l1_biases += self.lrate * l1_delta
```

Listing 9. Python implementation of Neural Network forward and backward passes.

the respective functions to evaluate the likelihood and the prior (Listing 12). In the case of computing the likelihood, the *evaluate_proposal* function in Listing 8 calls the decode function in Listing 10 to insert the values of the proposal (*theta*) into the weight matrices and bias vectors of the model defined in the *NeuralNetwork* class of Listing 8.

Next, we move to Listing 11 that implements the Langevin MCMC sampler for classification problems, as given in the notebook of Github repository. We note that we also provide the implementation for regression/prediction problems in the notebook. Furthermore, we also provide Python code implementation that features both classification and regression problems in the repository.

https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial/blob/main/04a-Bayesian-Neural-Network-Classification.ipynb

https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial/blob/main/04-Bayesian-Neural-Network.ipynb

https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial/tree/main/code

In Listing 11, we define the MCMC class with number of samples (*n_samples*), the burnin period (*n_burnin*), along with the training (*x_data* and *y_data*) and test datasets (*x_test* and *y_test*). Lines 12-14 initializes the hyperparameters, such as the *step_size* of the random-walk on *theta* and the *sigma_squared* that defined the spread of the Gaussian prior for the weights and biases. Lines 17-21 define the neural network model, the use of Langevin-gradients, the probability (*l_prob*) for using it, and the total number of weights and biases (*theta_size*). Next comes the storage of the parameters that are samples (Lines 24-25). Line 27 defines the function for *model_draws* - this is used post-sampling as a means to test the trained model. Line 50 defines the classification prediction accuracy, note that other error metrics can also be added.

Listing 12 defines the multinomial Log-likelihood and prior for classification problems. Line 1 implements the multinomial log-likelihood function that uses the parameters and data (Equation 33). Line 22 implements the log-prior function that uses the proposals (*theta*) and the user-defined

```
1      def langevin_gradient(self, x_data, y_data, theta, depth):
2          '''
3          Compute the Langevin-gradient proposal distribution
4          Input:
5              - x_data: (N x num_features) array of input data
6              - y_data: (N) array of target data
7              - theta: (w,v,b_h,b_o) vector of proposed parameters.
8              - depth: SGD depth
9          Output:
10             - theta_updated: Updated parameter proposal
11         '''
12         self.decode(theta)  # method to decode w into W1, W2, B1, B2.
13         size = x_data.shape[0]
14         # Update the parameters based on LG
15         for _ in range(0, depth):
16             for ii in range(0, size):
17                 self.forward_pass(x_data[ii,])
18                 self.backward_pass(x_data[ii,], y_data[ii])
19         theta_updated = self.encode()
20         return  theta_updated
21
22     def encode(self):
23         '''
24         Encode the model parameters into a vector
25         theta: vector of parameters.
26         '''
27         w1 = self.l1_weights.ravel()
28         w2 = self.l2_weights.ravel()
29         theta = np.concatenate([w1, w2, self.l1_biases, self.l2_biases])
30         return theta
31
32     def decode(self, theta):
33         '''
34         Decode the model parameters from a vector
35         theta: vector of parameters.
36         '''
37         w_layer1size = self.input_num * self.hidden_num
38         w_layer2size = self.hidden_num * self.output_num
39         w_layer1 = theta[0:w_layer1size]
40         self.l1_weights = np.reshape(w_layer1, (self.input_num, self.hidden_num))
41
42         w_layer2 = theta[w_layer1size:w_layer1size + w_layer2size]
43         self.l2_weights = np.reshape(w_layer2, (self.hidden_num, self.output_num))
44         self.l1_biases = theta[w_layer1size + w_layer2size:w_layer1size + w_layer2size + self.hidden_num]
45         self.l2_biases = theta[w_layer1size + w_layer2size + self.hidden_num:w_layer1size + w_layer2size
       + self.hidden_num + self.output_num]
```

Listing 10. Langevin-gradient functions in the Neural Network class.

variance (*sigma_squared*) for the Gaussian prior. Note that in the case of regression and prediction problems, the log-likelihood and log-prior are similar to the Bayesian linear regression (Listing 4) with the omission of terms related to τ^2.

In Listing 14, we begin sampling by first initializing variables that track the number of accepted proposals and how many times Langevin-gradients are utilized, this is just for analysis. In Line 4, we begin the sampling using a *for loop* that begins with 1 and ends with the number of samples. In Line 6, we propose the new values for the parameters (*theta*) using random-walk proposal distribution centered at the mean of 0 and given spread (*step_size*), which needs to be experimentally determined in trial experiments. Then, we decide if we wish to use the Langevin-gradients or random-walk proposal distribution (Lines 7-8). Lines 9-20 implement the Langevin-gradients where we get one-step gradients from the neural network model. Therefore, we need to run the *forward-pass* and a *backward-pass* functions using the new sets of the current weights and biases. Then, we need to obtain the gradients of the output and hidden layers of the network, and concatenate to return these as a vector (Lines 15-20) of Listing 10.

In Listing 14, we then use the gradient (*theta_grad*) as the center for the normal distribution to draw and add Gaussian noise to the gradients (Line 10). We again obtain the gradients, this is merely for the detailed balance condition. In Line 11, we get the gradients again, but this time we use the new values of theta (*theta_proposal*) that we earlier obtained in Line 6. As given in Equation 42, in the case when the proposals are not symmetric (i.e., Langevin-gradients), we need to get the q-ratio (Line 19 *diff_prop*). In the log-scale, this is obtained by the difference in the current *theta* (first) and the new *theta_proposal* (second) to account for the detailed balance condition as shown in Line 19. In order to obtain the q-ratio, we need to further evaluate the old and the new proposals using the multivariate normal distribution and for numerical stability. However, we need to have a simplified implementation for the multivariate distribution

```python

class MCMC:
    def __init__(self, model, n_samples, n_burnin, x_data, y_data, x_test, y_test):
        self.n_samples = n_samples # number of MCMC samples
        self.n_burnin = n_burnin # number of burn-in samples
        self.x_data = x_data # (N x num_features)
        self.y_data = y_data # (N x 1)
        self.x_test = x_test # (Nt x num_features)
        self.y_test = y_test # (Nt x 1)

        # MCMC parameters - defines how much variation you need in changes to theta, tau
        self.step_theta = 0.025;
        # Hyperpriors
        self.sigma_squared = 25

        # initisalise the neural network model class
        self.model = model
        self.use_langevin_gradients = True
        self.sgd_depth = 1
        self.l_prob = 0.5 # likelihood prob
        self.theta_size = self.model.n_params # weights for each feature and a bias term

        # store output
        self.pos_theta = None
        self.rmse_data = None

    def model_draws(self, num_draws = 10, verbose=False):
        ''' Calculate the output of the network from draws of the posterior distribution
        Input: num_draws: number of draws,   verbose: if True, print the details of each draw
        Output: pred_y: (num_draws x N) ouptut of the NN for each draw'''
        accuracy = np.zeros(num_draws)
        pred_y = np.zeros((num_draws, self.x_data.shape[0]))
        sim_y = np.zeros((num_draws, self.x_data.shape[0]))

        for ii in range(num_draws):
            theta_drawn = np.random.normal(self.pos_theta.mean(axis=0), self.pos_theta.std(axis=0), self.theta_size)
            [likelihood_proposal, pred_y[ii,], sim_y[ii,], accuracy[ii]] = self.likelihood_function(
                theta_drawn
            )
            if verbose:
                print(
                    'Draw {} - accuracy: {:.3f}. Theta: {}'.format(
                        ii, accuracy[ii], theta_drawn
                    )
                )
        return pred_y, sim_y

    # Additional error metric
    @staticmethod
    def accuracy(predictions, targets):
        '''
        Additional error metric - accuracy
        '''
        count = (predictions == targets).sum()
        return 100 * (count / predictions.shape[0])
```

Listing 11. **Bayesian neural network using MCMC sampler for classification problems.**

given a large set of weights and biases (*theta*). Since we are operating in the log-scale, we can further simplify the multivariate normal distribution as shown in Lines 14-19. Finally, Line 23 implements the case when random-walk proposal distribution would be used, note that in Line 22, the *diff_prop* is 0. This accounts for the detailed balance condition, since the proposals are naturally symmetric, in the case of random-walk proposal distribution (Line 23).

Next, we compute the log-likelihood in Listing 14 (Line 27) and the error metrics for the test dataset (Line 29). We note that this is a classification problem, and hence the classification accuracy (Listing 12 - Line 13) is reported using the log-likelihood function. We determine the Metropolis-Hastings (MH) acceptance rate using Lines 34-35. In Lines 32 and 33, we get the difference (ratio) for the proposed likelihood and the current likelihood, and the ratio for the prior with the current and proposed value of the prior. We utilize these to get the MH probability in Line 34, which also utilizes the difference (ratio) of proposed and current proposals, obtained either from Line 19 or Line 22. Finally, we either accept (Lines 39-45) or reject (Line 48) the proposal by comparing the MH probability with a random value obtained in Line 35. In the case if the proposal is accepted, the proposed values of theta along with prior and

```python
def likelihood_function(self, theta, test=False):
    ''' Calculate the multinomial log-likelihood of the data given the parameters and model
    Input:  theta: vector of parameters
    Output:  log_likelihood: log likelihood of the data given the parameters '''
    if test:
        x_data = self.x_test
        y_data = self.y_test
    else:
        x_data = self.x_data
        y_data = self.y_data
    model_prediction, probs = self.model.evaluate_proposal(x_data, theta)
    model_simulation = model_prediction
    accuracy = self.accuracy(model_prediction, y_data) #Accuracy error metric
    # now calculate the log-likelihood
    log_likelihood = 0
    for ii in np.arange(x_data.shape[0]):
        for jj in np.arange(self.model.output_num):
            if y_data[ii] == jj:
                log_likelihood += np.log(probs[ii,jj])
    return [log_likelihood, model_prediction, model_simulation, accuracy]

def prior(self, sigma_squared, theta):
    '''
    Calculate the prior of the parameters
    Input: sigma_squared - variance of normal prior for theta
    Output: log_prior
    '''
    n_params = self.theta_size # number of parameters in model
    part1 = -1 * (n_params / 2) * np.log(sigma_squared)
    part2 = 1 / (2 * sigma_squared) * (sum(np.square(theta)))
    log_prior = part1 - part2
    return log_prior
```

Listing 12. Multinomial Log-likelihood and prior for classification problems (Continued from Listing 11).

```python
 # MCMC sampler
def sampler(self):
    '''
    Run the sampler for a defined Neural Network model
    '''
    # define empty arrays to store the sampled posterior values
    # posterior of all weights and bias over all samples
    pos_theta = np.ones((self.n_samples, self.theta_size))

    # record output f(x) over all samples
    pred_y = np.zeros((self.n_samples, self.x_data.shape[0]))
    # record simulated values f(x) + error over all samples
    sim_y = np.zeros((self.n_samples, self.x_data.shape[0]))
    # record the RMSE of each sample
    accuracy_data = np.zeros(self.n_samples)
    # now for test
    test_pred_y = np.ones((self.n_samples, self.x_test.shape[0]))
    test_sim_y = np.ones((self.n_samples, self.x_test.shape[0]))
    test_accuracy_data = np.zeros(self.n_samples)

    ## Initialisation
    # initialise theta
    theta = np.random.randn(self.theta_size)
    # make initial prediction
    pred_y[0,], _ = self.model.evaluate_proposal(self.x_data, theta)

    # Hyperparameters of priors - considered by looking at distribution of  similar trained  models - i.e
     distribution of weights and bias
    sigma_squared = self.sigma_squared

    # calculate the prior
    prior_val = self.prior(sigma_squared, theta)
    # calculate the likelihood considering observations
    [likelihood, pred_y[0,], sim_y[0,], accuracy_data[0]] = self.likelihood_function(theta)
```

Listing 13. Implementation of MCMC sampler function (Continued from Listing 12).

likelihood, become the current value in the chain. In the case if it is rejected, then the chain maintains its last accepted value as the current value (Line 48). We remove the burn-in portion and store the posterior (Lines 53-57). Finally, we return the dictionary of the data that features the posterior and predictions using the Pandas library.

```python
    n_accept = 0
    n_langevin = 0
    # Run the MCMC sample for n_samples
    for ii in tqdm(np.arange(1,self.n_samples)):
        # Sample new values for theta
        theta_proposal = theta + np.random.normal(0, self.step_theta, self.theta_size)
        lx = np.random.uniform(0,1,1)
        if (self.use_langevin_gradients is True) and (lx < self.l_prob):
            theta_gd = self.model.langevin_gradient(self.x_data, self.y_data, theta.copy(), self.
    sgd_depth)
            theta_proposal = np.random.normal(theta_gd, self.step_theta, self.theta_size)
            theta_proposal_gd = self.model.langevin_gradient(self.x_data, self.y_data, theta_proposal.
    copy(), self.sgd_depth)
            # for numerical reasons, we will provide a simplified implementation that simplifies
            # the MVN of the proposal distribution
            wc_delta = (theta - theta_proposal_gd)
            wp_delta = (theta_proposal - theta_gd)
            sigma_sq = self.step_theta
            first = -0.5 * np.sum(wc_delta * wc_delta) / sigma_sq
            second = -0.5 * np.sum(wp_delta * wp_delta) / sigma_sq
            diff_prop =  first - second
            n_langevin += 1
        else:
            diff_prop = 0
            theta_proposal = np.random.normal(theta, self.step_theta, self.theta_size)
        # calculate the prior
        prior_proposal = self.prior(sigma_squared, theta_proposal)  # takes care of the gradients
        # calculate the likelihood considering observations
        [likelihood_proposal, pred_y[ii,], sim_y[ii,], accuracy_data[ii]] = self.likelihood_function(
    theta_proposal)
        # calculate the test likelihood
        [_, test_pred_y[ii,], test_sim_y[ii,], test_accuracy_data[ii]] = self.likelihood_function(
            theta_proposal, test=True)
        # since we using log scale: based on https://www.rapidtables.com/math/algebra/Logarithm.html
        diff_likelihood = likelihood_proposal - likelihood
        diff_prior = prior_proposal - prior_val
        mh_prob = min(1, np.exp(diff_likelihood + diff_prior + diff_prop))
        u = np.random.uniform(0, 1)
        # Accept/reject
        if u < mh_prob:
            # Update position
            n_accept += 1
            # update
            likelihood = likelihood_proposal
            prior_val = prior_proposal
            theta = theta_proposal
            # and store
            pos_theta[ii,] = theta_proposal
        else:
            # store
            pos_theta[ii,] = pos_theta[ii-1,]
    # print the % of times the proposal was accepted
    accept_ratio = (n_accept / self.n_samples) * 100
    print('{:.3}% were acepted'.format(accept_ratio))
    # store the posterior of theta
    self.pos_theta = pos_theta[self.n_burnin:, ]
    # Create a pandas dataframe to store the posterior samples of theta
    results_dict = {'w{}'.format(_): self.pos_theta[:, _].squeeze() for _ in range(self.theta_size-2)}
    results_dict['b0'] = self.pos_theta[:, self.theta_size-2].squeeze()
    results_dict['b1'] = self.pos_theta[:, self.theta_size-1].squeeze()
    results_df = pd.DataFrame.from_dict(results_dict)
    return results_df
```

Listing 14. **Begin with sampling loop (Continued from Listing 13).**

VI. RESULTS

A. DATA

We use the Sunspot time series data and Abalone datasets for regression problems. The Abalone dataset provides the ring age for Abalone based on eight features that represent physical properties such as length, width, and weight and associated target feature, i.e., the ring age. We note that determining the age of Abalone is difficult, as it requires cutting the shell and counting the number of rings using a microscope. However, other physical measurements can be used to predict the age and a model can be developed to use the physical features to determine the ring age. Sunspots are regions of reduced surface temperature in the Sun's

https://www.sidc.be/silso/datafiles
https://archive.ics.uci.edu/ml/datasets/abalone

photosphere caused by concentrations of the magnetic field flux, and appears as spots darker than the surrounding areas. The Sunspot cycles are about every eleven years and over the solar cycle, the number of Sunspot changes more rapidly. Sunspot activities are monitored since they have an impact on Earth's climate and weather. We obtain the Abalone dataset from the University of California (UCI) Machine Learning Repository and keep a processed version of all the datasets in our repository.

In the case of the Sunspot time series, we define a one-step ahead prediction problem and hence use one output neuron. We process the Sunspot dataset (univariate time series) using Taken's embedding theorem [124] to construct a state-space vector, also known as *data windowing*. This is essentially using a sliding window approach of size D overlapping T time lags. The window size D determines the number of input neurons in the Bayesian neural network and Bayesian linear model. We use $D = 4$ and $T = 2$ for data reconstruction as these values have given good performance in our previous works [62].

We also obtain datasets for classification problems from the same repository that features a large number of datasets for classification problems. We selected the Iris classification dataset that contains 4 features (sepal length, sepal width, petal length, petal with) of three types of Iris flower species, featuring 50 instances for each case. This dataset is one of the most prominent datasets used for machine learning. We also selected the Ionosphere dataset that features a binary classification task with 351 instances. It has 34 continuous features and the task is to filter the radio signals as "good" or "bad".

B. EXPERIMENT SETTING: HYPERPARAMETERS

In the Bayesian linear model and neural network, we choose the number of samples to be 25,000 for all problems, distributed across 5 chains and excluding 50% burn-in. In the Bayesian linear model, we choose the learning rate $r = 0.1$, and the step sizes for $\theta = 0.02$ and $\tau = 0.01$, respectively. Additionally, for the Gaussian prior distribution, we choose the parameters $\sigma^2 = 5, \nu_1 = 0$ and $\nu_2 = 0$, respectively. In the Bayesian neural network models, we choose the learning rate $r = 0.01$, and the step sizes for $\theta = 0.025$ and $\tau = 0.2$, respectively. In the Gaussian prior distribution, we choose the hyperparameter $\sigma^2 = 25$ determined from examining trained neural network models for similar problems. In the case of regression, we use inverse-Gamma prior for τ^2, and hence use hyperparameters for this prior, $\nu_1 = 0$ and $\nu_2 = 0$, respectively. We also use a burn-in rate of 0.5 for both the Bayesian linear and the Bayesian neural network models. In the case of Bayesian neural networks, we apply Langevin-gradients at a rate of 0.5.

https://archive-beta.ics.uci.edu/about

https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial/tree/main/data

https://archive.ics.uci.edu/ml/datasets/iris

https://archive.ics.uci.edu/ml/datasets/ionosphere

C. RESULTS: REGRESSION AND PREDICTION

We first present the results of Bayesian regression with the Sunspot (time series prediction) and Abalone (regression) datasets. We evaluate the model performance using the *root mean squared error* (RMSE), which is a standard metric for time series prediction and regression problems. We present the results obtained by the Bayesian linear model and Bayesian neural network model for regression problems in Table 1. Figures 8 and 9 present the prediction plots (observed, modelled and 95 % credible interval) that show a comparison between Bayesian linear model and neural network models for the fixed training (Panels a and b) and test (Panels c and d) datasets. Note that we report the results of a single experimental run, which is not conventional when considering the frequentest approach of reporting results featuring multiple experimental runs. We note that Panels (a) and (c) show the time series prediction (x-axis represents timestep number). Panels (b) and (d) present a scatter-plot of the change from timestep $t - 1$ to t in the observed (ΔY observed) and predicted values (ΔY modelled). This gives an indication of model's ability to predict change at each timestep with a skill better than persistence (as observed Y_{t-1} is given as an input to the model, a model predicting $y_t = Y_{t-1}$ could have a low RMSE). Thus, we note that the RMSE does not assess the skill (capability) of the model above the observed Y_{t-1} (given these models are conducting one step-ahead prediction); however, given the analysis using Panels (b) and (d), the performance of the two models can be further compared using RMSE.

In Table 1, we observe that the Bayesian neural network performs better for the Sunspot time series prediction problem, as it achieves a better accuracy (lower RMSE) on both the training and testing set. This can also be seen in Figures 8 and 9.

In the case of the Abalone - regression problem, Table 1 shows that both models obtain similar classification performance, but the Bayesian neural network has better test performance. However, we note that in both problems, Bayesian neural networks have a much lower acceptance rate; we prefer roughly a 23 % acceptance rate [125] that implies that the posterior distribution has been effectively sampled. However, we also note that such MCMC sampling acceptance rates are typically based on statistical and linear models, which may not apply to Bayesian neural networks. Therefore, more research needs to be done to determine a good acceptance rate that aligns with convergence and *ergodicity* [96].

D. RESULTS: CLASSIFICATION PROBLEMS

Next, we move to the case of classification problems using Bayesian neural networks and Bayesian linear models.

Table 2 presents results for the classification problems in the Iris and Ionosphere datasets. We notice that both models have similar test and training classification performance for the Iris classification problem, and Bayesian neural

TABLE 1. Results using Bayesian linear model and Bayesian neural networks via MCMC sampler for the Abalone (regression) and the Sunspot (prediction) problems. The results show the RMSE mean and standard deviation (in brackets) for the train and test datasets, respectively.

Method	Problem	Train RMSE	Test RMSE	Accept. rate
Bayesian linear model	Sunspot	0.025 (0.013)	0.022 (0.012)	13.5%
Bayesian neural network	Sunspot	0.027 (0.007)	0.026 (0.007)	7.4%
Bayesian linear model	Abalone	0.085 (0.005)	0.086 (0.005)	5.8%
Bayesian neural network	Abalone	0.080 (0.002)	0.080 (0.002)	3.8%

TABLE 2. Classification accuracy with Bayesian linear model and Bayesian neural networks via MCMC. The results show the accuracy mean and standard deviation (in brackets) for the train and test datasets.

Method	Problem	Train Accuracy	Test Accuracy	Accept. rate
Bayesian linear model	Iris	90.392% (2.832)	90.844% (3.039)	83.5%
Bayesian neural network	Iris	97.377% (0.655)	98.116% (1.657)	97.0%
Bayesian linear model	Ionosphere	89.060% (1.335)	85.316% (2.390)	58.8%
Bayesian neural network	Ionosphere	99.632% (0.356)	92.668% (1.890)	94.5%

networks give better results for the test dataset for the Ionosphere problem. The acceptance rate is much higher for Bayesian neural networks in the case of classification when compared to regression/prediction problems. In the case of regression (Table 1), the Bayesian linear model has a better acceptance rate, but that is also not suitable when compared to the acceptance rate in the literature for Bayesian linear models. This suggests that we need better tuning of the hyperparameters, especially when Langevin gradients are used.

VII. CONVERGENCE DIAGNOSIS

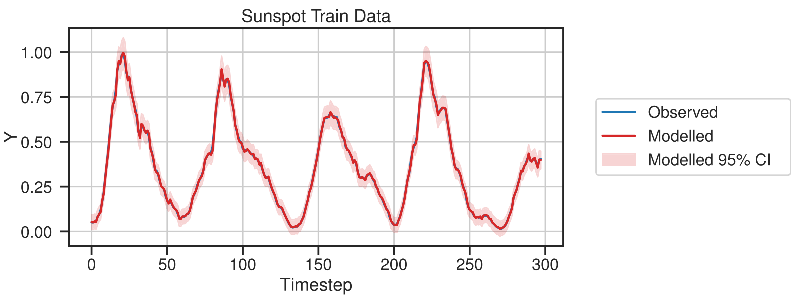
It is important to ensure that the MCMC sampling is adequately exploring the parameter space and constructing an accurate representation of the posterior distribution. One method of monitoring the performance of the adopted MCMC sampler is to examine *convergence diagnostics* that assess/monitor the extent to which the Markov chains have become a stationary distribution. The Gelman-Rubin (GR) convergence diagnostic [126] is developed by sampling from multiple MCMC chains, whereby the variance of each chain is assessed independently (within-chain variance) and then compared to the variance between the multiple chains (between-chain variance) for each parameter. A large difference between these two variances would indicate that the chains have not converged on the same stationary distribution.

In our case, we run several independent experiments and compare the MCMC chains using a modified Gelman-Rubin convergence diagnostic presented by Vehtari et al. [127]. It is beyond the scope of this publication to provide a detailed mathematical description of the convergence diagnostics, the reader can refer to [127] for a full description. A useful package for Bayesian model diagnostics, which contains an implementation of this modified diagnostic is *arviz* [128]. We refer to the modified Gelman-Rubin diagnostic as $\hat{R}$, where values close to 1 indicate convergence. In Listing 15, we present the code to prepare the MCMC sampler outputs for convergence diagnostic by *arviz*.

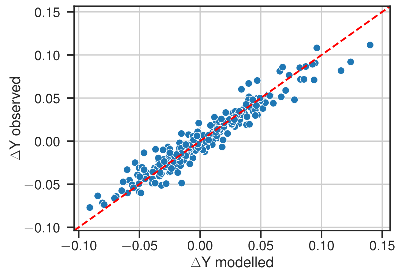
A. RESULTS FOR CONVERGENCE DIAGNOSIS

We show results for the modified Gelman-Rubin diagnostic for Bayesian linear and Bayesian neural network models for each of the four datasets. We test five chains for each model, each with 5,000 samples of weights (excluding 50% burn-in samples) and computing the $\hat{R}$ values for each parameter. We also provide an additional example "Linear+" for the linear models where each of the 5 chains has 50,000 samples (excluding 50% burn-in samples). The basic sampler presented here is not state of the art in terms of sampling efficiency (see Section IX for further discussion), and for some of the problems (particularly those where parameters are difficult to identify) it may take a large number of samples to converge. The additional example (Linear+) demonstrates that the convergence is improving as the number of samples grows. Figure 10 shows the distribution of the $\hat{R}$ values, and we observe that the $\hat{R}$ values of the weights for the Bayesian linear regression model are much smaller than Bayesian neural network. We can observe that based on the Gelman-Rubin diagnostics, the Bayesian neural

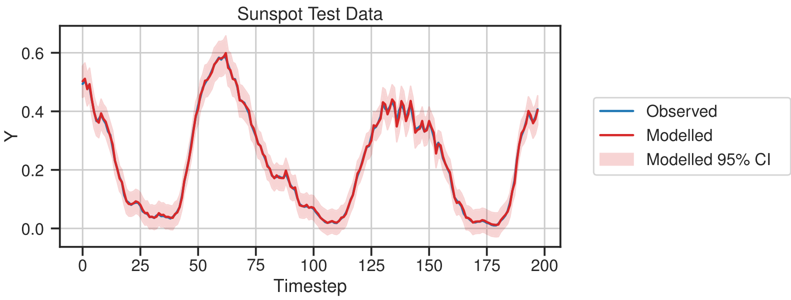
https://python.arviz.org/en/stable/api/generated/arviz.rhat.html



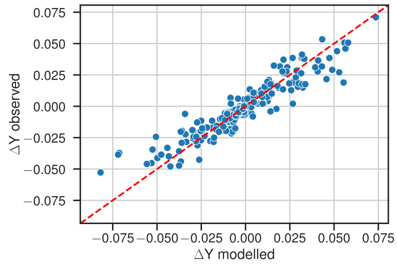
(a) Predictions and actual train dataset for Sunspot



(b) Predictions and actual of ΔY train dataset for Sunspot prediction.

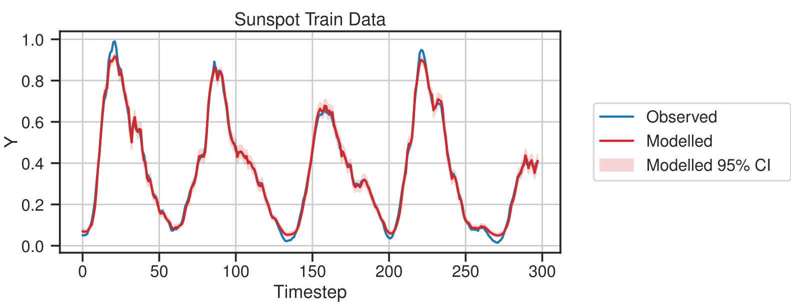


(c) Predictions and actual test dataset for Sunspot

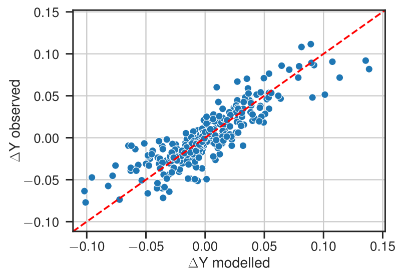


(d) Predictions and actual of ΔY test dataset for Sunspot prediction.

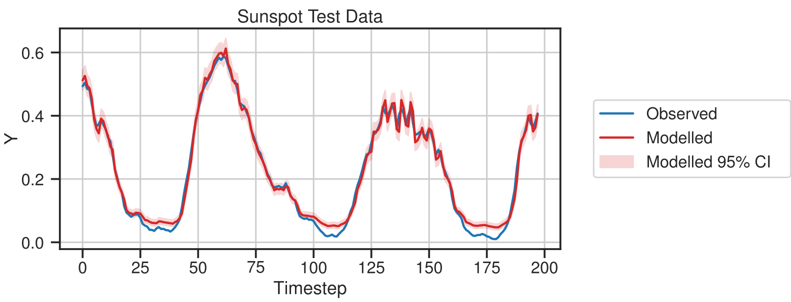
FIGURE 8. **Bayesian linear regression model - Sunspot dataset.**



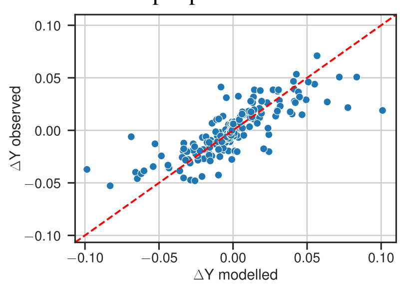
(a) Predictions and actual train dataset for Sunspot



(b) Predictions and actual of ΔY train dataset for Sunspot prediction.



(c) Predictions and actual test dataset for Sunspot



(d) Predictions and actual of ΔY test dataset for Sunspot prediction.

FIGURE 9. **Bayesian neural network model - Sunspot dataset.**

network shows poor convergence. The additional samples in the ''Linear+'' case improve convergence, in particular for the Abalone, Iris, and Ionosphere cases.

By closely examining each weight individually, we observe that the problem of non-convergence mainly arises from the multi-modality of the posterior. In Figure 11, we look at samples from a single chain of 20,000 samples excluding 20% burn-in. In Figure 11-Panel (a) and 11-Panel (b), we present a visualization for a selected weight from the Bayesian linear model. In Figure 11-Panel (c) and 11-Panel (d), we present a

```python
import arviz as az
# Sample a second chain as more than one chain is required to generate the Rhat diagnostic
# Run second chain using a different seed
np.random.seed(2)
# setup second chain
mcmc_chain2 = MCMC(n_samples, burn_in, x_data, y_data, x_test, y_test)
# Run the sampler
results_chain2 = mcmc_chain2.sampler()

# Now combine these two chains by stacking and convert to arviz dataset
# To convert to arviz, convert the pandas dataframe of samples into a dictionary of lists, stack and ingest into arviz
res_dict = results.to_dict(orient='list')
res_dict_chain2 = results_chain2.to_dict(orient='list')
# stack the chains looping through each parameter and ingest into arviz
az_results = az.from_dict(
    {par: np.vstack([res_dict[par], res_dict_chain2[par]]) for par in res_dict}
)

# now we can use arviz to obtain summary statistics with Rhat ("r_hat") for each parameter.
az.summary(az_results)
```

Listing 15. Convergence diagnostics using the *arviz* python package that could be run after Listing 7.

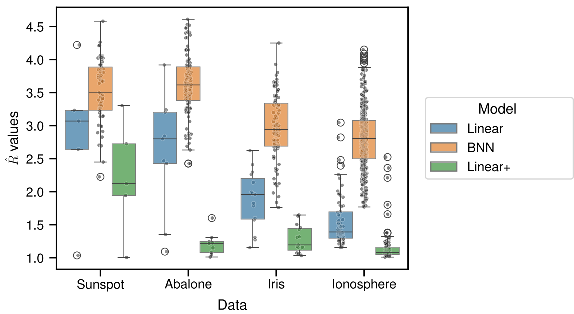


FIGURE 10. Distribution of $\hat{R}$ values for our Bayesian linear model (Linear), Bayesian neural network (BNN) and a linear model with 10 times more samples (Linear+). The scattered points show the underlying data summarized in the box plots. The "Linear+" case shows the ability of the sampler to converge as more samples are taken.

visualization for a selected weight from the Bayesian neural network model. We observe potential multimodal distributions in both cases, with a high degree of auto-correlation and poor convergence. To examine the impact of longer MCMC chains in achieving convergence, we ran an additional test by taking 400,000 samples excluding 20% burn-in, and then thinning the chain by a factor of 50 for visualization. Thinning is the process of reducing the memory burden of the chain, particularly where samples may be auto-correlated [129]. We can also use thinning to visualize long chains with many samples. In our case, we implement thinning by retaining every 50th sample of the chain. We present these results for the Iris dataset in Figure 12. We can see that the chains exhibit more desirable properties, particularly in the case of the linear model (Figure 12- Panels (a) and (b). We can see in Figure 12 - Panels (c) and (d), that the Bayesian neural network exhibits a multimodel posterior for this parameter.

Furthermore, other approaches for convergence diagnosis such as autocorrection analysis can also be implemented with packages such as the *integrated autocorrelation time* [130] in *Emcee* [131]. We refer to readers for a comprehensive review of MCMC convergence diagnostics given by [132], [133], and [134].

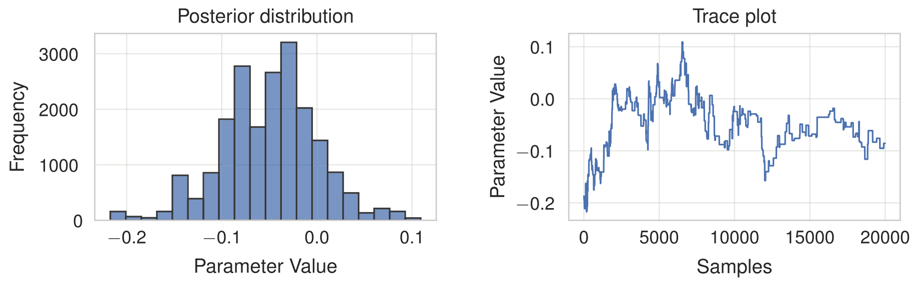
VIII. MCMC PACKAGES

We note that there are avenues to further improve the sampling efficiency, we utilized Langevin MCMC but other gradient-based approaches such as Hamiltonian MCMC (HMC) [60] also exist. It is worthwhile to evaluate the performance of Langevin MCMC against other advanced gradient-based sampling algorithms (e.g., No-U-turn Sampler (NUTS) [135]) to assess convergence properties in cases of multimodal posterior distributions, such as in Bayesian neural networks. We note that implementation of HMC and NUTS exist in probabilistic programming libraries such as PyMC [136] and Stan [137], [138]. We implemented an additional notebook using the NumPyro [139] probabilistic programming library to perform an equivalent Bayesian linear regression (Listing 3-7) as an example.

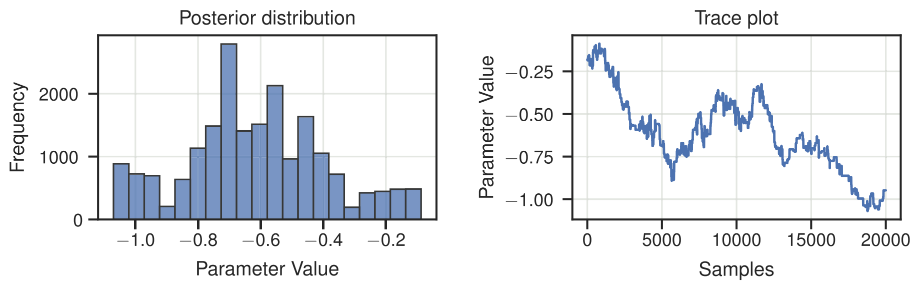
IX. DISCUSSION

We presented a Python tutorial for Bayesian neural networks and Bayesian linear models using MCMC sampling. In general, we observed that the Bayesian neural network performs better than Bayesian linear regression in our selected problems (Tables 1 and 2), despite showing no poor convergence (Figure 10 and Figure 12-Panel (d)). This could be due to the challenge of sampling a relatively large number of weights and biases of Bayesian neural networks, which also have multimodal posterior distribution. Hence, we conclude that for the case of Bayesian neural networks, a poor performance in the Gelman-Rubin diagnostics does not necessarily imply a poor performance in prediction tasks. We revisit the principle of *equifinality* [140], [141]

https://emcee.readthedocs.io/en/stable/tutorials/autocorr/
https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial/blob/main/05-Linear-Model_NumPyro.ipynb

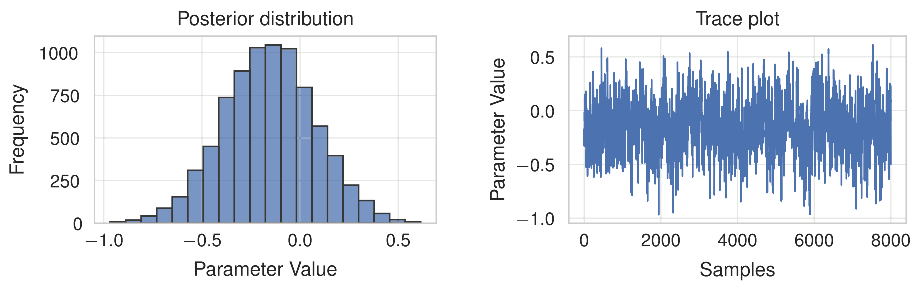


(a) Linear model - selected parameter posterior (b) Linear model - selected parameter trace-plot

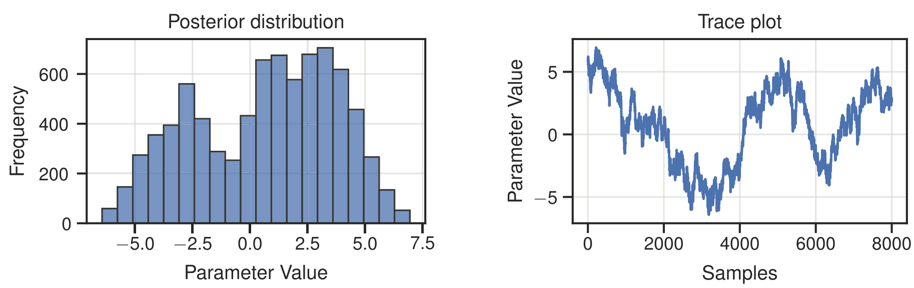


(c) Bayesian neural network - selected parameter (weight) posterior (d) Bayesian neural network - selected parameter (weight) trace-plot

FIGURE 11. Posterior and trace-plot for a parameter in each of the Bayesian linear and Bayesian neural network models - Sunspot data. In this example, 20,000 samples were taken excluding 20% burn-in with no thinning.



(a) Linear model - selected parameter posterior (b) Linear model - selected parameter trace-plot



(c) Bayesian neural network - selected parameter (weight) posterior (d) Bayesian neural network - selected parameter (weight) trace-plot

FIGURE 12. Posterior and trace-plot for a selected parameter in each of the Bayesian linear and Bayesian neural network models - Iris data. We took 400,000 samples (excluding 20% burn-in) with the chain thinned by a factor of 50 for visualisation.

which states that in open systems, a given end state can be reached by many potential means. In our case, the system is a neural network model and many solutions exist that represent a trained model displaying an accepted level of performance accuracy. However, we note that Bayesian models offer uncertainty quantification in predictions, and proper convergence is required. The original Gelman-Rubin diagnosis [126] motivated several enhancements for different types of problems [127], [133], [142], [143]; and we may need to develop a better diagnosis for Bayesian neural networks. Nonetheless, in our case comparing Bayesian logistic regression (converged) and Bayesian neural networks (not converged but achieved good accuracy), we can safely state that the Bayesian neural networks presented can only provide a means for uncertainty quantification, and not mature enough to qualify as a robust Bayesian model.

We also revisit the convergence issue in the case of the Bayesian linear model as shown in Panel (b) - Figure 11, which shows a multi-modal distribution that has not well converged. It may also be the case that certain features are not contributing much to the decision-making process (predictions) and the weights associated (coefficients) with those features may be difficult to sample. This is similar to the case of neural networks, where certain weight links are not needed and can be pruned. Pruning neural networks create compact models that also have the potential to get better generalization performance [144].

We note that we attained a much higher acceptance rate in Bayesian neural networks for classification problems (Table 2) when compared to regression problems (Table 2). We note that different likelihood functions are used for classification and regression (Multinomial and Gaussian likelihood) and we utilized the Langevin-gradient proposal distributions that accounted for the detailed balanced condition. Hence, we need to further fine-tune the hyperparameters associated with the proposal distribution to ensure we get a higher acceptance rate for the regression problems. Fine-turning the hyperparameters for the proposal distribution is a laborious task, which can be seen as a major limitation of MCMC sampling in large models such as Bayesian neural networks. Although 23 % acceptance rate [125] has been prominently used as a ''golden rule'', the optimal acceptance rate depends on the nature of the problem. The number of parameters, Langevin-based proposal distribution, and type of model would raise questions about the established acceptance rate [145]. Hence, more work needs to be done to establish what acceptance rates are appropriate for simple neural networks and deep learning models.

A way to address the issue of convergence would be to develop an ensemble of linear models that can compete with the accuracy of neural networks or deep learning models. In ensemble methods such as bagging and boosting, we can use linear models that have attained convergence as per Gelman-Rubin diagnosis and then combine the results of the ensemble using averaging and voting, as done in ensemble methods.

Our previous work has shown that despite the challenges, the combination of Langevin-gradients with parallel tempering MCMC [62], presents opportunities for sampling larger neural network architectures such as autoencoders and graph-based CNNs [69], [70]. The need to feature a robust methodology for uncertainty quantification in CNNs will make them more suitable for applications where uncertainty in decision-making poses major risks, such as medical image analysis [146] and human security [147]. CNNs have been considered for modelling temporal sequences, they have proven to be successful for time series classification [148], [149], and time series forecasting problems [150], [151], [152]. It has also been shown that one-dimensional CNNs provide better prediction performance than the conventional LSTM network for multistep ahead time series prediction problems [152]. Leveraging CNNs within a Bayesian framework can provide better uncertainty quantification in predictions and make them useful for cutting-edge real-world applications. We need a comprehensive evaluation of prominent gradient-based MCMC sampling methods for deep learning models such as CNNs, autoencoders, and LSTM networks.

We envision that this tutorial will enable statisticians, machine learning and deep learning experts to utilize MCMC sampling more effectively when developing new models and Bayesian frameworks for existing deep learning models. The tutorial has introduced basic concepts with code and provides an overview of challenges when it comes to the convergence of Bayesian neural networks. It can be further extended to utilize parallel computing via tempered MCMC [62], HMC for Bayesian neural networks, and Langevin MCMC and HMC for Bayesian deep learning.

CODE AND DATA

The code and data presented in this paper are available in the associated GitHub repository. This repository presents the implementations in separate Jupyter notebooks in the base directory, with sub-directories containing data, convenient functions and details of the environment setup.

ACKNOWLEDGMENT AND CONTRIBUTIONS

Rohitash Chandra contributed by conceptualization, coding and experiments, analysis, writing, and project supervision. Joshua Simmons contributed by coding, analysis, and writing.

The authors would like to thank Royce Chen from UNSW Sydney for the initial experiments and analysis.

REFERENCES

[1] G. Chamberlain and G. W. Imbens, ''Nonparametric applications of Bayesian inference,'' *J. Bus. Econ. Statist.*, vol. 21, no. 1, pp. 12–18, 2003.
[2] J. Geweke, ''Bayesian inference in econometric models using Monte Carlo integration,'' *Econometrica*, vol. 57, no. 6, p. 1317, Nov. 1989.
[3] G. E. Box and G. C. Tiao, *Bayesian Inference in Statistical Analysis*, vol. 40. Hoboken, NJ, USA: Wiley, 2011.
[4] R. Chandra, R. D. Müller, D. Azam, R. Deo, N. Butterworth, T. Salles, and S. Cripps, ''Multicore parallel tempering bayeslands for basin and landscape evolution,'' *Geochem., Geophys., Geosyst.*, vol. 20, no. 11, pp. 5082–5104, Nov. 2019.
[5] J. Pall, R. Chandra, D. Azam, T. Salles, J. M. Webster, R. Scalzo, and S. Cripps, ''Bayesreef: A Bayesian inference framework for modelling reef growth in response to environmental change and biological dynamics,'' *Environ. Model. Softw.*, vol. 125, Mar. 2020, Art. no. 104610.

https://github.com/sydney-machine-learning/Bayesianneuralnetworks-MCMC-tutorial

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.

[7] L. Martino and V. Elvira, "Metropolis sampling," in *StatsRef: Statistics Reference Online*. Hoboken, NJ, USA: Wiley, 2014, pp. 1–18.

[8] D. Mackay, "Probable networks and plausible predictions—A review of practical Bayesian methods for supervised neural networks," *Netw., Comput. Neural Syst.*, vol. 6, no. 3, pp. 469–505, Aug. 1995.

[9] R. M. Neal, *Bayesian Learning for Neural Networks*, vol. 118. Cham, Switzerland: Springer, 2012.

[10] D. F. Specht, "Probabilistic neural networks," *Neural Netw.*, vol. 3, no. 1, pp. 109–118, Jan. 1990.

[11] M. D. Richard and R. P. Lippmann, "Neural network classifiers estimate Bayesian a posteriori probabilities," *Neural Comput.*, vol. 3, no. 4, pp. 461–483, Dec. 1991.

[12] E. A. Wan, "Neural network classification: A Bayesian interpretation," *IEEE Trans. Neural Netw.*, vol. 1, no. 4, pp. 303–305, Dec. 1990.

[13] Z. Ghahramani, "Probabilistic machine learning and artificial intelligence," *Nature*, vol. 521, no. 7553, pp. 452–459, May 2015.

[14] S. Chib and E. Greenberg, "Understanding the metropolis-hastings algorithm," *Amer. Statistician*, vol. 49, no. 4, p. 327, Nov. 1995.

[15] C. P. Robert and G. Casella, "The metropolis—Hastings algorithm," in *Monte Carlo Statistical Methods*. Cham, Switzerland: Springer, 1999, pp. 231–283.

[16] D. B. Hitchcock, "A history of the metropolis—Hastings algorithm," *Amer. Statistician*, vol. 57, no. 4, pp. 254–257, 2003.

[17] G. Casella and E. I. George, "Explaining the Gibbs sampler," *Amer. Statistician*, vol. 46, no. 3, p. 167, Aug. 1992.

[18] C. K. Carter and R. Kohn, "On Gibbs sampling for state space models," *Biometrika*, vol. 81, no. 3, p. 541, Aug. 1994.

[19] G. O. Roberts and A. F. M. Smith, "Simple conditions for the convergence of the Gibbs sampler and metropolis—Hastings algorithms," *Stochastic Processes Appl.*, vol. 49, no. 2, pp. 207–216, Feb. 1994.

[20] P. J. Rossky, J. D. Doll, and H. L. Friedman, "Brownian dynamics as smart Monte Carlo simulation," *J. Chem. Phys.*, vol. 69, no. 10, pp. 4628–4633, Nov. 1978.

[21] G. O. Roberts and R. L. Tweedie, "Exponential convergence of Langevin distributions and their discrete approximations," *Bernoulli*, vol. 2, no. 4, p. 341, Dec. 1996.

[22] G. O. Roberts and J. S. Rosenthal, "Optimal scaling of discrete approximations to Langevin diffusions," *J. Roy. Stat. Soc. Ser. B, Stat. Methodol.*, vol. 60, no. 1, pp. 255–268, Jan. 1998.

[23] B. D. Flury, "Acceptance–rejection sampling made easy," *SIAM Rev.*, vol. 32, no. 3, pp. 474–476, Sep. 1990.

[24] W. R. Gilks and P. Wild, "Adaptive rejection sampling for Gibbs sampling," *Appl. Statist.*, vol. 41, no. 2, p. 337, 1992.

[25] S. T. Tokdar and R. E. Kass, "Importance sampling: A review," *Wiley Interdiscipl. Reviews, Comput. Statist.*, vol. 2, no. 1, pp. 54–60, 2010.

[26] F. Llorente, L. Martino, D. Delgado-Gómez, and G. Camps-Valls, "Deep importance sampling based on regression for model inversion and emulation," *Digit. Signal Process.*, vol. 116, Sep. 2021, Art. no. 103104.

[27] A. Brockwell, P. Del Moral, and A. Doucet, "Sequentially interacting Markov chain Monte Carlo methods," *Ann. Statist.*, vol. 38, no. 6, pp. 3387–3411, Dec. 2010.

[28] C. Andrieu and J. Thoms, "A tutorial on adaptive MCMC," *Statist. Comput.*, vol. 18, no. 4, pp. 343–373, Dec. 2008.

[29] R. H. Swendsen and J.-S. Wang, "Replica Monte Carlo simulation of spin-glasses," *Phys. Rev. Lett.*, vol. 57, no. 21, pp. 2607–2609, Nov. 1986.

[30] K. Hukushima and K. Nemoto, "Exchange Monte Carlo method and application to spin glass simulations," *J. Phys. Soc. Jpn.*, vol. 65, no. 6, pp. 1604–1608, Jun. 1996.

[31] D. J. Earl and M. W. Deem, "Parallel tempering: Theory, applications, and new perspectives," *Phys. Chem. Chem. Phys.*, vol. 7, no. 23, p. 3910, 2005.

[32] M. Sambridge, "A parallel tempering algorithm for probabilistic sampling and multimodal optimization," *Geophys. J. Int.*, vol. 196, no. 1, pp. 357–374, Jan. 2014.

[33] Y. Fan and S. A. Sisson, "Reversible jump MCMC," in *Handbook of Markov Chain Monte Carlo*. Boca Raton, FL, USA: CRC Press, 2011, pp. 67–92.

[34] S. P. Brooks, P. Giudici, and G. O. Roberts, "Efficient construction of reversible jump Markov chain Monte Carlo proposal distributions," *J. Roy. Stat. Soc. Ser. B, Stat. Methodol.*, vol. 65, no. 1, pp. 3–39, Feb. 2003.

[35] C. Tarantola, "MCMC model determination for discrete graphical models," *Stat. Model.*, vol. 4, no. 1, pp. 39–61, Apr. 2004.

[36] F. Liang and C. Liu, "Efficient MCMC estimation of discrete distributions," *Comput. Statist. Data Anal.*, vol. 49, no. 4, pp. 1039–1052, Jun. 2005.

[37] G. Zanella, "Informed proposals for local MCMC in discrete spaces," *J. Amer. Stat. Assoc.*, vol. 115, no. 530, pp. 852–865, Apr. 2020.

[38] W. J. Browne, "MCMC algorithms for constrained variance matrices," *Comput. Statist. Data Anal.*, vol. 50, no. 7, pp. 1655–1677, Apr. 2006.

[39] A. R. Gallant, H. Hong, M. P. Leung, and J. Li, "Constrained estimation using penalization and MCMC," *J. Econometrics*, vol. 228, no. 1, pp. 85–106, May 2022.

[40] S. A. Sisson and Y. Fan, *Likelihood-Free MCMC*. Boca Raton, FL, USA: CRC Press, 2011.

[41] D. A. Van Dyk and X.-L. Meng, "The art of data augmentation," *J. Comput. Graph. Statist.*, vol. 10, no. 1, pp. 1–50, 2001.

[42] L. L. Duan, J. E. Johndrow, and D. B. Dunson, "Scaling up data augmentation MCMC via calibration," *J. Mach. Learn. Res.*, vol. 19, no. 1, pp. 2575–2608, 2018.

[43] J. M. Zobitz, A. R. Desai, D. J. P. Moore, and M. A. Chadwick, "A primer for data assimilation with ecological models using Markov chain Monte Carlo (MCMC)," *Oecologia*, vol. 167, no. 3, pp. 599–611, Nov. 2011.

[44] C. Andrieu, P. M. Djurić, and A. Doucet, "Model selection by MCMC computation," *Signal Process.*, vol. 81, no. 1, pp. 19–37, Jan. 2001.

[45] C. Andrieu and A. Doucet, "Joint Bayesian model selection and estimation of noisy sinusoids via reversible jump MCMC," *IEEE Trans. Signal Process.*, vol. 47, no. 10, pp. 2667–2676, Oct. 1999.

[46] A. S. Mugglin, B. P. Carlin, L. Zhu, and E. Conlon, "Bayesian areal interpolation, estimation, and smoothing: An inferential approach for geographic information systems," *Environ. Planning A, Economy Space*, vol. 31, no. 8, pp. 1337–1352, Aug. 1999.

[47] M. Sambridge, "Geophysical inversion with a neighbourhood algorithm—II. appraising the ensemble," *Geophys. J. Int.*, vol. 138, no. 3, pp. 727–746, Sep. 1999.

[48] M. Sambridge and K. Mosegaard, "Monte Carlo methods in geophysical inverse problems," *Rev. Geophys.*, vol. 40, no. 3, pp. 1–14, Sep. 2002.

[49] R. Scalzo, D. Kohn, H. Olierook, G. Houseman, R. Chandra, M. Girolami, and S. Cripps, "Efficiency and robustness in Monte Carlo sampling for 3-D geophysical inversions with obsidian v0.1.2: Setting up for success," *Geoscientific Model Develop.*, vol. 12, no. 7, pp. 2941–2960, Jul. 2019.

[50] R. Chandra, D. Azam, R. D. Müller, T. Salles, and S. Cripps, "Bayeslands: A Bayesian inference approach for parameter uncertainty quantification in badlands," *Comput. Geosci.*, vol. 131, pp. 89–101, Oct. 2019.

[51] H. K. H. Olierook, R. Scalzo, D. Kohn, R. Chandra, E. Farahbakhsh, C. Clark, S. M. Reddy, and R. D. Müller, "Bayesian geological and geophysical data fusion for the construction and uncertainty quantification of 3D geological models," *Geosci. Frontiers*, vol. 12, no. 1, pp. 479–493, Jan. 2021.

[52] L. Marshall, D. Nott, and A. Sharma, "A comparative study of Markov chain Monte Carlo methods for conceptual rainfall-runoff modeling," *Water Resour. Res.*, vol. 40, no. 2, pp. 1–14, Feb. 2004.

[53] J. A. Vrugt, C. J. F. T. Braak, M. P. Clark, J. M. Hyman, and B. A. Robinson, "Treatment of input uncertainty in hydrologic modeling: Doing hydrology backward with Markov chain Monte Carlo simulation," *Water Resour. Res.*, vol. 44, no. 12, pp. 1–17, Dec. 2008.

[54] G. I. Valderrama-Bahamóndez and H. Fröhlich, "MCMC techniques for parameter estimation of ODE based models in systems biology," *Frontiers Appl. Math. Statist.*, vol. 5, p. 55, Nov. 2019.

[55] Y. Nishiyama, Y. Saikawa, and N. Nishiyama, "Interaction between the immune system and acute myeloid leukemia: A model incorporating promotion of regulatory T cell expansion by leukemic cells," *Biosystems*, vol. 165, pp. 99–105, Mar. 2018.

[56] B. Rannala, "Identifiability of parameters in MCMC Bayesian inference of phylogeny," *Systematic Biol.*, vol. 51, no. 5, pp. 754–760, Sep. 2002.

[57] R. Guerra, "Likelihood, Bayesian and MCMC methods in quantitative genetics," *J. Amer. Stat. Assoc.*, vol. 103, no. 481, p. 432, Mar. 2008.

[58] A. J. Drummond, M. A. Suchard, D. Xie, and A. Rambaut, "Bayesian phylogenetics with beauti and the BEAST 1.7," *Mol. Biol. Evol.*, vol. 29, no. 8, pp. 1969–1973, Aug. 2012.

[59] M. Girolami and B. Calderhead, "Riemann manifold Langevin and Hamiltonian Monte Carlo methods," *J. Roy. Stat. Soc. Ser. B, Stat. Methodol.*, vol. 73, no. 2, pp. 123–214, Mar. 2011.
[60] R. M. Neal, "MCMC using Hamiltonian dynamics," in *Handbook of Markov Chain Monte Carlo*, vol. 2, no. 11. Boca Raton, FL, USA: CRC Press, 2011, pp. 1–12.
[61] M. Welling and Y. W. Teh, "Bayesian learning via stochastic gradient Langevin dynamics," in *Proc. Int. Conf. Mach. Learn.*, Bellevue, WA, USA, 2011, pp. 681–688.
[62] R. Chandra, K. Jain, R. V. Deo, and S. Cripps, "Langevin-gradient parallel tempering for Bayesian neural learning," *Neurocomputing*, vol. 359, pp. 315–326, Sep. 2019.
[63] M. M. Drugan and D. Thierens, "Evolutionary Markov chain Monte Carlo," in *Proc. Int. Conf. Artif. Evol.*, 2004, pp. 63–76.
[64] C. J. F. T. Braak, "A Markov chain Monte Carlo version of the genetic algorithm differential evolution: Easy Bayesian computing for real parameter spaces," *Statist. Comput.*, vol. 16, no. 3, pp. 239–249, Sep. 2006.
[65] C. J. F. T. Braak and J. A. Vrugt, "Differential evolution Markov chain with snooker updater and fewer chains," *Statist. Comput.*, vol. 18, no. 4, pp. 435–446, Dec. 2008.
[66] A. Kapoor, E. Nukala, and R. Chandra, "Bayesian neuroevolution using distributed swarm optimization and tempered MCMC," *Appl. Soft Comput.*, vol. 129, Nov. 2022, Art. no. 109528.
[67] L. Li, A. Holbrook, B. Shahbaba, and P. Baldi, "Neural network gradient Hamiltonian Monte Carlo," *Comput. Statist.*, vol. 34, no. 1, pp. 281–299, Mar. 2019.
[68] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proc. COMPSTAT'2010*, 2010, pp. 177–186.
[69] R. Chandra, M. Jain, M. Maharana, and P. N. Krivitsky, "Revisiting Bayesian autoencoders with MCMC," *IEEE Access*, vol. 10, pp. 40482–40495, 2022.
[70] R. Chandra, A. Bhagat, M. Maharana, and P. N. Krivitsky, "Bayesian graph convolutional neural networks via tempered MCMC," *IEEE Access*, vol. 9, pp. 130353–130365, 2021.
[71] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational inference: A review for statisticians," *J. Amer. Stat. Assoc.*, vol. 112, no. 518, pp. 859–877, Apr. 2017.
[72] A. Graves, "Practical variational inference for neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, vol. 24, 2011, pp. 2348–2356.
[73] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra, "Weight uncertainty in neural network," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 1613–1622.
[74] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 56, pp. 1929–1958, 2014.
[75] Y. Gal and Z. Ghahramani, "Dropout as a Bayesian approximation: Representing model uncertainty in deep learning," in *Proc. 33rd Int. Conf. Mach. Learn.*, vol. 48, Jun. 2016, pp. 1050–1059.
[76] K. Shridhar, F. Laumann, and M. Liwicki, "A comprehensive guide to Bayesian convolutional neural network with variational inference," 2019, *arXiv:1901.02731*.
[77] Y. Gal and Z. Ghahramani, "A theoretically grounded application of dropout in recurrent neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 29, 2016, pp. 1–6.
[78] J. M. Bernardo and A. F. Smith, *Bayesian Theory*, vol. 405. Hoboken, NJ, USA: Wiley, 2009.
[79] G. A. Barnard and T. Bayes, "Studies in the history of probability and statistics: IX. Thomas bayes's essay towards solving a problem in the doctrine of chances," *Biometrika*, vol. 45, nos. 3–4, p. 293, Dec. 1958.
[80] A. I. Dale, *A History of Inverse Probability*. Cham, Switzerland: Springer, 2012.
[81] P. S. Laplace, "Memoir on the probability of the causes of events," *Stat. Sci.*, vol. 1, no. 3, pp. 364–378, Aug. 1986.
[82] S. R. Johnson, G. A. Tomlinson, G. A. Hawker, J. T. Granton, and B. M. Feldman, "Methods to elicit beliefs for Bayesian priors: A systematic review," *J. Clin. Epidemiology*, vol. 63, no. 4, pp. 355–369, Apr. 2010.
[83] K. M. Banner, K. M. Irvine, and T. J. Rodhouse, "The use of Bayesian priors in ecology: The good, the bad and the not great," *Methods Ecol. Evol.*, vol. 11, no. 8, pp. 882–889, Aug. 2020.
[84] R. van de Schoot, S. Depaoli, R. King, B. Kramer, K. Märtens, M. G. Tadesse, M. Vannucci, A. Gelman, D. Veen, J. Willemsen, and C. Yau, "Bayesian statistics and modelling," *Nature Rev. Methods Primers*, vol. 1, no. 1, pp. 1–26, 2021.
[85] W. Li and G. Lin, "An adaptive importance sampling algorithm for Bayesian inversion with multimodal distributions," *J. Comput. Phys.*, vol. 294, pp. 173–190, Aug. 2015.
[86] S. Hu, D. S. Poskitt, and X. Zhang, "Bayesian adaptive bandwidth kernel density estimation of irregular multivariate distributions," *Comput. Statist. Data Anal.*, vol. 56, no. 3, pp. 732–740, Mar. 2012.
[87] R. Rojas, "A short proof of the posterior probability property of classifier neural networks," *Neural Comput.*, vol. 8, no. 1, pp. 41–43, Jan. 1996.
[88] H. Akaike, "Likelihood and the Bayes procedure," in *Springer Series in Statistics*. Springer, 1998, pp. 309–332.
[89] D. J. C. MacKay, "Hyperparameters: Optimize, or integrate out?" in *Maximum Entropy and Bayesian Methods*. Cham, Switzerland: Springer, 1996, pp. 43–59.
[90] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin, *Bayesian Data Analysis*. Boca Raton, FL, USA: CRC Press, 2004.
[91] R. McElreath, *Statistical Rethinking: A Bayesian Course With Examples in R and Stan*. Boca Raton, FL, USA: CRC Press, Mar. 2020.
[92] C. R. Harris et al., "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 7825.
[93] P. Virtanen et al., "SciPy 1.0: Fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, pp. 261–272, Feb. 2020.
[94] G. O. Roberts and J. S. Rosenthal, "General state space Markov chains and MCMC algorithms," *Probab. Surveys*, vol. 1, no. none, pp. 20–71, Jan. 2004.
[95] G. O. Roberts, J. S. Rosenthal, J. Segers, and B. Sousa, "Extremal indices, geometric ergodicity of Markov chains, and MCMC," *Extremes*, vol. 9, nos. 3–4, pp. 213–229, Dec. 2006.
[96] C. Andrieu and É. Moulines, "On the ergodicity properties of some adaptive MCMC algorithms," *Ann. Appl. Probab.*, vol. 16, no. 3, pp. 1462–1505, Aug. 2006.
[97] J. Grazzini, "Analysis of the emergent properties: Stationarity and ergodicity," *J. Artif. Societies Social Simul.*, vol. 15, no. 2, p. 7, 2012.
[98] B. Cheng and D. M. Titterington, "Neural networks: A review from a statistical perspective," *Stat. Sci.*, vol. 9, no. 1, pp. 2–30, Feb. 1994.
[99] A. F. Smith, "A general Bayesian linear model," *J. Roy. Stat. Society: Ser. B Methodol.*, vol. 35, no. 1, pp. 67–75, 1973.
[100] E. J. Bedrick, R. Christensen, and W. Johnson, "A new perspective on priors for generalized linear models," *J. Amer. Stat. Assoc.*, vol. 91, no. 436, p. 1450, Dec. 1996.
[101] V. Fortuin, "Priors in Bayesian deep learning: A review," *Int. Stat. Rev.*, vol. 90, no. 3, pp. 563–591, Dec. 2022.
[102] B. P. Hobbs, D. J. Sargent, and B. P. Carlin, "Commensurate priors for incorporating historical information in clinical trials using general and generalized linear models," *Bayesian Anal.*, vol. 7, no. 3, p. 639, Sep. 2012.
[103] E. de Bézenac, A. Pajot, and P. Gallinari, "Deep learning for physical processes: Incorporating prior scientific knowledge," *J. Stat. Mech., Theory Exp.*, vol. 2019, no. 12, Dec. 2019, Art. no. 124009.
[104] R. Chandra and Y. He, "Bayesian neural networks for stock price forecasting before and during COVID-19 pandemic," *PLoS ONE*, vol. 16, no. 7, Jul. 2021, Art. no. e0253217.
[105] A. Krogh and J. A. Hertz, "A simple weight decay can improve generalization," in *Proc. Adv. Neural Inf. Process. Syst.*, 1992, pp. 950–957.
[106] G. C. McDonald, "Ridge regression," *Wiley Interdiscipl. Reviews: Comput. Statist.*, vol. 1, no. 1, pp. 93–100, 2009.
[107] S. I. Gallant, "Perceptron-based learning algorithms," *IEEE Trans. Neural Netw.*, vol. 1, no. 2, pp. 179–191, Jun. 1990.
[108] S. Ruder, "An overview of gradient descent optimization algorithms," 2016, *arXiv:1609.04747*.
[109] E. I. George, U. Makov, and A. Smith, "Conjugate likelihood distributions," *Scandin. J. Statist.*, vol. 1, pp. 147–156, Jun. 1993.
[110] K. P. Murphy, "Conjugate Bayesian analysis of the Gaussian distribution," *Def*, vol. 1, no. 2, p. 16, 2007.
[111] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Netw.*, vol. 2, no. 5, pp. 359–366, Jan. 1989.
[112] C. M. Bishop, *Neural Networks for Pattern Recognition*. London, U.K.: Oxford Univ. Press, 1995.

[113] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.
[114] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986.
[115] B. J. Wythoff, "Backpropagation neural networks: A tutorial," *Chemometrics Intell. Lab. Syst.*, vol. 18, no. 2, pp. 115–155, 1993.
[116] D. E. Rumelhart, R. Durbin, R. Golden, and Y. Chauvin, "Backpropagation: The basic theory," in *Backpropagation: Theory, Architectures Application*. USA: Psychology Press, 1995, pp. 1–34.
[117] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proc. Int. Conf. Mach. Learn.*, 2013, pp. 1139–1147.
[118] M. Moreira and E. Fiesler, "Neural networks with adaptive learning rate and momentum terms," IDIAP Res. Inst., Switzerland, Tech. Rep. 95-04, 1995, pp. 1–29.
[119] R. Battiti, "First- and second-order methods for learning: Between steepest descent and Newton's method," *Neural Comput.*, vol. 4, no. 2, pp. 141–166, Mar. 1992.
[120] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 12, pp. 2121–2159, Jul. 2011.
[121] M. D. Zeiler, "ADADELTA: An adaptive learning rate method," 2012, *arXiv:1212.5701*.
[122] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. ICLR Poster*, 2015, pp. 1–20.
[123] H. Suwa and S. Todo, "Markov chain Monte Carlo method without detailed balance," *Phys. Rev. Lett.*, vol. 105, no. 12, Sep. 2010, Art. no. 120603.
[124] F. Takens, "Detecting strange attractors in turbulence," in *Dynamical Systems and Turbulence, Warwick 1980* (Lecture Notes in Mathematics), vol. 898, D. Rand and L. S. Young, Eds. Berlin, Germany: Springer, 1981.
[125] A. Gelman, W. R. Gilks, and G. O. Roberts, "Weak convergence and optimal scaling of random walk metropolis algorithms," *Ann. Appl. Probab.*, vol. 7, no. 1, pp. 110–120, Feb. 1997.
[126] A. Gelman and D. B. Rubin, "Inference from iterative simulation using multiple sequences," *Stat. Sci.*, vol. 7, no. 4, pp. 457–472, Nov. 1992.
[127] A. Vehtari, A. Gelman, D. Simpson, B. Carpenter, and P.-C. Bürkner, "Rank-normalization, folding, and localization: An improved R^ for assessing convergence of MCMC (with discussion)," *Bayesian Anal.*, vol. 16, no. 2, pp. 1–17, Jun. 2021.
[128] R. Kumar, C. Carroll, A. Hartikainen, and O. Martin, "ArviZ a unified library for exploratory analysis of Bayesian models in Python," *J. Open Source Softw.*, vol. 4, no. 33, p. 1143, Jan. 2019.
[129] W. A. Link and M. J. Eaton, "On thinning of chains in MCMC," *Methods Ecol. Evol.*, vol. 3, no. 1, pp. 112–115, Feb. 2012.
[130] U. Wolff, "Monte Carlo errors with less errors," *Comput. Phys. Commun.*, vol. 156, no. 2, pp. 143–153, Jan. 2004.
[131] D. Foreman-Mackey, D. W. Hogg, D. Lang, and J. Goodman, "EMCEE: The MCMC hammer," *Publications Astronomical Soc. Pacific*, vol. 125, no. 925, pp. 306–312, Mar. 2013.
[132] M. K. Cowles and B. P. Carlin, "Markov chain Monte Carlo convergence diagnostics: A comparative review," *J. Amer. Stat. Assoc.*, vol. 91, no. 434, p. 883, Jun. 1996.
[133] V. Roy, "Convergence diagnostics for Markov chain Monte Carlo," *Annu. Rev. Statist. Its Appl.*, vol. 7, no. 1, pp. 387–412, Mar. 2020.
[134] L. F. South, M. Riabiz, O. Teymur, and C. J. Oates, "Postprocessing of MCMC," *Annu. Rev. Statist. Its Appl.*, vol. 9, no. 1, pp. 529–555, Mar. 2022.
[135] M. D. Hoffman and A. Gelman, "The no-u-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1593–1623, 2014.
[136] A. Patil, D. Huard, and C. Fonnesbeck, "PyMC: Bayesian stochastic modelling inPython," *J. Stat. Softw.*, vol. 35, no. 4, p. 1, 2010.
[137] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell, "Stan: A probabilistic programming language," *J. Stat. Softw.*, vol. 76, no. 1, pp. 1–59, 2017.
[138] J. Annis, B. J. Miller, and T. J. Palmeri, "Bayesian inference with stan: A tutorial on adding custom distributions," *Behav. Res. Methods*, vol. 49, no. 3, pp. 863–886, Jun. 2017.
[139] D. Phan, N. Pradhan, and M. Jankowiak, "Composable effects for flexible and accelerated probabilistic programming in NumPyro," 2019, *arXiv:1912.11554*.
[140] D. Cicchetti and F. A. Rogosch, "Equifinality and multifinality in developmental psychopathology," *Develop. Psychopathology*, vol. 8, no. 4, pp. 597–600, 1996.
[141] C. Gresov and R. Drazin, "Equifinality: Functional equivalence in organization design," *Acad. Manage. Rev.*, vol. 22, no. 2, pp. 403–428, Apr. 1997.
[142] S. P. Brooks and A. Gelman, "General methods for monitoring convergence of iterative simulations," *J. Comput. Graph. Statist.*, vol. 7, no. 4, p. 434, Dec. 1998.
[143] D. Vats and C. Knudson, "Revisiting the gelman–rubin diagnostic," *Stat. Sci.*, vol. 36, no. 4, pp. 518–529, Nov. 2021.
[144] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, pp. 370–403, Oct. 2021.
[145] M. Bédard, "Optimal acceptance rates for metropolis algorithms: Moving beyond 0.234," *Stochastic Processes Appl.*, vol. 118, no. 12, pp. 2198–2222, Dec. 2008.
[146] S. M. Anwar, M. Majid, A. Qayyum, M. Awais, M. Alnowami, and M. K. Khan, "Medical image analysis using convolutional neural networks: A review," *J. Med. Syst.*, vol. 42, no. 11, pp. 1–75, Nov. 2018.
[147] S. Akcay, M. E. Kundegorski, C. G. Willcocks, and T. P. Breckon, "Using deep convolutional neural network architectures for object classification and detection within X-ray baggage security imagery," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 9, pp. 2203–2215, Sep. 2018.
[148] B. Zhao, H. Lu, S. Chen, J. Liu, and D. Wu, "Convolutional neural networks for time series classification," *J. Syst. Eng. Electron.*, vol. 28, no. 1, pp. 162–169, Feb. 2017.
[149] C.-L. Liu, W.-H. Hsaio, and Y.-C. Tu, "Time series classification with multivariate convolutional neural network," *IEEE Trans. Ind. Electron.*, vol. 66, no. 6, pp. 4788–4797, Jun. 2019.
[150] A. Borovykh, S. Bohte, and C. W. Oosterlee, "Conditional time series forecasting with convolutional neural networks," 2017, *arXiv:1703.04691*.
[151] M. Binkowski, G. Marti, and P. Donnat, "Autoregressive convolutional neural networks for asynchronous time series," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 580–589.
[152] R. Chandra, S. Goyal, and R. Gupta, "Evaluation of deep learning models for multi-step ahead time series prediction," *IEEE Access*, vol. 9, pp. 83105–83123, 2021.

ROHITASH CHANDRA (Senior Member, IEEE) is currently a Senior Lecturer in data science with the UNSW School of Mathematics and Statistics. He leads a program of research encircling methodologies and applications of artificial intelligence. The methodologies include Bayesian deep learning, neuroevolution, ensemble learning, and data augmentation. The applications include climate extremes, geoscientific models, mineral exploration, biomedicine, and language modeling. Since 2021, he has been featured in the Stanford's list of top 2% scientists in the world. He served as an Associate Editor for IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS, and *Neurocomputing*, from 2021 to 2022. He is on the Editorial Board of *Geoscientific Model Development*.

JOSHUA SIMMONS received the Ph.D. degree from UNSW Sydney. He is currently a Research Engineer with the ARC Training Centre in Data Analytics for Resources and Environments (DARE). His research has included numerical and statistical modeling of coastal processes, with a particular focus on storm erosion of sandy beaches. More recently, his work has focused on interpretable machine learning/data science approaches and Bayesian methods for uncertainty quantification in the hydrology domain.