MSc e-Science Research Report

Comparison of Hamiltonian Monte Carlo Variants in Bayesian Deep Learning

By

Mufunwa Nemushungwa

Supervisor

Dr Farai Mlambo

A research report submitted to the University of the Witwatersrand's Science Faculty in partial fulfilment of the requirements for the degree of Master of E-Science

February 14, 2025

Abstract

This research investigates the integration of Bayesian inference (BI) with deep learning (DL) models, focusing on Bayesian Deep Learning (BDL) as a solution to overfitting and uncertainty in predictions. By employing posterior probability distributions for uncertainty quantification, BDL enhances decision-making and reliability. Despite various posterior density estimation techniques, including Hamiltonian Monte Carlo (HMC) and its variants, a comprehensive comparative analysis of their performance, computational efficiency, and scalability in BDL remains lacking. This study compares three Bayesian Neural Network (BNN) models using different HMC variants—standard HMC, No-U-Turn Sampler (NUTS), and Stochastic Gradient HMC (SGHMC)—for regression tasks on the California Housing Prices dataset. Metrics such as mean squared error (MSE), root mean squared error (RMSE), and mean absolute error (MAE) were used to assess predictive accuracy, alongside training times to measure computational efficiency and scalability across varying dataset sizes. Results indicate that BNN-SGHMC outperforms the others in predictive accuracy, computational efficiency, and scalability, particularly with larger datasets. This research provides recommendations for practitioners, emphasising the use of BNN-SGHMC for larger datasets, and contributes valuable insights to the field of BDL, paving the way for future studies on advanced HMC techniques.

Contents

List of Figures

List of Tables

List of Acronyms

AF	Activation Function
AI	Artificial Intelligence
ANN	Artificial Neural Networks
BDL	Bayesian Deep Learning
BGD	Batch Gradient Descent
BI	Bayesian Inference
BNN	Bayesian Neural Network
BNN-HMC	Bayesian Neural Network with Hamiltonian Monte Carlo sampler
BNN-NUTS	Bayesian Neural Network with No-U-Turn Sampler
BNN-SGHMC	Bayesian Neural Network with Stochastic Gradient Hamiltonian Monte Carlo sampler
DL	Deep Learning
FFN	Feed-forward Network
HMC	Hamiltonian Monte Carlo
MAE	Mean Absolute Error
MCMC	Markov Chain Monte Carlo
ML	Machine Learning
MSE	Mean Squared Error
MH	Metropolis-Hastings
Mini-BGD	Mini-Batch Gradient Descent
NN	Neural Network
NUTS	No-U-Turn Sampler
RNN	Recurrent Neural Network
RMSE	Root Mean Squared Error
SGD	Stochastic Gradient Descent
SGHMC	Stochastic Gradient Hamiltonian Monte Carlo
VI	Variational Inference

Dedication

I would like to dedicate this research report to my beloved family for their unwavering support, to my partner for being my strength and motivation, and to my precious son, who inspires me daily to push myself and achieve my goals. Thank you all for being my foundation and guiding me to pursue my dreams with passion and dedication.

Acknowledgement

I would like to thank God for guiding me through this research; He has given me strength and courage. I am deeply grateful to my research supervisor, Dr. Farai Mlambo, for his invaluable guidance and encouragement, which have shaped this work in countless ways. To my partner, Sabelo, thank you for your constant support and belief in me—I truly could not have done this without you.

Declaration

I declare that this research report is my own work. It is being submitted for the Degree of Master of Science in e-Science to the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

Chapter 1

Introduction

1.1 Introduction

Traditional deep learning (DL) methods often struggle with overfitting and excessive confidence in predictions, particularly in critical fields such as autonomous driving and medical diagnosis (Jospin et al., 2022). These limitations highlight the need for improved models that can effectively quantify uncertainty. Bayesian Deep Learning (BDL) addresses these challenges by merging principles of Bayesian inference (BI) with DL models. A fundamental advantage of BDL lies in its focus on uncertainty quantification, which enhances the reliability of predictions. Unlike traditional DL methods that provide point estimates, BDL utilises posterior probability distributions to model uncertainty, treating model weights as random variables (Abdullah et al., 2024). This approach involves selecting a prior distribution for model parameters and calculating the likelihood based on observed data, ultimately facilitating effective uncertainty quantification and improved decision-making (Abdullah et al., 2024).

The accurate estimation of posterior density is crucial in BDL, as it allows for the updating of beliefs about unknown parameters with new evidence. Bayesian statistical techniques are employed to approximate the posterior distribution after integrating this new data. However, computing the precise posterior distribution directly is often mathematically infeasible, necessitating the use of alternative estimation techniques (Ma et al., 2021). These techniques are vital in modern statistics and BDL, where accurately approximating complex probability densities is fundamental. When data and model parameters are uncertain, BDL improves uncertainty quantification by efficiently calculating the posterior density, which results in more accurate forecasts and well-informed decision-making (Kianiharchegani, 2023).

1.2 Background

Various techniques for posterior density estimation in BI include Markov Chain Monte Carlo (MCMC), Laplace approximations, and Variational Inference (VI). Hamiltonian Monte Carlo (HMC) is a widely used MCMC method in BDL (Wang et al., 2020). HMC enhances parameter estimation through Hamiltonian dynamics and leapfrog integration, allowing for efficient state space exploration with high acceptance probabilities (Arakawa et al., 2021; Chen et al., 2014).

To improve HMC's performance, variants such as the No-U-Turn Sampler (NUTS) and Stochastic Gradient Hamiltonian Monte Carlo (SGHMC) have been developed. NUTS automates path length tuning, addressing parameter tuning challenges (Nishio and Arakawa, 2022), while SGHMC incorporates stochastic gradients for greater computational efficiency and scalability (Havasi et al., 2018).

Focusing on HMC and its variants is justified by their ability to navigate complex, high-dimensional posterior distributions, reduce autocorrelation, and ensure robust uncertainty quantification—crucial for BDL applications (Cobb and Jalaian, 2021). In contrast, while VI methods prioritise speed, they can produce biased estimates in high dimensions (Cobb and Jalaian, 2021). Thus, the selection of HMC, NUTS, and SGHMC in this research highlights their relevance to the prediction tasks addressed in this study.

1.3 Real-World Implications of HMC Variants

Understanding how these HMC variants perform compared to each other is not just an academic exercise; it has real-world implications that matter. For instance, in finance, improved posterior density estimation can enhance risk predictions, helping analysts make more informed decisions about investments (Kazemi and Mosleh, 2012). In healthcare, effective uncertainty quantification in diagnostic models can lead to more cautious and reliable decision-making, ultimately improving patient outcomes (Seoni et al., 2023). Moreover, the insights gained from comparing these methods could lead to more efficient training processes and greater accuracy in DL models, boosting the reliability of AI systems in high-stakes situations. By shedding light on the strengths of different HMC variants, this research aims to promote the wider adoption of BDL techniques, helping to advance the field and its impact on society.

1.4 Statement of the Problem

Despite the development of HMC variants such as NUTS and SGHMC, a comprehensive comparison of their performance, computational efficiency, and scalability in BDL models for prediction tasks remains insufficient. Previous studies have often analysed these methods in isolation, resulting in a significant gap in research. This lack of comparative analysis leaves researchers uncertain about which HMC variants are most effective for posterior density estimation in practical applications. Consequently, there is no clear guidance on optimising BDL model performance, hindering advancements in fields that rely on robust predictive modelling.

1.5 Research Questions

This research will answer the following questions:

1. How do the HMC, NUTS and SGHMC methods perform in BDL models for prediction tasks in terms of prediction accuracy?
2. What is the computational efficiency of the HMC, NUTS and SGHMC methods in the context of BDL models for prediction tasks?
3. What is the scalability of the HMC, NUTS and SGHMC methods in the context of BDL models for prediction tasks?

1.6 Aims and Objectives

1.6.1 Aims

This research aims to compare the performance, computational efficiency, and scalability of different HMC variants in BDL models, specifically focusing on their application to prediction tasks. This study seeks to provide clear guidance on the most effective HMC variants for optimising posterior density estimation in BDL models.

1.6.2 Objectives

1. To compare the performance of the HMC, NUTS, and SGHMC methods in BDL models in terms of predictive accuracy.
2. To evaluate the computational efficiency of the HMC, NUTS, and SGHMC methods in the context of BDL models for prediction tasks.

3. To evaluate the scalability of the HMC, NUTS, and SGHMC methods in the context of BDL models for prediction tasks.

1.7 Limitations and Assumptions

This research acknowledges several limitations and assumptions. It assumes the selected HMC variants are the most relevant for BDL models, possibly overlooking newer or less common techniques. Performance metrics for prediction are considered comprehensive but may not capture all real-world details. Computational constraints could limit the experiment's scope, especially regarding efficiency and scalability. Finally, the findings depend on specific HMC variant implementations, affecting reproducibility and generalisability.

1.8 Overview of the Report

The remainder of the report is structured as follows: Chapter 2 provides a comprehensive literature review covering the applications of Bayesian neural networks, theoretical foundations of Bayesian neural networks, posterior density estimation techniques, performance evaluation metrics, imputation methods, data reduction techniques, and feature selection methods used in this research. Chapter 3 outlines the research methodology, followed by Chapter 4, which discusses the research results. Chapter 5 concludes the report by summarising the research's key objectives, outcomes, and contributions to the field of BDL.

Chapter 2

Literature Review

2.1 Introduction

This chapter begins by discussing the diverse applications of Bayesian Neural Networks (BNNs) in prediction tasks, emphasising their advantages over traditional models (Section 2.2). The chapter goes on to discuss the application of HMC variants in BDL (Section 2.3). The chapter also reviews the theoretical foundations of these techniques (Section 2.4), including artificial neural networks (ANNs), DL, BDL, BI, and posterior density estimation techniques. Furthermore, relevant metrics for prediction tasks, data sampling, data imputation and feature selection techniques will be analysed.

2.2 Application of Bayesian Neural Networks in Prediction Tasks

Recent studies illustrate the effectiveness of BNNs across various domains:

Fissha et al. (2023) employed BNNs to predict peak particle velocity (PPV) at the Mikurahana quarry in Japan, comparing it with traditional machine learning (ML) models like k-nearest neighbours (kNN) and random forest (RF). The BNN demonstrated superior accuracy and the ability to provide probabilistic predictions, outperforming other models due to its low error rates and nonlinear structure.

In a study by Tirink (2022), BNNs were utilised to predict body weight in Thalli sheep, comparing several algorithms including Bayesian Regularised Neural Network (BRNN) and Support Vector Regression (SVR). The BRNN outperformed the others, showcasing improved accuracy

and robustness in predicting continuous values through various evaluation metrics.

Clare and Piggott (2022) focused on predicting wind speed and direction in offshore environments using BNNs structured with TensorFlow Probability. Their model effectively quantified both epistemic and aleatoric uncertainties, offering calibrated estimates critical for optimising wind farm operations.

These studies underline the growing importance of BNNs in various applications, but they do not address the comparative effectiveness of different HMC variants for posterior density estimation in BDL models.

2.3 Application of Hamiltonian Monte Carlo Methods in Bayesian Deep Learning

The application of HMC methods in BDL has garnered attention due to their potential for improved performance:

Ramchoun and Ettaouil (2020) used HMC to train BNNs for regression and classification tasks, introducing a novel prior distribution that enhanced regularization. Their findings revealed that while HMC methods are computationally intensive, they lead to improved accuracy and reduced error rates.

He et al. (2024) used a BNN model with Automatic Differentiation Variational Inference (ADVI) and NUTS for predicting multiaxial fatigue life, validated on six materials. The BNN outperformed traditional ML models like Extreme Gradient Boost (XGB) and Support Vector Machines (SVM), achieving superior uncertainty quantification and accuracy.

A study by Li et al. (2019) introduced two advanced HMC algorithms—Stochastic Variance Reduced Gradient Hamiltonian Monte Carlo (SVRG-HMC) and Stochastic Average Gradient Augmented Hamiltonian Monte Carlo (SAGA-HMC). These algorithms aimed to enhance Bayesian inference in large-scale datasets, achieving superior convergence rates and accuracy compared to traditional methods.

While these studies highlight advancements in HMC methodologies, they often lack a comparative framework that assesses the effectiveness of different HMC variants in BDL applications. This gap underscores the need for a comprehensive analysis to provide clear guidance for researchers and practitioners seeking to optimise their BDL models.

2.4 Theoretical Review

This section delves into the theoretical foundations of the techniques employed in this research, encompassing ANNs (Subsection 2.4.1), DL (Subsection 2.4.2), and BDL (Subsection 2.4.3), with a focus on BNNs and BI. We describe the BNN training and prediction process (Subsection 2.4.4) and examine posterior density estimation techniques (Subsection 2.4.5), particularly Markov Chain Monte Carlo (MCMC) methods such as Metropolis-Hastings (MH), HMC, the NUTS, and SGHMC. Additionally, we will explore performance evaluation methods and metrics for prediction (Subsection 2.4.6), data imputation (Subsection 2.4.7), data sampling (Subsection 2.4.8), and feature selection techniques (Subsection 2.4.9).

2.4.1 Artificial Neural Networks (ANNs)

This section provides an overview of ANNs, with a specific focus on their definition and structure (Subsection 2.4.1.1), the activation functions commonly used in ANNs (Subsection 2.4.1.2), and optimisation techniques for enhancing ANN performance (Subsection 2.4.1.3).

2.4.1.1 Definition and Structure of Artificial Neural Networks

ANNs, drawing inspiration from the human brain, are artificial intelligence (AI) algorithms composed of interconnected nodes called neurons. Each neuron processes inputs by computing a weighted sum, incorporating a bias, and using an activation function to produce an output (Krenker et al., 2011). During training, the network adjusts these weights to improve performance, while biases adjust outputs for accuracy (Abiodun et al., 2018). Architecturally, ANNs consist of input, output layers, and hidden layers. Information is received by the input layer, transformed by hidden layers, and the output layer generates predictions or classifications (Ghiassi and Saidane, 2005). Feed-forward networks (FFNs) process data sequentially, making them suitable for non-sequential outputs. Recurrent neural networks (RNNs), with feedback loops, manage sequential data, maintaining memory across cycles.

Figure 2.1 illustrates the structure of an FFN (Krenker et al., 2011), a form of ANN. In this diagram, $\mathbf{x_1}, \ldots, \mathbf{x_n}$ represent the input values, $\mathbf{h_1^1}, \ldots, \mathbf{h_m^1}$ denote the hidden layer's nodes, and $\mathbf{y_1}, \ldots, \mathbf{y_k}$ correspond to the output nodes. This ANN does not have feedback loops, and information moves in a single path into the output layer from the input layer. The amount of links between separate distinct artificial nodes, the type of transfer function used in each artificial node, and the amount of layers are all unconstrained (Suzuki, 2011).

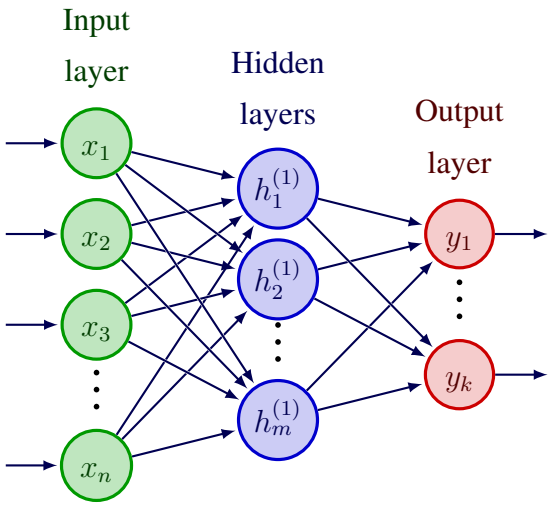


Figure 2.1: Structure of Feed Forward Neural Network Architecture

2.4.1.2 Activation Functions

Activation functions (AFs) are important in ANN as they bring about non-linearity, allowing the network to effectively learn from complicated data patterns (Ferri Borredà, 2024). The weighted total of the inputs is transformed into an output signal by these functions so that it may be sent to the following layer. AFs that are often used include:

a. Sigmoid Activation Function:

The sigmoid AF is a bounded and differentiable function that is non-decreasing, with precisely one inflection point. In simpler terms, it is a smooth, "S-shaped" curve. Because they can compress real values into a bounded interval, sigmoid functions are also called squashing functions (Lederer, 2021). Because it is non-linear, it is the most often utilised AF. The values are changed from 0 to 1 using the sigmoid AF (Sharma et al., 2017). The sigmoid AF is defined by:

$$\Theta(y) = \frac{1}{1 + e^{-y}} \tag{2.1}$$

where we define y as:

$$y = \sum x_i w_i + b \tag{2.2}$$

Here, x_i represents the input values, b is the bias value, and w_i is the weight matrix.

b. Hyperbolic Tangent (Tanh) Activation Function:

The hyperbolic tangent AF, or the tanh function, is a smooth function centred at zero with a range between -1 and 1 (Sharma et al., 2017). The hyperbolic tangent AF is as follows:

$$\tanh(y) = \frac{e^{y} - e^{-y}}{e^{y} + e^{-y}} \tag{2.3}$$

c. Rectified Linear Unit (ReLU):

The Rectified Linear Unit (ReLU) is a non-linear function that outputs zero if the input is negative and outputs the data directly otherwise (Nwankpa et al., 2018). ReLU is favoured for its computational effectiveness and its capability to overcome the disappearing gradient problem, which often affects sigmoid and tanh functions during backpropagation in deep networks (Nwankpa et al., 2018). The ReLU AF is defined as:

$$relu(y) = \begin{cases} y & \text{if } y > 0 \\ 0 & \text{if } y \leq 0 \end{cases} \tag{2.4}$$

d. Softmax Function:

The Softmax function is a type of AF commonly utilised in neural computing. It uses a collection of real numbers to compute a probability distribution. The output of this function lies between 0 and 1, guaranteeing that the probabilities sum up to 1 (Nwankpa et al., 2018). The Softmax function is particularly useful in models with multiple classes, as it provides probabilities for every class. The highest probability is assigned to the target class (Nwankpa et al., 2018). The Softmax AF is as follows:

Suppose the network has k output nodes: $n_1, \ldots, n_k$. For each i, let z_{n_i} be the variable for the sum of the inputs to the node plus bias at node n_i, then the softmax AF at node n_i is calculated as follows:

$$\text{softmax}\,(n_i) = \frac{e^{y_{n_i}}}{\sum_{j=1}^{k} e^{y_{n_j}}} \tag{2.5}$$

e. Linear Function:

The linear AF is directly proportional to the input, leaving the inputs unchanged (Sharma et al., 2017). This AF is also referred to as the identity linear function in the context of NN (Lederer, 2021).
The linear activation function is given by:

$$\text{lin}(y) = y \tag{2.6}$$

2.4.1.3 Artificial Neural Network Optimisation Techniques: Gradient Descent Techniques

Gradient descent is a popular optimisation technique in ML, which minimises a function by repeatedly updating the parameters in the direction of the steepest descent (Büyükkaya et al., 2024). This subsection discusses three gradient descent techniques: Batch Gradient Descent, Stochastic Gradient Descent and Mini-batch Gradient Descent.

1. Batch Gradient Descent

The traditional form of gradient descent, known as batch gradient descent (BGD), uses the whole data set to calculate the cost function's gradient and update the parameters (Mustapha et al., 2020). This approach is known for its stability but may be computationally costly for big datasets (Jha et al., 2023).

Mustapha et al. (2020) states that the update rule for BGD is given by:

$$\Theta = \Theta - \eta \nabla F(\Theta)$$

where:

- Θ represents the parameters,
- η is the learning rate,
- $F(\Theta)$ is the cost function,
- $\nabla F(\Theta)$ is the gradient of the cost function with respect to Θ.

Büyükkaya et al. (2024) further elaborates that BGD will inevitably converge to a local minimum for non-convex functions and to the global minimum for convex functions. However, its computational demand increases significantly as the dataset size grows, leading to slower convergence.

2. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) revises the parameters for every training example, introducing more noise into the process, which can potentially lead to faster convergence (Mustapha et al., 2020). However, this noise can also make the path to the minimum less stable (Büyükkaya et al., 2024).

The update rule for Stochastic Gradient Descent, as discussed by Mustapha et al. (2020), is:

$$\Theta = \Theta - \eta \nabla F(\Theta; w^{(i)}, z^{(i)})$$

where:

- $(w^{(i)}, z^{(i)})$ is the i-th training example,
- $\nabla F(\Theta; w^{(i)}, z^{(i)})$ is the cost function's gradient for the i-th training example.

Büyükkaya et al. (2024) notes that SGD can be significantly faster than BGD since it adjusts the parameters more often. However, the convergence may be less stable, and there is a risk of not converging to the global minimum, especially for non-convex functions.

3. Mini-Batch Gradient Descent

Mini-batch Gradient Descent (Mini-BGD) is the middle point between BGD and SGD. It divides the training dataset into mini-batches and adjusts the parameters using each small-batch (Jha et al., 2023). This approach combines the benefits of both methods: the stability of BGD and the efficiency of SGD (Büyükkaya et al., 2024).

The rule for updating the Mini-BGD Descent as outlined by Mustapha et al. (2020), is:

$$\Theta = \Theta - \eta \nabla F(\Theta; W^{(i:i+n)}, Z^{(i:i+n)})$$

where:

- $W^{(i:i+n)}$ and $Z^{(i:i+n)}$ are the i-th mini-batch of training examples,
- $\nabla F(\Theta; W^{(i:i+n)}, Z^{(i:i+n)})$ is the cost function's gradient for the mini-batch.

Büyükkaya et al. (2024) emphasises that Mini-BGD updates the parameters more frequently than BGD, which can enhance computational efficiency. Additionally, it provides better stability than SGD, as the gradient is computed over multiple examples.

Jha et al. (2023) further adds that Mini-BGD helps capture time-wise variations in data, making it less time-consuming and more computationally efficient. They also highlight that the choice of mini-batch size is crucial, as it impacts the performance of the algorithm.

4. Comparison and Applications

According to Jha et al. (2023), BGD can be computationally expensive and slow due to the need to compute gradients for the entire dataset at each iteration Jha et al. (2023). In contrast, Mini-BGD, as noted by Mustapha et al. (2020), strikes a balance by reducing computational load while maintaining stability. SGD, on the other hand, is noted for its speed and reduced computational expense, making it suitable for dynamic environments where fuel and combustion characteristics vary over time (Jha et al., 2023).

Mustapha et al. (2020)'s comparison of these techniques highlights:

- **BGD**: Offers stable convergence but is slow and computationally expensive for large datasets.
- **SGD**: Provides faster updates and can escape local minima effectively due to its noisy updates, but may lead to oscillatory convergence.
- **Mini-BGD**: Balances the trade-offs between BGD and SGD, providing faster convergence than BGD and less oscillatory updates than SGD.

2.4.2 Deep Learning

DL first emerged in ML in 1986 and became prominent in 2006 with Hinton et al.'s introduction of DL based on ANN (Minar and Naher, 2018). This sparked renewed interest in NNs, which excel in regression and classification problems, making DL a key topic in AI, ML, data science, and analytics (Sarker, 2021). DL mimics the human brain's data processing using NNs with input, hidden, and output layers (Abdullah et al., 2022; Sarker, 2021). Unlike traditional statistical methods, DL autonomously generates complex hypotheses and learns nonlinear relationships (Dong et al., 2021). This makes it highly effective for classification, feature learning, and pattern recognition in various fields, including sentiment analysis, business intelligence, healthcare, cybersecurity, and visual recognition (Sarker, 2021).

2.4.3 Bayesian Deep Learning (BDL)

This section gives an overview of BDL (Subsection 2.4.3.1), and delves further into it, with a specific focus on defining and structuring BNNs (Subsection 2.4.3.2). It covers key elements such as uncertainty quantification (Subsection 2.4.3.3), Bayes' Theorem (Subsection 2.4.3.4), and the roles of prior distribution (Subsection 2.4.3.5), likelihood (Subsection 2.4.3.6), posterior distribution (Subsection 2.4.3.7), and BI (Subsection 2.4.3.8) in constructing BNNs.

2.4.3.1 Overview of Bayesian Deep Learning

Recently, BDL has become known as a comprehensive probabilistic framework that merges DL with Bayesian models (Wang and Yeung, 2020). It enhances DL models with uncertainty handling by assigning distributions to their weights (Song et al., 2021). BDL uses Bayesian probability theory to enhance traditional NNs by combining prior expert knowledge with data likelihood. This approach generates posterior distributions that capture various uncertainties within the model (Abdullah et al., 2022). BDL, often referred to as BNN, focuses on estimating the posterior distribution of data. It achieves this by employing Bayesian methods to tune weights, optimise models, and define AFs. This approach provides a robust framework for handling uncertainties in DL tasks (Abdullah et al., 2022).

2.4.3.2 Definition and Structure of Bayesian Neural Networks

A BNN integrates BI into NNs, introducing stochastic elements such as stochastic weights and AFs. This allows BNNs to model multiple potential configurations, each with its own probability distribution. This approach is similar to ensemble learning principles, which focus on understanding prediction uncertainty rather than solely optimising performance (Jospin et al., 2022). Unlike traditional ANNs, where weights are fixed, BNNs treat weights and biases as probability distributions during training and inference. This approach provides probabilistic guarantees on predictions, offering insights into model parameter distributions and associated uncertainties (Mullachery et al., 2018). Epistemic uncertainty in BNNs arises from the variability in model parameters, contrasting with deterministic outputs from ANNs (Song et al., 2021; Yu et al., 2022).

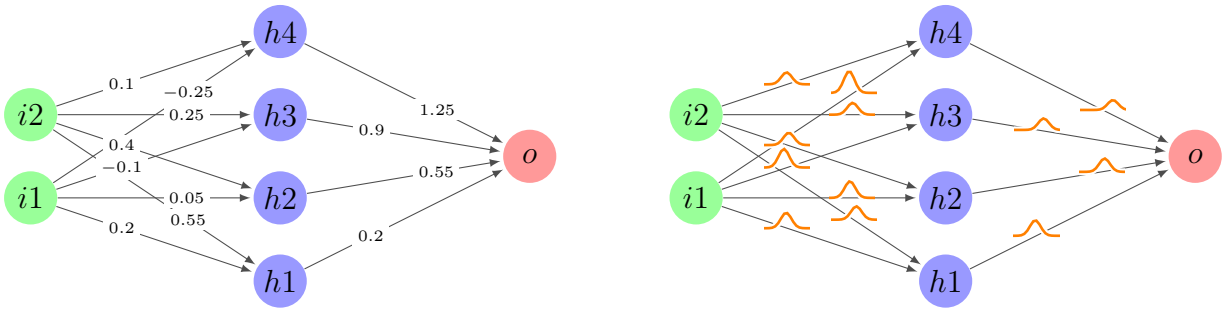


Figure 2.2: Architectural Differences Between ANN and BNN

Figure 2.2 illustrates the architectural differences between a BNN and traditional deep networks. BNNs employ probability distributions for the parameters of their model (biases and weights), unlike traditional deep networks. This allows them to interpret model outputs without the need for repeated testing or dataset cross-validation. This capability stems from the use of likelihood and prior distributions, which come in a variety of forms such as Beta, Normal, Cauchy and Gamma (Abdullah et al., 2022).

2.4.3.3 Uncertainty Quantification and Practical Applications

The primary motivation for using BNNs is their ability to quantify uncertainty in predictions. BNNs quantify uncertainty through the comparison of the predictions of various sampling model parameterisations: low uncertainty if models agree, and high uncertainty if they don't (Bykov et al., 2021). This capability is crucial in critical applications such as medicine and autonomous driving, where knowing prediction uncertainties can lead to more informed decision-making, potentially saving lives and improving safety by better handling ambiguous situations (Wang and Yeung, 2020).

2.4.3.4 Bayes Theorem

A cornerstone of probability theory, Bayes' theorem is essential in diverse fields including predictive analytics, disease forecasting, and intrusion detection. The theorem allows for calculating conditional probabilities by incorporating prior knowledge or beliefs (Venkatesh et al., 2019). It is critically important in advanced ML models and inferential statistics. Bayesian reasoning offers a systematic method for revising the probability of hypotheses based on new evidence (Berrar, 2019). The Bayesian framework enables posterior inference of parameters, allowing for the revision of initial beliefs using data-driven likelihood. This process of posterior estimation adheres to Bayes' theorem, which is as follows (Mbuvha and Marwala, 2020):

$$P(\nu \mid \eta) = \frac{P(\eta \mid \nu)P(\nu)}{P(\eta)} \tag{2.7}$$

The Bayesian interpretation treats ν as a hypothesis about which there is an initial belief, and η as data that updates this belief. The model's aleatoric uncertainty, or uncertainty resulting from randomness, is represented by the likelihood also known as the probability distribution $P(\eta|\nu)$. The Prior is represented by $P(\nu)$, while the evidence is represented by $P(\eta)$. $P(\nu|\eta)$, known as the posterior, captures the uncertainty due to insufficient data or epistemic uncertainty. The product $P(\eta|\nu)P(\nu)$, equal to $P(\eta, \nu)$, is the joint probability of η and ν (Jospin et al., 2022).

2.4.3.5 Prior Distribution

In Bayesian statistics, priors represent the initial beliefs about the parameters before observing any data. Priors encode subject matter knowledge into parameter estimation (Mbuvha and Marwala, 2020). Mathematically, the prior probability $P(\nu)$ is the probability of a hypothesis ν before having any information about the data η (Westbury, 2010). Similarly, $P(\eta)$ is the prior probability of η, which is computed by averaging the probabilities of η across all possible hypotheses weighted by their prior probabilities. The prior probability is calculated as (Etz and Vandekerckhove, 2018):

$$P(\eta) = P(\eta|\nu_1)P(\nu_1) + P(\eta|\nu_2)P(\nu_2) \tag{2.8}$$

where $P(\nu_1)$ and $P(\nu_2)$ are the prior probabilities of the competing hypotheses ν_1 and ν_2, and $P(\nu_1) = 1 - P(\nu_2)$. The prior $P(\nu)$ thus represents the initial degree of belief in the hypothesis ν, quantifying its a priori plausibility (Berrar, 2019).

2.4.3.6 Likelihood

The likelihood function is a mathematical representation of the probability of observing a given dataset under a particular model or hypothesis (Mbuvha and Marwala, 2020). Noted as $P(\eta|\nu)$, it signifies the likelihood of observing the reported data η assuming that the hypothesis ν is true

Berrar (2019). This function can be understood as the degree to which the data supports the hypothesis and is essential for evaluating different hypotheses based on their abilities to predict the observed data (Etz and Vandekerckhove, 2018).

2.4.3.7 Posterior Distribution

The posterior distribution, represented as $P(\nu|\eta)$, is the conditional probability of a hypothesis ν for the observed data η. Lambert (2018); Westbury (2010). It results from updating the prior probability $P(\nu)$ with the information summarised by the likelihood $P(\eta|\nu)$ through Bayes' rule (Lambert, 2018). This distribution represents the updated knowledge or uncertainty about the parameter after incorporating the observed data (Robert et al., 2010).

2.4.3.8 Bayesian Inference in BNN

BI is a versatile approach for summarising uncertainty and generating estimates and predictions. It relies on probability statements that are based on observed data and a specified model (Robert et al., 2010). In the context of training a BNN, this involves selecting a prior distribution over the model parameters, which represents our initial assumptions before observing any data. The posterior distribution, which reflects our revised assumptions about the parameters, is then created by updating this prior in light of the observed data (Chang, 2021). BI uses Bayes' Theorem to compute the probability that our hypothesis, represented by a model with either known or unknown parameters, is correct given the observed sample data (Ellison, 2004). This process is fundamental to the operation of BNNs.

2.4.4 Training Bayesian Neural Networks

This section explains the design and use of BNNs (Subsection 2.4.4.1), covering their application in prediction (Subsection 2.4.4.2), training processes (Subsection 2.4.4.3), and the challenges inherent in training BNNs effectively (Subsection 2.4.4.4).

2.4.4.1 Designing and Using Bayesian Neural Networks

Designing a BNN involves several crucial steps. Initially, one selects a deep neural network (DNN) architecture as the functional model. Subsequently, a stochastic model is chosen, encompassing a prior distribution $p(\vartheta)$ of the model's potential parameters and the prior belief in the predictive capability of the model $p(z|w, \vartheta)$ (Jospin et al., 2022).

2.4.4.2 Using Bayesian Neural Networks for Prediction

BNNs excel in predictive tasks by estimating uncertainties. The posterior distribution $p(\vartheta|S)$ of weights is used to derive a pivotal metric known as the marginal distribution $p(z|w,S)$ (Jospin et al., 2022):

$$p(z|w,S) = \int_{\vartheta} (z|w,\vartheta')p(\vartheta'|S)d\vartheta' \tag{2.9}$$

In practical terms, the indirect sampling of $p(z|w,S)$ occurs. A large collection of weights ϑ_i are sampled from the posterior distribution $p(\vartheta|S)$ and are used to calculate a series of potential outputs z_i. This process is described in Algorithm 1:

Algorithm 1 BNN Inference procedure

Specify $p(\vartheta|S) = \frac{p(S_z|S_w,\vartheta)p(\vartheta)}{\int_{\vartheta} p(S_z|S_w,\vartheta')p(\vartheta')d\vartheta'}$

for $j = 0$ to M **do**

Select $\vartheta_j \sim p(\vartheta|S)$;

$z_j = \Psi_{\vartheta_j}(w)$;

end for

Return $Z = \{z_j|j \in [0,M]\}$, $\vartheta = \{\vartheta_j|j \in [0,M]\}$;

A collection of samples from $p(z|w,S)$ is represented by Z in this algorithm, and a set of samples from $p(\vartheta|D)$ is represented by ϑ. These samples are used to estimate the output z, also known as $\hat{z}$ and to measure the BNN's uncertainty. (Jospin et al., 2022).

For regression tasks, the predictions of a BNN are typically summarised through model averaging (Jospin et al., 2022):

$$\hat{z} = \frac{1}{|\vartheta|} \sum_{\vartheta_j \in \vartheta} \Psi_{\vartheta_j}(w). \tag{2.10}$$

This ensemble method is frequently used in ML. To measure uncertainty, the covariance matrix is computed as follows:

$$\Sigma_{z|w,S} = \frac{1}{|\vartheta| - 1} \sum_{\vartheta_j \in \vartheta} (\Psi_{\vartheta_j}(w) - \hat{z})(\Psi_{\vartheta_j}(w) - \hat{z})^{\top}. \tag{2.11}$$

2.4.4.3 Training and Prediction

During training, a BNN undergoes forward-propagation using sampled weights $\vartheta = (\mu, \sigma^2)$ where the distribution's mean vector is represented by μ and the standard deviation vector by σ. Backward-propagation updates μ and σ gradients via the reparameterization trick, crucial

for sampling from distributions like the standard normal (Yu et al., 2022).

Post-training, predictions in BNNs rely on Bayes' theorem for integration over the model's probabilistic parameters (weights), facilitating Monte Carlo (MC) sampling to generate predictions $P(z|w, S)$ (Yu et al., 2022). Epistemic uncertainty, indicative of weights' uncertainty, is estimated using predictive entropy H, a metric reflecting prediction confidence (Yu et al., 2022).

2.4.4.4 Challenges in Training Bayesian Neural Networks

One of the main obstacles to training BNNs is the computational challenge of estimating and sampling from the posterior distribution. This distribution is often high-dimensional and non-convex, making the process complex and resource-intensive (Jospin et al., 2022). Exact computation is often infeasible, necessitating the use of approximation methods. Common approaches include variational inference, various forms of dropout techniques like Monte Carlo dropout and variational dropout, and MCMC.

2.4.5 Techniques for Posterior Density Estimation

This section examines posterior density estimation techniques in BDL, with a particular focus on Markov Chain Monte Carlo (MCMC) methods. It provides an overview of MCMC techniques (Subsection 2.4.5.1) and a detailed discussion of those best suited for BDL, including Metropolis-Hastings (Subsection 2.4.5.2), HMC (Subsection 2.4.5.3), NUTS (Subsection 2.4.5.4), and SGHMC (Subsection 2.4.5.5).

2.4.5.1 Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) is a set of methods that use computer simulation of a Markov chain to determine unknown parameters or properties of a given distribution. Quantiles, density functions, moments, and many others could be included in this list. The resulting estimates are subsequently utilised for statistical inference (Jones and Qin, 2022). MCMC methods require a burn-in period to minimise initial bias, followed by subsampling to obtain independent samples. For drawing samples from posterior distributions in Bayesian statistics, MCMC works well despite issues like sample autocorrelation and computing load (Jospin et al., 2022). It starts with a random draw from an initial distribution and applies a stochastic transition operator iteratively, converging to the exact posterior distribution. However, it can be difficult and time-consuming to determine how many iterations are needed for a decent approximation (Salimans et al., 2015).

In Bayesian statistics, MCMC methods are widely recognised as effective and popular techniques for sampling exact posterior distributions. However, it's essential to note that not all MCMC algorithms are suitable for BDL applications. The following MCMC algorithms have been identified as particularly well-suited for BDL:

2.4.5.2 Metropolis-Hastings (MH) Algorithm

The MH algorithm which was developed in 1953 by Nicholas Metropolis, accelerates computations for substances with interacting molecules by using a generalised rejection sampling approach. It proposes samples based on the outcome of previous iterations and reuses values when samples are rejected, resulting in interdependent but efficiently sampled sequences (Luengo et al., 2020). Unlike traditional rejection sampling methods, the MH algorithm can accommodate proposal densities that are lower than the target distribution. This feature assists in the convergence of its underlying Markov chain toward the target distribution (Luengo et al., 2020). This algorithm's flexibility makes it particularly suitable for BNNs, as it only requires a proportional function $f(x)$ rather than exact knowledge of $P(x)$, simplifying sampling from Bayesian posterior distributions (Jospin et al., 2022).

The algorithm begins with an initial random guess ϑ_0 and uses a proposal distribution $Q(\vartheta'|\vartheta_0)$ to generate a new candidate point ϑ' around ϑ_0. ϑ' is accepted if, according to the target distribution, it is more likely to be accepted than ϑ_0; if not, it may be accepted with a certain probability or rejected (Jospin et al., 2022). When Q is symmetric, meaning $Q(\vartheta' \mid \vartheta_n) = Q(\vartheta_n \mid \vartheta')$, the acceptance probability reduces to:

$$p = \min\left(1, \frac{f\left(\boldsymbol{\vartheta}'\right)}{f\left(\boldsymbol{\vartheta}_n\right)}\right), \tag{2.12}$$

where $f(\boldsymbol{\vartheta})$ denotes the target distribution.

This setup characterises the Metropolis method. Popular choices for Q include normal and uniform distributions centred on the preceding sample.

The full MH technique introduces a corrective term that has to be incorporated into non-symmetric proposal distributions. The variance of $Q(\vartheta' \mid \vartheta_n)$ must be carefully tuned: if it is too small, the samples will exhibit high autocorrelation; the rejection rate will be high if it is too large. There is no one-size-fits-all approach to optimising these parameters, but a well-designed strategy can mitigate their impact, which is why methods such as HMC have been proposed. (Jospin et al., 2022).

The MH algorithm is as follows (Jospin et al., 2022):

Algorithm 2 Metropolis-Hastings Algorithm

Select $\vartheta_0 \sim$ Initial probability distribution

for $m = 0$ to M **do**

Select $\vartheta' \sim Q(\vartheta' \mid \vartheta_m)$

$p = \min\left(1, \frac{Q(\vartheta'|\vartheta_m)}{Q(\vartheta_m|\vartheta')} \frac{f(\vartheta')}{f(\vartheta_m)}\right)$

Draw $j \sim \text{Bernoulli}(p)$

if j **then**

$\vartheta_{m+1} = \vartheta'$

else

$\vartheta_{m+1} = \vartheta_n$

end if

end for

2.4.5.3 Hamiltonian Monte Carlo (HMC)

The HMC technique is an effective MCMC tool for drawing samples from probability distributions. The HMC techique uses a Hamiltonian system with flexible settings for the duration of the Hamiltonian dynamics and the size of each integration step (Bou-Rabee and Sanz-Serna, 2017). This flexibility allows HMC to propose moves over a broad range of distances within the parameter space, while still maintaining elevated acceptance rates (Bou-Rabee and Sanz-Serna, 2017).

HMC is part of the MH family and addresses issues like high rejection rates and strong sample autocorrelation found in traditional methods (Jospin et al., 2022). It uses Hamiltonian dynamics to propose new states, defined by a Hamiltonian function that combines potential and kinetic energy (Chen et al., 2014):

$$H(\upsilon, \rho) = U(\upsilon) + \frac{1}{2}\rho^T M^{-1} \rho \tag{2.13}$$

where, υ represents the position variables (parameters of interest), ρ represents auxiliary momentum variables, $U(\upsilon)$ represents the potential energy function and $\frac{1}{2}\rho^T M^{-1}\rho$ represents the kinetic energy term with M being the mass matrix.

The motion under Hamiltonian dynamics follows these differential equations:

$$\begin{cases} d\upsilon = M^{-1}\rho \, dt \\ d\rho = -\nabla U(\upsilon) \, dt \end{cases} \tag{2.14}$$

Chen et al. (2014) state that these dynamics guide the proposed states along a path based on physical principles, enabling the exploration of distant states in the parameter space with elevated acceptance probabilities .

The leapfrog method is used to achieve discretization. It maps the state from time t to $t+s$, ensuring reversibility and preservation of the total energy H. The leapfrog method consists of the following steps (Chen et al., 2014):

1. Update the momentum ρ by half-step: $\rho \leftarrow \rho - \frac{\epsilon}{2}\nabla U(\upsilon)$.
2. Update the position υ by full step: $\upsilon \leftarrow \upsilon + \epsilon M^{-1}\rho$.
3. Update the momentum ρ by another half-step: $\rho \leftarrow \rho - \frac{\epsilon}{2}\nabla U(\upsilon)$.

Where ϵ is the step size.

The HMC algorithm is as follows:

Algorithm 3 Hamiltonian Monte Carlo

Input: Step size ϵ and initial position $\upsilon^{(1)}$

for $t = 1, 2, \cdots$ **do**

Sample new momentum $\rho^{(t)} \sim N(0, M)$

$(\upsilon_0, \rho_0) = (\upsilon^{(t)}, \rho^{(t)})$

Replicate discretisation of Hamiltonian motion in Eq. 2.14:

$\rho_0 \leftarrow \rho_0 - \frac{\epsilon}{2}\nabla U(\upsilon_0)$

for $j = 1$ to n **do**

$\upsilon_j \leftarrow \upsilon_{j-1} + \epsilon M^{-1}\rho_{j-1}$

$\rho_j \leftarrow \rho_{j-1} - \epsilon\nabla U(\upsilon_j)$

end for

$\rho_n \leftarrow \rho_n - \frac{\epsilon}{2}\nabla U(\upsilon_n)$

$(\hat{\upsilon}, \hat{\rho}) = (\upsilon_n, \rho_n)$

MH correction:

$u \sim \text{Uniform}[0, 1]$

$\varphi = \exp\left(H(\hat{\upsilon}, \hat{r}) - H(\upsilon^{(t)}, \rho^{(t)})\right)$

if $u < \min(1, \varphi)$ **then**

$\upsilon^{(t+1)} = \hat{\upsilon}$

end if

end for

The leapfrog method tends to suffer from discretization inaccuracies. However, it maintains high acceptance rates for distant proposals by including an MH correction step, as shown in Algorithm 3. This ensures good accuracy and acceptance rates, making HMC an effective method for drawing samples from intricate, high-dimensional posterior distributions (Chen et al., 2014).

2.4.5.4 No-U-Turn Sampler (NUTS)

The NUTS expands upon HMC by automating the determination of trajectory length (L) and step size (ϵ), eliminating the need for manual specification (Hoffman et al., 2014). It dynamically adjusts these parameters during the burn-in phase, optimising the exploration of the parameter space without user intervention (Wahle, 2021).

NUTS simulates Hamiltonian dynamics forward and backwards in time, constructing a balanced binary tree of candidate states. It stops when states begin to retrace or when discretization errors become significant, proposing distant states to enhance exploration while avoiding redundancy (Wahle, 2021). This approach improves efficiency and performance compared to traditional HMC, making NUTS particularly effective for complex, high-dimensional models where manual tuning can be challenging and time-consuming (Chen et al., 2014).

Nishio and Arakawa (2019) show that the NUTS sampling procedure can be summarised by the following algorithm:

Algorithm 4 No-U-Turn Sampler (NUTS)

Set initial values of ϑ, ϵ, δ, μ, γ, j_0, and κ.
while sampling **do**
  Generate momentum $\rho \sim \mathcal{N}(0, I)$.
  Generate auxiliary variable $u \sim \text{Uniform}(0, \exp(\log f(\vartheta) - \frac{1}{2}\rho' M^{-1}\rho))$.
  Using the doubling approach and the transition kernel T, generate the candidate set C.
  At the jth iteration, accept the proposal (ϑ^*, ρ^*) with probability α_j
  if during warm-up phase **then**
    Update ϵ_j by dual averaging.
  end if
end while

The NUTS algorithm, as outlined by Nishio and Arakawa (2019), initialises variables ϑ, ϵ, δ, μ, γ, j_0, and κ. During sampling, momentum ρ is generated from a normal distribution $\mathcal{N}(0, I)$, and a uniform distribution is used to generate an auxiliary variable u. Using the doubling approach and a given transition kernel T, the algorithm constructs a candidate set C. Each

iteration evaluates whether to accept a proposal (ϑ^*, ρ^*) based on the acceptance probability α_j. During the warm-up phase, ϵ_j is updated using dual averaging. Parameters such as δ, μ, γ, j_0, and κ adjust the acceptance probability and convergence behavior. This approach improves efficiency in sampling from complex distributions by dynamically adapting parameters without manual intervention.

2.4.5.5 Stochastic Gradient Hamiltonian Monte Carlo (SGHMC)

SGHMC integrates Hamiltonian dynamics and stochastic gradients to improve efficiency in sampling complex posterior distributions, especially in large-scale data contexts. It enhances traditional methods by adding properly scaled Gaussian noise to stochastic gradients, facilitating exploration towards global minima while avoiding the high computational costs associated with exact gradient computations in large datasets (Havasi et al., 2018).

An auxiliary variable ρ is introduced by SGHMC to sample from the joint distribution $p(\upsilon, \rho|z)$, which can be approximated by:

$$p(\upsilon, \rho|z) \approx \exp\left(-U(\upsilon) - \frac{1}{2}\rho^T M^{-1}\rho\right) \tag{2.15}$$

where $U(\upsilon) = -\log p(\upsilon|z)$. This method effectively utilises stochastic gradient estimates derived from data subsampling to approximate gradients $\nabla U(\upsilon)$, enabling efficient posterior sampling without the need for exhaustive gradient computations. The update equations incorporate terms like the friction coefficient C, mass matrix M, and Fisher information matrix $\hat{B}$, with ϵ representing the step size (Havasi et al., 2018).

The SGHMC algorithm is as follows:

Algorithm 5 Stochastic Gradient Hamiltonian Monte Carlo

for $t = 1, 2, \ldots$ **do**

Reinitialise momentum ρ as $\rho^{(t)} \sim \mathcal{N}(0, M)$, if desired.

$(\vartheta_0, \rho_0) = (\vartheta^{(t)}, \rho^{(t)})$

replicate dynamics:

for $k = 1$ to n **do**

$\vartheta_k \leftarrow \vartheta_{k-1} + \epsilon_t M^{-1}\rho_{k-1}$

$\rho_k \leftarrow \rho_{k-1} - \epsilon_t \nabla\tilde{U}(\vartheta_k) - \epsilon_t C M^{-1}\rho_{k-1} + \mathcal{N}(0, 2(C - \hat{B})\epsilon_t)$

end for

$(\vartheta^{(t+1)}, \rho^{(t+1)}) = (\vartheta_m, \rho_m)$, does not have Metropolis-Hastings step

end for

Parameter tuning in SGHMC, such as C, M, $\hat{B}$, and ϵ, poses challenges, addressed through auto-tuning methods, particularly beneficial for BNNs and Deep Gaussian Processes (DGPs) (Havasi et al., 2018). SGHMC demonstrates faster convergence rates compared to alternatives like Stochastic Gradient Langevin Dynamics (SGLD), showcasing its efficacy in diverse ML tasks, including covariance estimation and NN classification (Li et al., 2019).

2.4.6 Performance Evaluation

This section discusses the evaluation metrics used for prediction tasks and highlights their relevance to this research. The metrics include Mean Squared Error (Subsection 2.4.6.1), Root Mean Squared Error (Subsection 2.4.6.2), and Mean Absolute Error (Subsection 2.4.6.3).

2.4.6.1 Mean Squared Error (MSE)

The mean squared error (MSE) calculates the difference between the expected and actual values (Fissha et al., 2023). It is derived by summing these squared differences and dividing by the number of instances, as demonstrated below:

$$MSE = \frac{1}{m}\sum_{i}^{m}(z_i - \hat{z}_i)^2 \tag{2.16}$$

2.4.6.2 Root Mean Squared Error (RMSE)

The extent of the discrepancy between predicted and observed values is measured by the Root Mean Square Error (RMSE), a statistic that is used to assess a model's performance (Fissha et al., 2023). It is calculated as follows:

$$RMSE = \sqrt{\frac{1}{m}\sum_{i}^{m}(z_i - \hat{z}_i)^2} \tag{2.17}$$

2.4.6.3 Mean Absolute Error (MAE)

The MAE, also known as Mean Absolute Deviation, gives us the average absolute difference between the observed value and the predicted values (Tatachar, 2021). It is computed as follows:

$$MAE = \frac{1}{m}\sum_{i}^{m}|z_i - \hat{z}_i| \tag{2.18}$$

These metrics offer a comprehensive evaluation of the predictive capabilities of the BNN models, highlighting the impact of each sampling method—HMC, NUTS, and SGHMC—on performance and informing future applications in fields where accurate predictions are essential.

2.4.7 Data Imputation Techniques

The dataset used for the prediction task in this research contains missing values. This section outlines the imputation methods used to address these gaps, specifically mean imputation (Subsection 2.4.7.1) and k-nearest neighbours (kNN) imputation (Subsection 2.4.7.2), selected for their effectiveness in handling missing data within the dataset.

2.4.7.1 Mean Imputation

Mean imputation, a simple yet common technique for handling missing data, replaces missing values with the average of the existing data points in a specific variable. While easy to implement, it can introduce bias, particularly when the missing data isn't randomly distributed (Little and Rubin, 2002). This method assumes the missing values are similar to the overall mean, which may not always hold. However, it can be a suitable choice for datasets with a small percentage of missing values and limited time constraints (James et al., 2013).

2.4.7.2 KNN Imputation

k-Nearest Neighbors (kNN) imputation is a widely recognised, model-free approach used for estimating missing values in datasets (Zhang, 2012). It operates on the premise that similar instances in the dataset are likely to have similar values, making it particularly useful when there is no prior assumption about the data distribution (Zhang, 2012). By leveraging the k nearest neighbours of a given instance, missing data can be imputed based on the observed values of those neighbours (Choudhury and Kosorok, 2020). The method typically employs two rules: the majority rule for categorical variables and the mean rule for continuous variables (Zhang, 2012). This simplicity and effectiveness have made kNN imputation a popular technique in various real-world applications (Zhang, 2012).

Basic kNN Imputation Method

The standard procedure of kNN imputation involves three main steps. First, for any instance with missing values, the algorithm calculates the distance between this instance and all others in the dataset (Jadhav et al., 2019). The choice of distance metric, such as Euclidean or Minkowski distance, plays a critical role in determining the accuracy of the imputation (Jadhav et al., 2019). The Euclidean distance between two instances in an n-dimensional space is given as follows (Jadhav et al., 2019) :

$$d(z_i, z_j) = \sqrt{\sum_{k=1}^{n} (z_{ik} - z_{jk})^2}$$

where $d(z_i, x_j)$ is the distance between instances i and j, n is the number of features, and z_{ik} and z_{jk} are the values of the k-th feature for instances i and j, respectively (Choudhury and Kosorok, 2020).

Once the distances are calculated, the algorithm identifies the k-nearest neighbours based on the chosen metric. Finally, missing values are estimated by aggregating the values of these neighbours (Jadhav et al., 2019). For continuous variables, the mean is often used, while the mode is used for categorical variables. The general formula for imputing a missing value is (Jadhav et al., 2019):

$$z_{\text{missing}} = \frac{1}{k} \sum_{i=1}^{k} z_{\text{neighbor}_i}$$

This simple yet effective method has been proven to outperform traditional single imputation techniques, such as mean and median imputation, especially in datasets with a high percentage of missing values (Jadhav et al., 2019).

2.4.8 Stratified Sampling

HMC variants encounter computational challenges when handling large datasets. To address this, the research applies a stratified sampling approach to reduce dataset size, allowing BNN models using HMC variants to process data more efficiently while preserving representativeness. This section discusses the definition and methodology of stratified sampling (Subsection 2.4.8.1) along with its advantages (Subsection 2.4.8.2).

2.4.8.1 Definition and Methodology

Stratified sampling involves dividing the entire population into distinct subgroups known as strata based on shared characteristics. Once these strata are established, researchers randomly select samples from each stratum. This contrasts with simple random sampling, where every individual has an equal chance of being assigned without regard to subgroup characteristics (Elfil and Negida, 2017).

The total sample size n is determined based on the desired confidence level and margin of error, and can be calculated using the formula (Elfil and Negida, 2017):

$$n = \frac{Z^2 \cdot q \cdot (1 - q)}{F^2} \tag{2.19}$$

where:

- Z = Z-score (the number of standard deviations a data point is from the mean),
- q = estimated proportion of the population,
- F = margin of error .

The sample size for each stratum, n_h, can be calculated using proportional allocation (Elfil and Negida, 2017):

$$n_k = \frac{N_k}{N} \cdot n \tag{2.20}$$

where:

- n_k = sample size for stratum k,
- N_k = population size of stratum k,
- N = total population size,
- n = total sample size.

2.4.8.2 Advantages of Stratified Sampling

Stratified sampling offers several advantages over simple random sampling:

- **Increased Precision**: By ensuring that all relevant subgroups are represented, stratified sampling leads to more precise estimates of population parameters (Setia, 2016).
- **Enhanced Comparability**: Researchers can analyze differences between strata, which helps understand how various factors influence outcomes (Elfil and Negida, 2017).
- **Reduced Sampling Error**: Stratified sampling can reduce overall sampling error compared to simple random sampling, especially when there are significant differences between strata (Setia, 2016).

2.4.9 Feature Selection

This section discusses three methods for feature selection utilised in this research: Recursive Feature Elimination (RFE) (Subsection 2.4.9.1), Lasso Regression (Subsection 2.4.9.2), and Random Forest Regressor (Subsection 2.4.9.3).

2.4.9.1 Recursive Feature Elimination

Recursive Feature Elimination (RFE) is a wrapper method that recursively removes the least important features based on model performance (Altartouri, 2021). It begins with all features, fits the model, and then eliminates the least significant ones until the desired number of features is reached, ensuring that only the most relevant features contribute to the predictive power of the model (Van de Schoot and Miocević, 2020).

2.4.9.2 Lasso (Least Absolute Shrinkage and Selection Operator) Regression

Lasso Regression is a linear regression technique that incorporates L1 regularisation, which penalises some coefficients to zero (Kumari, 2024). This regularisation encourages sparsity in the feature set, effectively shrinking some coefficients to zero, thus selecting only the most important features while discarding others (Kumari, 2024).

2.4.9.3 Random Forest Regressor

Random Forest Regressor is an ensemble learning method that builds multiple decision trees and merges their outputs (Ndirangu, 2023). It assesses feature importance by measuring how much each feature contributes to the overall model's accuracy. Features that provide greater predictive power across the trees are deemed more important, allowing for the effective selection of significant variables (Ndirangu, 2023).

Chapter 3

Methodology

3.1 Introduction

This chapter outlines the research methodology employed to assess the performance of various HMC variants, specifically the NUTS and SGHMC, in comparison to the standard HMC method. The methodology encompasses a description of the dataset (Section 3.2), the exploratory data analysis (Section 3.3), data pre-processing steps (Section 3.4), model setup and implementation (Section 3.5), performance evaluation metrics (Section 3.6), the process for comparing the models in terms of computational efficiency and scalability (Section 3.7), and the hardware and computational resources used (Section 3.8). All implementations were carried out using the Python programming language.

3.2 Data Description

This section provides an overview of the California Housing Prices dataset (Nugent, 2017) used in the prediction task. This dataset is publicly available and can be accessed on the Kaggle website.

3.2.1 California Housing Prices Dataset

The dataset pertains to houses located in a given Californian district and includes summary statistics based on the 1990 census data. It contains 20,640 entries, comprising nine predictor variables and one target variable (the median house value). Due to its large size, the dataset may pose computational challenges for the HMC variant models. To address this, only a small sample of 5,000 rows was utilised for model fitting, obtained through stratified sampling. The predictor variables include factors such as the total number of rooms, longitude, median age of housing, and latitude, among others. Table 3.1 provides a summary of these variables:

Table 3.1: Variables in the California Housing Prices Dataset

Variable Name	Data Type	Description
Longitude	Float	Westward extent of a house
Latitude	Float	The northward extent of a house
Housing Median Age	Float	The house's median age in a block
Total rooms	Integer	Total no. of rooms in a block
Total bedrooms	Integer	Total no. of bedrooms in a block
Population	Integer	Total no. of individuals living within a block
Households	Integer	Total number of households per block
Median income	Float	Median income for Households ($)
Median house Value	Float	Median house value for households ($)
Ocean proximity	String	Location of the house to the ocean/sea

This dataset uses census block groups, which are the smallest geographical units for which the U.S. Census Bureau publishes sample data (learn Developers, 2018). Each block group typically contains a population of 600 to 3,000 people, and the data aggregates information such as population, total rooms, median income, and house values for all households within that group (Wang, 2018). These block groups provide a localised level of granularity for analysis, summarising trends across communities rather than individual households.

3.3 Exploratory Data Analysis

This section presents the exploratory data analysis conducted in this research, including a dataset overview (Subsection 3.3.1), summary statistics (Subsection 3.3.2), feature distributions (Subsection 3.3.3), and an analysis of relationships between features (Subsection 3.3.4).

3.3.1 Data Overview

This research utilised the California Housing Prices dataset, to perform regression using BNNs and three HMC variants. Table 3.2 below gives a summary of the data:

Table 3.2: Summary of Dataset

Dataset	No. of rows	No. of features
California Housing Prices dataset	20,640	10

3.3.2 Summary Statistics

Summary statistics were calculated to examine the distribution of variables in terms of mean, standard deviation, and quartiles. Tables 3.3 and 3.4 display these summary statistics.

Table 3.3: Summary Statistics of the California Housing Prices Dataset Features (I)

	longitude	latitude	house_med_age	tot_rooms	tot_bedrooms
std	2.003532	2.135952	12.585558	2181.615252	421.385070
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000
min	-124.350000	32.540000	1.000000	2.000000	1.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000

Table 3.4: Summary Statistics of the California Housing Prices Dataset Features (II)

	population	households	median_income	median_house_value
std	1132.462122	382.329753	1.899822	115395.615874
mean	1425.476744	499.539680	3.870671	206855.816909
25%	787.000000	280.000000	2.563400	119600.000000
50%	1166.000000	409.000000	3.534800	179700.000000
75%	1725.000000	605.000000	4.743250	264725.000000
min	3.000000	1.000000	0.499900	14999.000000
max	35682.000000	6082.000000	15.000100	500001.000000
count	20640.000000	20640.000000	20640.000000	20640.000000

Key insights:
The summary statistics of the dataset in Tables 3.3 and 3.4 reveal several key insights into the features, particularly focusing on the target variable (median house value).

- The dataset comprises 20,640 observations. The standard deviation is $115,395.62, indicating considerable variability. The mean median house value is $206,855.82.

- The minimum and maximum values of the median house value are $14,999 and $500,001, respectively.
- The median house value at the 25th percentile is $119,600, the 50th percentile (median) is $179,700, and the 75th percentile is $264,725.
- Other features such as longitude and latitude show mean values of -119.57 and 35.63, respectively. Housing median age averages 28.64 years, with the number of rooms and bedrooms averaging 2,635.76 and 537.87, respectively.
- Population and households have means of 1,425.48 and 499.54. The average median income in the dataset is $3.87 (in tens of thousands).

Overall, these statistics provide a comprehensive overview of the distribution and central tendencies of the features in the dataset, essential for understanding the factors influencing the median house value.

3.3.3 Distributions of the Features

Two plots were used to explore the variable distributions within the dataset: a bar chart for the categorical variable and histograms for the numerical variables.

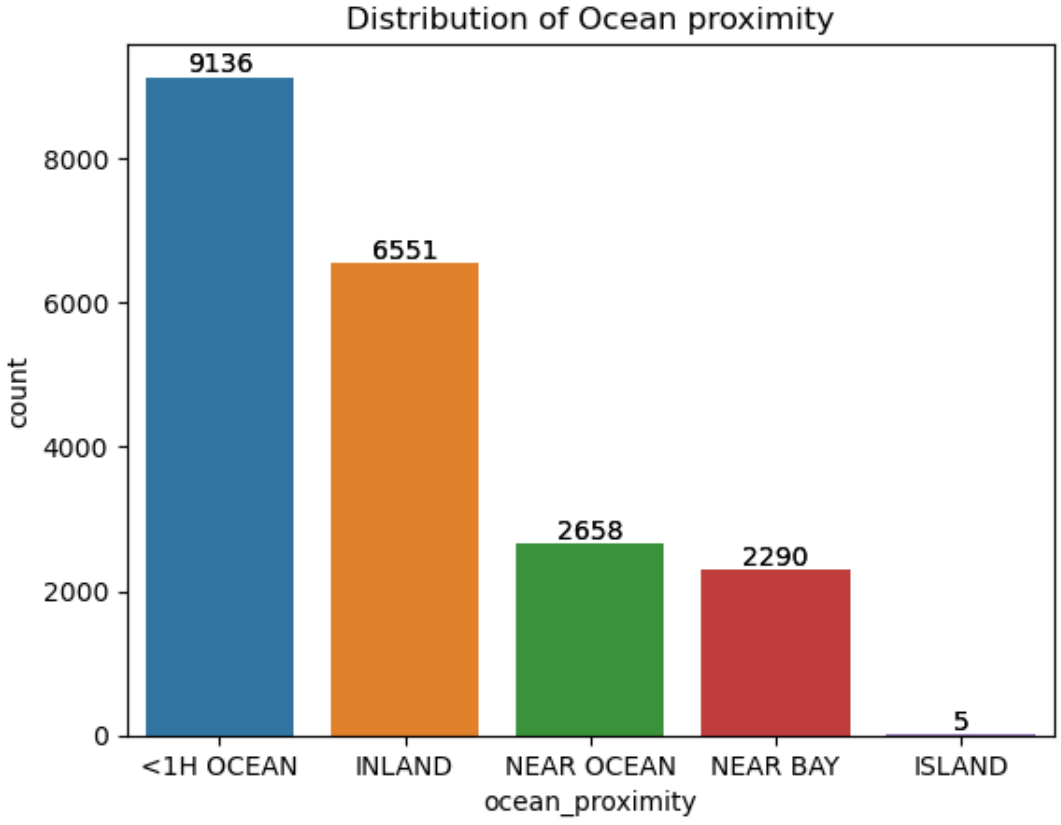


Figure 3.1: Bar Chart of the Ocean Proximity Variable

The bar chart in Figure 3.1 displays the distribution of the ocean proximity feature. It shows that 44.26% (9,136) of the population lives less than 1 hour from the ocean, 31.74% (6,551) live inland, 12.88% (2,658) live near the ocean, 11.09% (2,290) live near a bay, and 0.02% (5) live on an island.

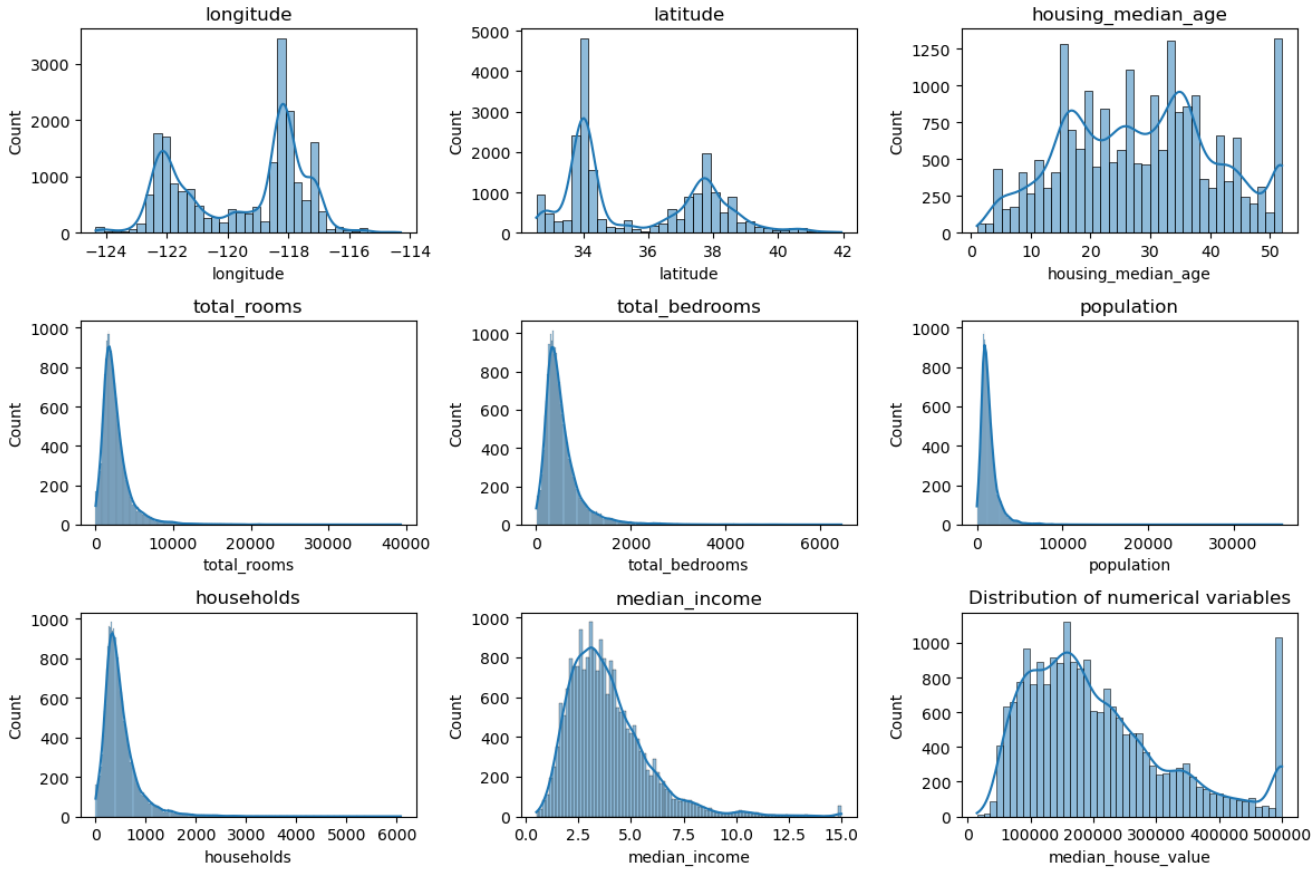


Figure 3.2: Histogram of Numerical Variables

Figure 3.2 shows the distributions of the numerical variables in the dataset. The longitude and latitude histograms are multimodal, indicating the presence of multiple geographic clusters or regions within the dataset. The median house age histogram is also multimodal, indicating that there are several distinct age groups or periods during which houses were commonly built. The histograms of the population, total bedrooms, total rooms and households are right-skewed, indicating that most properties have lower counts for these variables, with a smaller number of houses having significantly higher values. The median house value variable histogram is skewed to the right, with the majority of houses having median values between $100,000 and $300,000, and only a few houses exceeding $500,000. This indicates that most properties are concentrated in the lower to mid-range value segments, while higher-value properties are less common and represent a smaller portion of the dataset.

3.3.4 Relationship Between Features

A correlation heatmap was used to illustrate the strength of relationships between variables in the dataset. Bright red blocks indicate strong positive correlations, while light red (peach) blocks represent weaker positive correlations. Similarly, dark blue blocks denote strong negative correlations, and light blue blocks reflect weaker negative correlations. Variables with correlations close to 1 or -1 are strongly correlated, whereas those near zero show weak or no correlation.

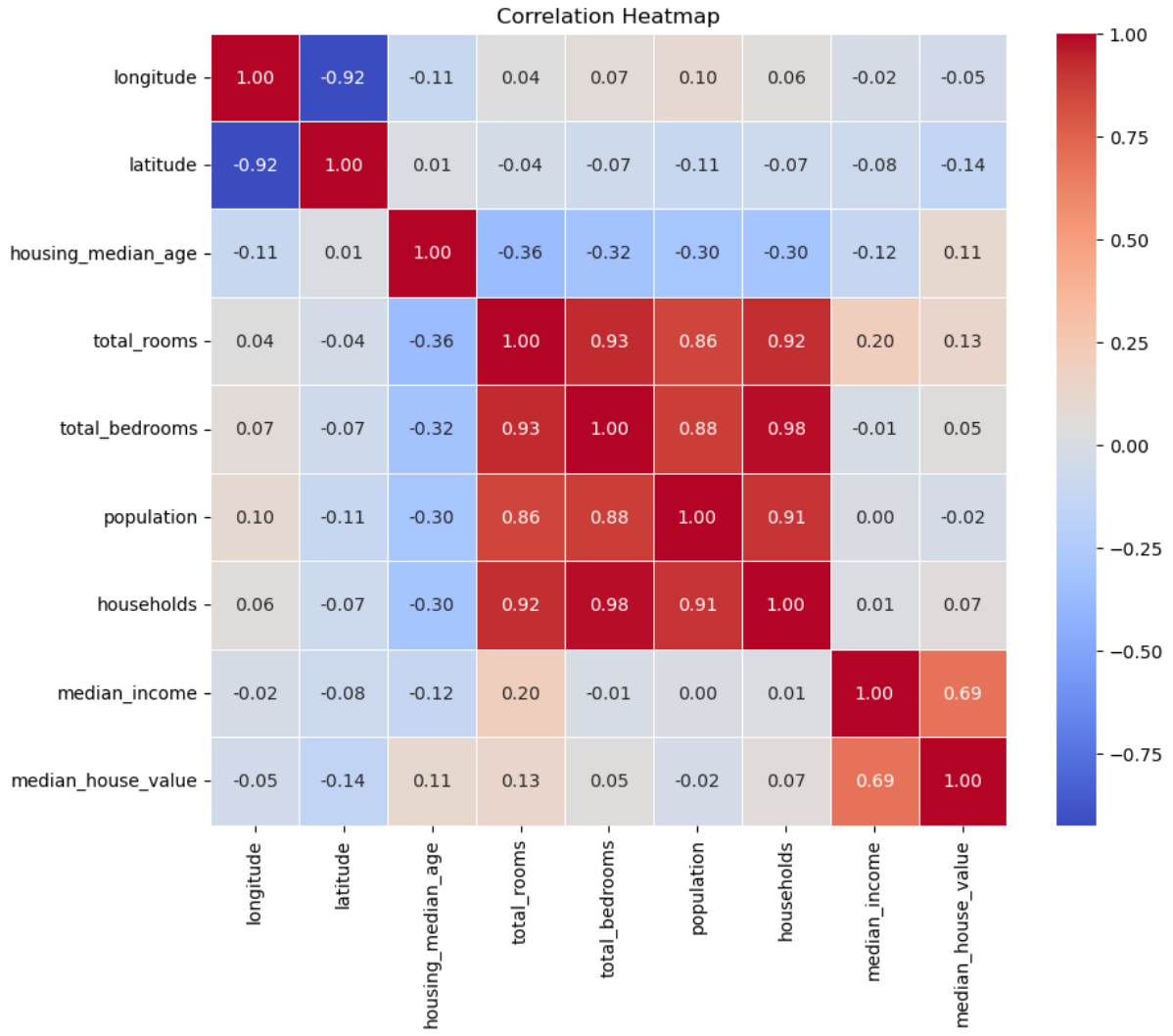


Figure 3.3: Correlation Heatmap of Features

The correlation heat map in Figure 3.3 highlights the relationships between various features in the dataset. Longitude and latitude exhibit a strong negative correlation of -0.925, indicating that as the longitude increases, the latitude decreases. Housing median age shows a moderate negative correlation with total rooms, total bedrooms, population, and households, suggesting that newer houses have more rooms, bedrooms, and residents. Total rooms and total bedrooms have a high positive correlation of 0.930, indicating that houses with more rooms also tend

to have more bedrooms. Similarly, total bedrooms and population (0.878) and total rooms and population (0.857) are strongly positively correlated, reflecting that larger houses tend to accommodate more people. Households also have a very high positive correlation with total rooms (0.918) and total bedrooms (0.980), suggesting that larger households are associated with larger houses. Median income shows a moderately positive correlation with median house value (0.688), indicating that higher incomes are associated with higher house values. Conversely, there is no significant correlation between longitude, latitude, and median income.

3.4 Data Pre-Processing

This section discusses the data pre-processing that was conducted to ensure the dataset was ready for modelling. It discusses several key steps, including data cleaning (Subsection 3.4.1), data transformation (Subsection 3.4.2), feature engineering (Subsection 3.4.3), feature selection (Subsection 3.4.4), dataset reduction (Subsection 3.4.5) and data standardisation (Subsection 3.4.6).

3.4.1 Data Cleaning

3.4.1.1 Handling Duplicates and Missing Values

Data completeness is essential for the accuracy of the models. The dataset was first checked for duplicate entries and missing values.

Duplicates

The dataset was examined for duplicates using the *duplicated* function (Pandas-Authors, 2024), and none were found. As a result, no rows needed to be removed due to duplication.

Missing Values

We assessed the datasets for any missing values using the Pandas *isnull* function (Pandas, 2024) from the Pandas library and used imputation techniques to handle missing values. Only the 'total bedrooms' variable had missing values, with 207 missing entries out of 20,640 rows. No other feature contained missing values.

3.4.1.2 Data Imputation

Missing values in key features could affect the accuracy of model predictions. Therefore, appropriate imputation techniques were applied to avoid losing valuable information. Data imputation techniques such as mean imputation and kNN were used to fill in missing values in the total bedrooms variable.

- Mean Imputation: Replaced absent values with the mean of the existing values in the total bedrooms variable.
- kNN Imputation: Imputed missing values by considering the most similar rows (neighbours) based on other variable values.

Table 3.5 shows the changes in the standard deviation and mean of the total bedrooms variable resulting from each data imputation method used:

Table 3.5: Results of Data Imputation

	Original data	Mean imputed data	KNN imputed data
Mean	537.87	537.87	537.57
Standard deviation	421.39	419.27	420.41

Both mean imputation and kNN imputation preserve the mean value quite well, with mean imputation exactly matching the original data's mean. The standard deviation for kNN imputed data (420.41) is closer to the original data's standard deviation (421.39) compared to the mean imputed data (419.27). This suggests that kNN imputation better preserves the variability of the original dataset.

Figure 3.4 below shows the changes in the standard deviation and mean of the total bedrooms variable resulting from each data imputation method used: kNN and mean imputation.

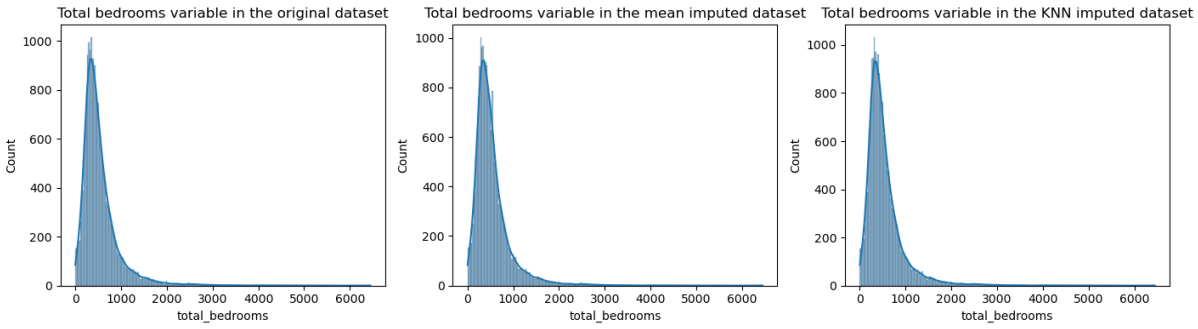


Figure 3.4: Differences in the Distribution of the Total Bedrooms Variable After Imputation

A comparison of the results from both methods was conducted:

- Mean Imputation: Maintained the mean value but introduced an artificial spike at the imputed value, disrupting the natural distribution of the feature.
- kNN Imputation: Preserved the variability of the original data more effectively, maintaining a smoother, more natural distribution without creating artificial spikes.

Based on these results, kNN imputation was selected for handling missing values, as it better preserved the feature's distribution and variability, which is crucial for accurate Bayesian analysis.

3.4.1.3 Detecting and Handling Outliers

Outliers can distort the learning process in neural networks, especially in regression models. Boxplots were used to visually identify outliers in the numerical features. Figure 3.5 illustrates the presence of outliers in some features of the dataset. The boxplots provide a visual summary of the data distribution for each feature, highlighting key statistics such as the median, interquartile range (IQR), and the presence of extreme values. Outliers are depicted as black dots, indicating data points that fall significantly outside the expected range defined by the lower and upper quartiles. Specifically, these outliers are values that are either above the upper quartile plus 1.5 times the IQR or below the lower quartile minus 1.5 times the IQR.

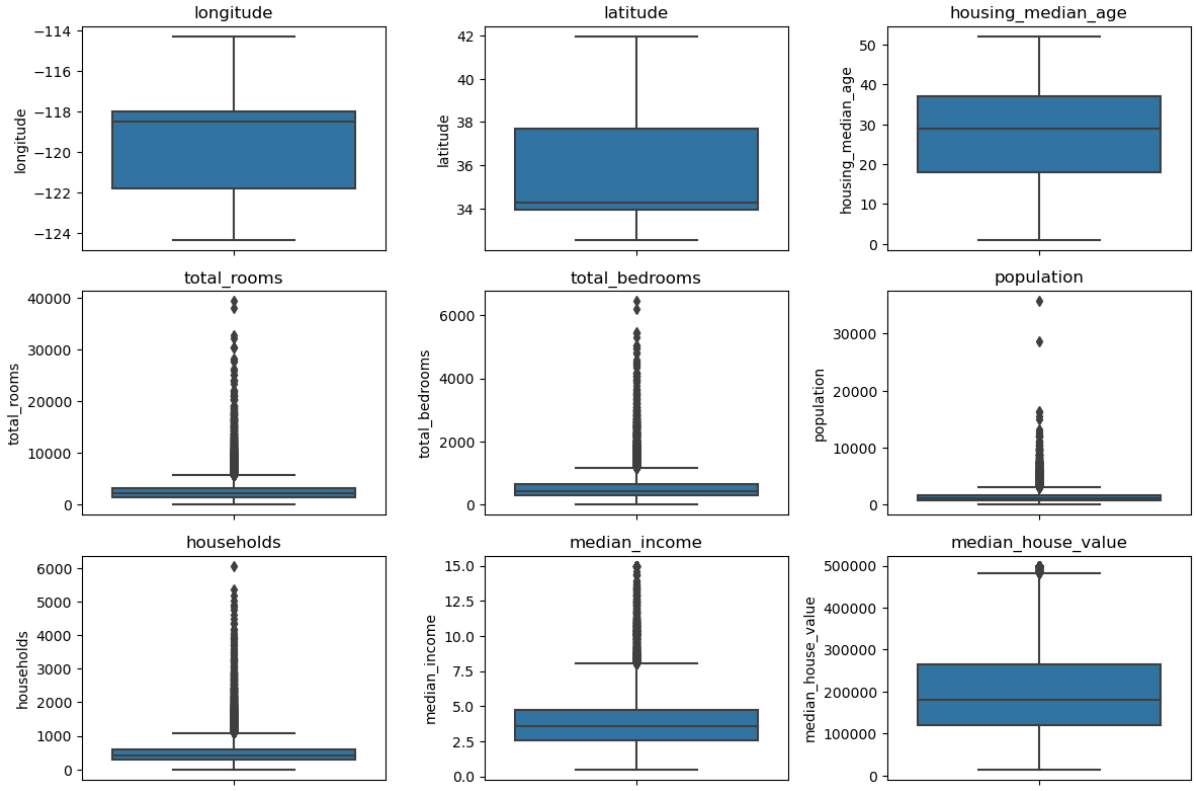


Figure 3.5: Boxplot of the Features

The Z-score test (with a threshold of 3 standard deviations) was used to identify the most extreme values in each feature. Figure 3.5 indicates no outliers in latitude and longitude values, which is expected since these features only represent the geographical locations of the houses. The Z-score test also shows no extreme outliers in the housing median age and median house value features.

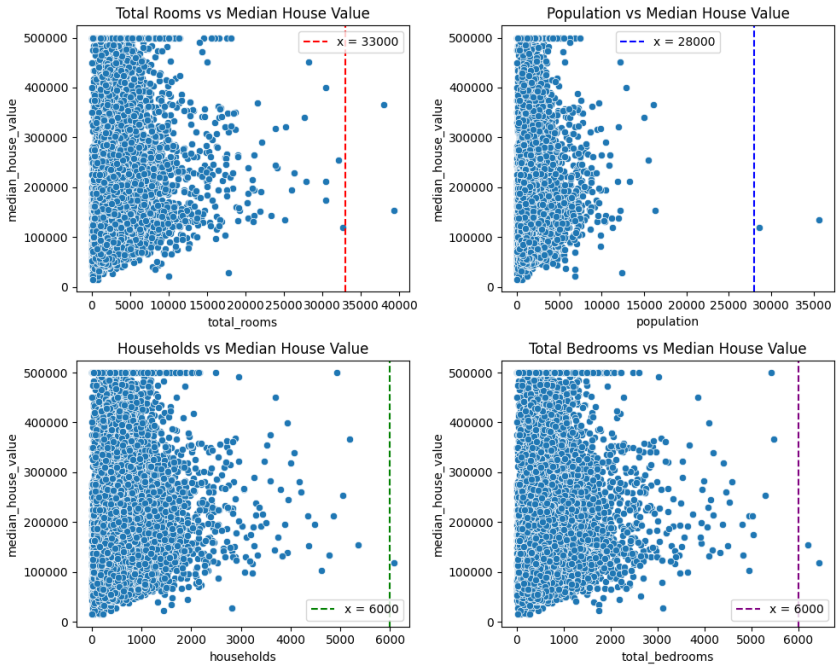


Figure 3.6: Scatterplots of the Variables with Outliers

Figure 3.6 shows the outliers detected by the Z-score test for the total rooms, population, households and total bedrooms variables. The outliers are as follows:

1. Total Rooms: Blocks with a total number of rooms of around 33,000 or more were identified as outliers.
2. Total Bedrooms: Blocks with a total number of bedrooms exceeding approximately 6,000 were identified as outliers.
3. Population: Blocks with a population of about 28,000 or more were identified as outliers.
4. Households: Blocks with a total number of households surpassing approximately 6,000 were identified as outliers.

Despite the presence of outliers, they were retained in the dataset because they likely represent real and significant variations in housing patterns, such as highly populated blocks with larger houses. Removing these outliers could risk losing valuable information about housing clusters. Outliers in Bayesian models are handled through the probabilistic nature of the model. Instead of removing them, we allow the model to account for the uncertainty these outliers introduce, providing a more nuanced analysis of the data.

3.4.2 Data Transformation

We standardised the numerical features using Min-Max scaling to ensure that the model is not disproportionately influenced by features with varying scales.

The data was rescaled using the Min-Max scaler to fit into a certain range, usually [0, 1]. The Min-Max formula is given as:

$$w^{'} = \frac{w - min(w)}{max(w) - min(w)} \tag{3.1}$$

where the scaled data is $w^{'}$, and the data to be scaled is w.

The scikit-learn library *Min-Max Scaler* function was used to perform data standardisation in Python (Scikit, 2024).

3.4.3 Feature Engineering

Two feature engineering techniques were applied to improve the model's predictive power. These are the label encoding and log-transformation techniques. The label encoding technique converted categorical (ordinal) features into numerical values. This transformation was achieved using the *LabelEncoder* function from the scikit-learn library (Scikit-learn, 2024). The highly skewed target variable was normalised using a log-transformation technique (*np.log()* function).

3.4.3.1 Label Encoding of Ocean Proximity

The ocean proximity variable was transformed using label encoding to enhance the model's ability to process categorical data. This technique assigns unique integer values to each category based on their proximity to the ocean.

Mapping Categories to Numeric Values: The ocean proximity variable, which contains categorical data representing different proximity levels to the ocean, was converted into a numerical format using the following mapping in Table 3.6:

Table 3.6: Mapping of Ocean Proximity Categories to Numeric Values

Category	Numeric Value
<1H OCEAN	0
INLAND	1
NEAR OCEAN	2
NEAR BAY	3
ISLAND	4

Label encoding is particularly effective for ordinal data, as it preserves the natural order among the categories. This transformation enables the BNNs to leverage the 'ocean proximity' feature during training, improving predictive performance.

3.4.3.2 Log-Transformation of Target Variable

The target variable (median house value) exhibited a right-skewed distribution with a peak at 500,000. To make the data more normally distributed, a log transformation was applied. This transformation reduces skewness and improves the stability and performance of the Bayesian models by ensuring that the data better satisfy the assumptions of normality

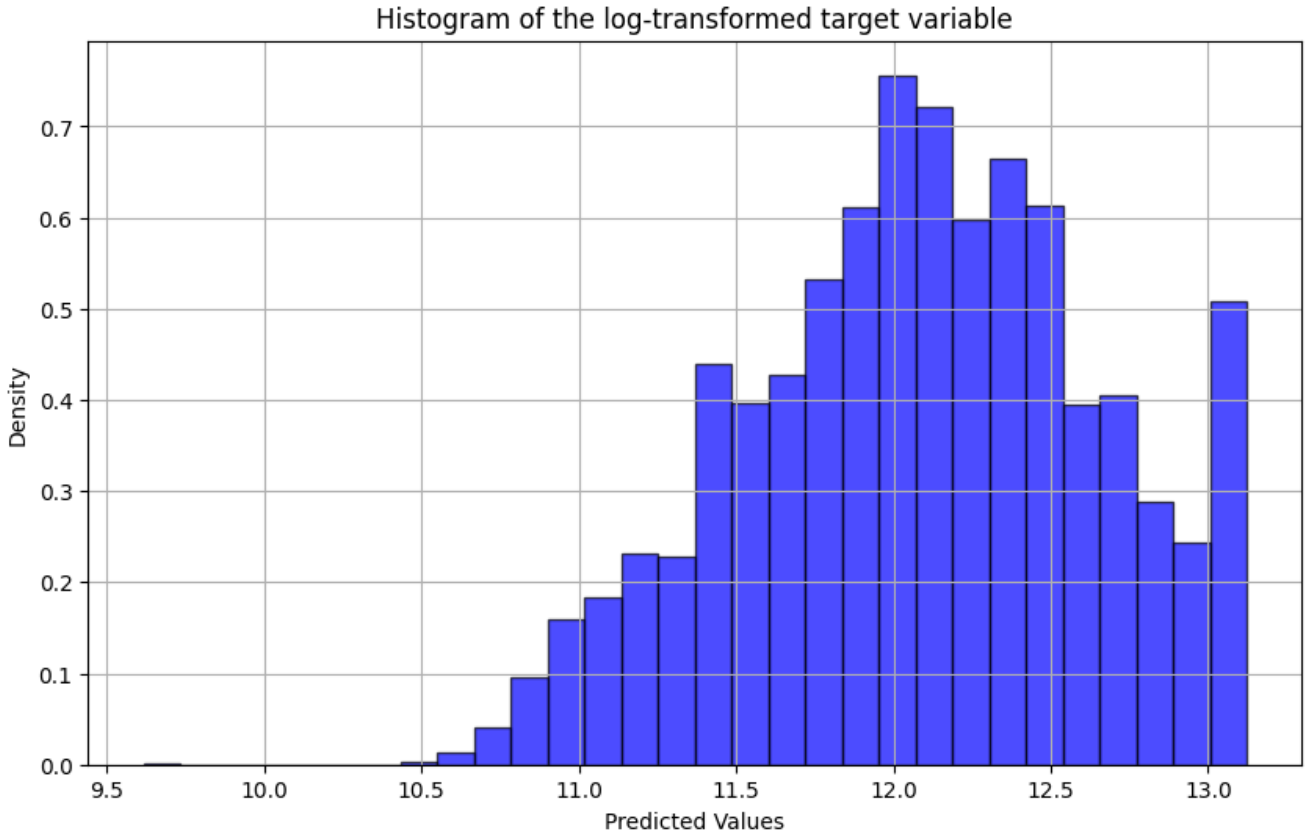


Figure 3.7: Histogram of the Log-Transformed Target Variable

Figure 3.7 demonstrates that applying the log transformation to the target variable results in a distribution closer to normal. After the transformation, the distribution exhibits a leftward skew, in contrast to the original rightward skew observed before the transformation in Figure 3.2.

The target variable was log-transformed to address its skewness and large scale, ensuring that regression models met the assumption of normality in residuals, which improved model accuracy and interpretability. The independent variables, such as total rooms and population, were

left untransformed because their raw scales were interpretable, and their relationships with the target variable did not necessitate transformation.

3.4.4 Feature Selection

Feature selection is crucial in BDL to reduce the dimensionality of the input space, improving both model interpretability and computational efficiency, especially when using HMC-based methods. Three feature selection methods were employed to identify the most significant predictors of the median house value. This research utilised Recursive Feature Elimination, Lasso Regression, and the Random Forest Regressor to determine the key features influencing the predictions.

1. Recursive Feature Elimination (RFE): Features were chosen by iteratively eliminating the least significant ones and then assessing the model's performance again.

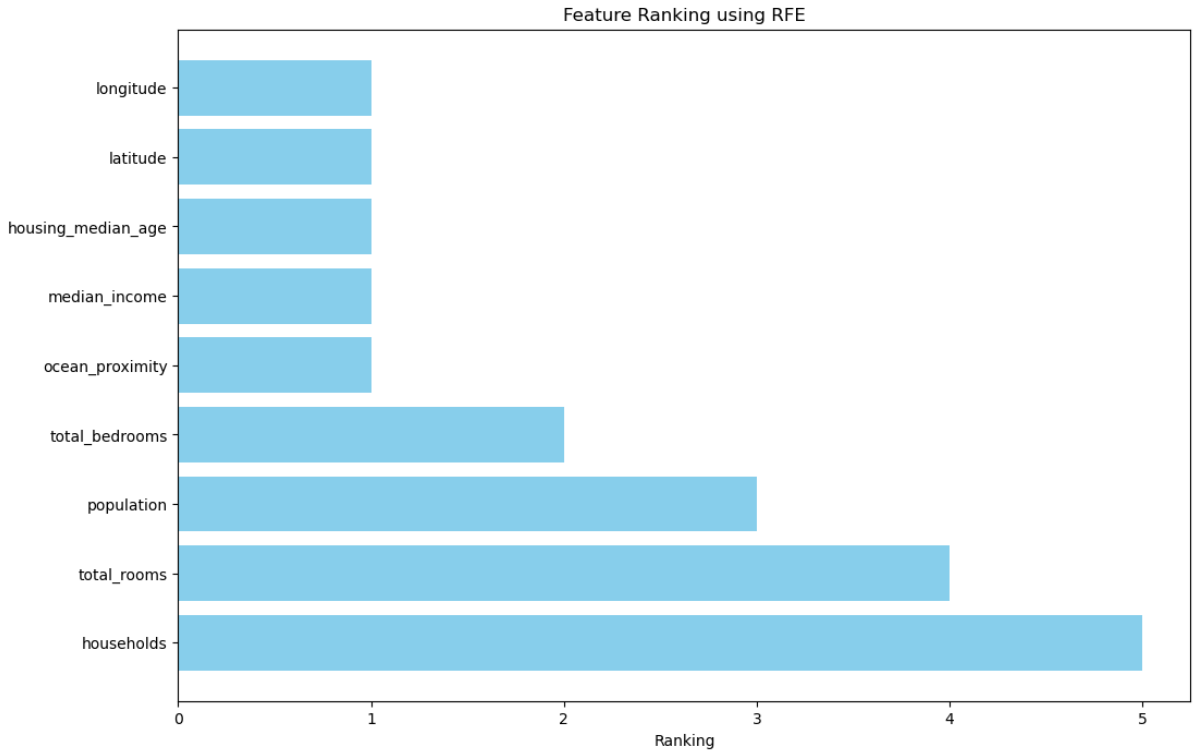


Figure 3.8: Feature Importance Using the RFE Selection Method

Figure 3.8 shows that the RFE method selected the latitude, longitude, housing median age, ocean proximity and median income as the top five most important variables for predicting the median house value.

2. Lasso Regression: Penalised the coefficients of less important features, effectively shrinking them to zero.

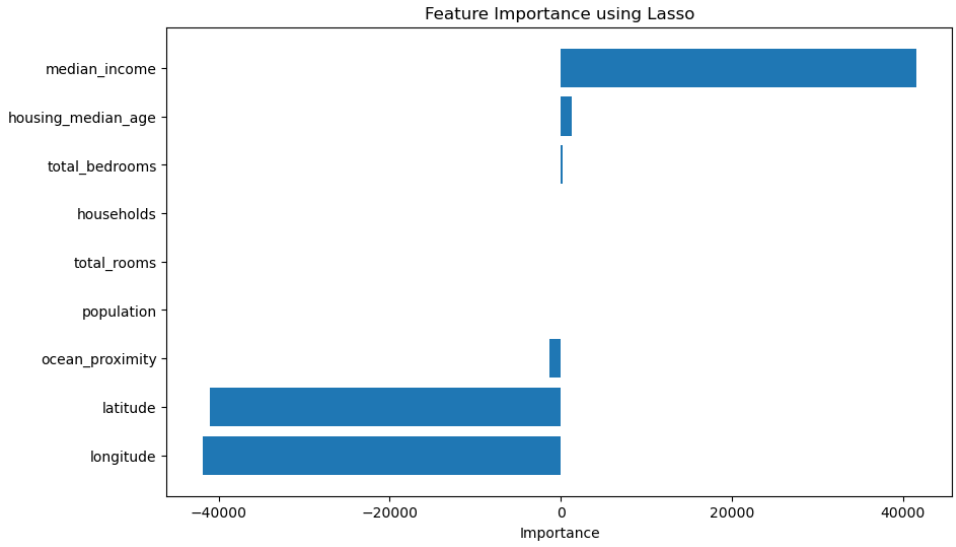


Figure 3.9: Feature Importance Using the Lasso Selection Method

Figure 3.9 shows that the lasso regression method chose the total bedrooms, median income, latitude, ocean proximity, housing median age and longitude as the most important variables for predicting the median house value.

3. Random Forest Regressor: Used the mean decrease in impurity to rank the importance of each feature.

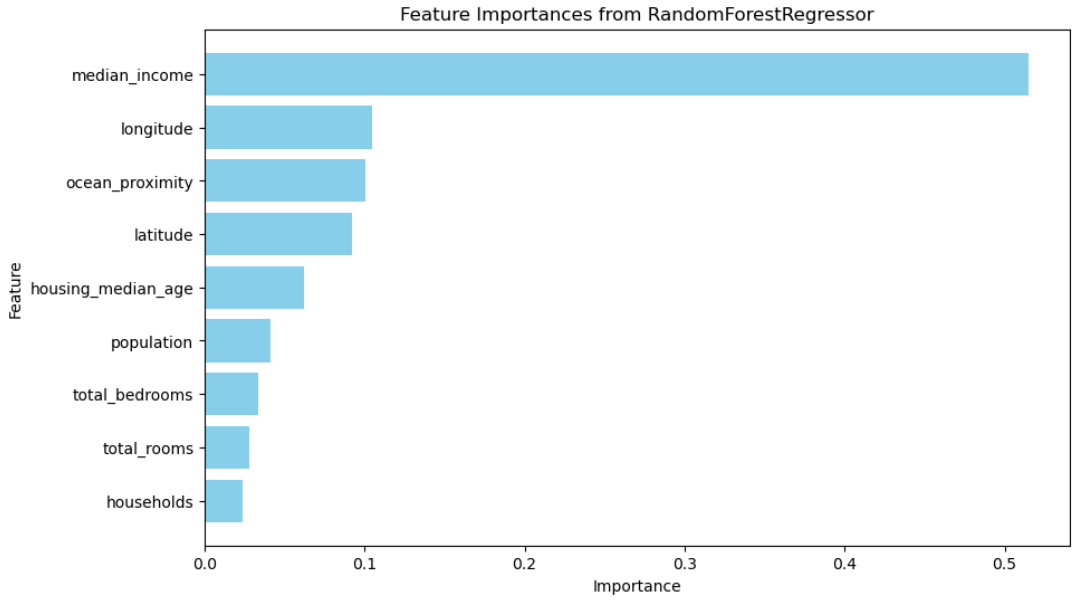


Figure 3.10: Feature Importance Using the Random Forest Regressor Selection Method

Figure 3.10 shows that the random forest regressor chose the median income, longitude, ocean proximity, latitude and housing median age as the five most important features for predicting the median house value.

All three methods indicated that the following features were the most important: median income, longitude, latitude, housing median age and ocean proximity. These five features were retained for model training. This reduced feature set helps minimise model complexity while retaining predictive power, which is particularly important in Bayesian models where overfitting can be a concern.

3.4.5 Dataset Reduction via Stratified Sampling

Given the computational challenges associated with HMC variants when handling large datasets, a stratified sampling approach was employed to reduce the dataset size from 20,640 rows to 5,000 rows. This step was performed before feature selection to ensure the models could efficiently handle the dataset while retaining representative data.

- Stratification Feature: The dataset was stratified based on the ocean proximity feature. This categorical variable contains five classes (ISLAND, NEAR BAY, NEAR OCEAN, INLAND, and <1H OCEAN), and stratification ensures that the sampled data maintains the same proportional representation of each category as in the original dataset.
- Stratified Sampling Method: The train test split function from the scikit-learn library was used to sample 5,000 rows while preserving the distribution of ocean proximity. This ensures that the selected subset of data is representative of the overall population.

By reducing the dataset to 5,000 rows, the models can be trained more efficiently while ensuring that the essential patterns in the data, particularly those related to ocean proximity, are retained.

3.4.6 Data Standardisation

The final preprocessing step involved standardising the feature set. Standardisation is important in BNN because it prevents features with bigger ranges from overly influencing the posterior estimates during the inference process. This is particularly relevant when using gradient-based methods like HMC, where the input data's scale affects the algorithm's efficiency.

The Min-Max normalisation technique was used to scale the selected predictor features to the range [0, 1]. This step was necessary to ensure that all features, especially those with different units of measurement had the same influence on the model.

3.5 Model Setup and Implementation

We used the BNN discussed in Section 2.4.4 with a linear AF in the output layer. This method allows for estimating uncertainty in the predictions, providing a distribution of potential outcomes as opposed to a single-point estimate. Linear activation functions are particularly well-suited for BNNs in this context because they preserve the distribution of outputs from the network due to the weights' inherent uncertainty (Neal, 2012). Since BNNs learn weight distributions during training, a non-linear activation could introduce non-linearities that mask this uncertainty. A linear activation, acting like a window allows the uncertainty to flow through to the final prediction, providing valuable information about the model's confidence in its estimates (Neal, 2012).

The three HMC variants discussed in Section 2.4.5 were used to estimate the posterior density distribution. The variants are the standard HMC, NUTS and SGHMC. These algorithms were implemented using Python libraries such as PyMC (PyMC, 2024) and TensorFlow Probability (Tensorflow, 2024), which provide robust tools for BI. The following sections detail the data splitting strategy (Subsection 3.5.1), model descriptions (Subsection 3.5.2), and architectural design for each model (Subsection 3.5.3).

3.5.1 Data Splitting

To assess the performance of the BNN models accurately, the dataset was split into training and testing subsets with an 80-20 ratio. This ensures that the model is trained on a majority of the data while retaining a portion for validating its performance on unseen data.

- Training Set: Eighty percent of the data was allocated for model training.
- Test Set: The other twenty percent was set aside to evaluate the model's performance on new data

The training set comprised 4,000 rows, while the test set included 1,000, allowing for a robust model performance evaluation. This separation is crucial for assessing how well the model generalises to new, unseen data.

3.5.2 Model Description

Each model utilised a different variant of HMC for posterior inference: the BNN-HMC used the standard HMC algorithm, while the BNN-NUTS utilised the NUTS, both implemented with the PyMC library. The BNN-SGHMC model used the SGHMC algorithm and was developed using TensorFlow and TensorFlow Probability.

3.5.3 Model Architecture

3.5.3.1 BNN-HMC and BNN-NUTS Architecture

The architectures for the BNN-HMC and BNN-NUTS models were identical. The BNNs using HMC and NUTS are implemented in PyMC, with two hidden layers parameterised by normally distributed biases and weights with standard deviation 1 and mean 0 ($N \sim (0, 1)$), followed by an output layer. Each hidden layer has weights and biases drawn from normal distributions ($N \sim (0, 1)$), with 10 units in the first layer connected to the input and 10 units in the second layer connected to the first layer. The output layer produces a linear combination of the second hidden layer's outputs. The predicted values follow a normal distribution with a mean given by the output layer and a half-normal distribution for the standard deviation, modelling the observed data.

3.5.3.2 BNN-SGHMC Architecture

The BNN-SGHMC model utilised an architecture specifically designed for its implementation in TensorFlow. The BNN is implemented as a subclass of tf.keras.Model, with two hidden layers, each having 64 units and ReLU activations, followed by an output layer with a single unit and a linear activation, making it suitable for regression tasks.

3.6 Performance Evaluation

We assessed the performance of the three HMC variants using functions from both the scikit-learn and numpy libraries (Numpy, 2024): *mean_squared_error* and NumPy's *sqrt* for the RMSE, as well as *mean_absolute_error*. The results were presented in comparative tables.

3.7 Comparative Analysis

3.7.1 Computational Efficiency

To evaluate the computational efficiency of each HMC variant, the training time was systematically measured using Python's time module. This involved capturing the total time to train the BNN model for each HMC variant, from the start of sampling to its completion. Specifically, the time.time() function was used to record the time before the sampling process began and immediately after it concluded. The difference between these timestamps provided the total training duration in seconds.

3.7.2 Scalability

To investigate the scalability of each HMC variant, we evaluated model performance as the dataset size increased. This analysis involved using subsets of the reduced dataset, representing 25%, 50%, and 75% of the original data, to train and assess the BNN models. Each subset was generated using the *sample()* function in Pandas, with a fixed random seed (random state=42) set to ensure reproducibility. The data was divided into training and testing sets for each subset with an 80-20 ratio using the *train test split* function from sklearn. The input features were normalised using the MinMaxScaler to improve model performance and ensure numerical stability. For each subset, we measured the training time and calculated performance metrics—MSE, RMSE, and MAE—to compare the predictive accuracy and computational efficiency of each model at different data sizes.

3.8 Hardware and Computational Resources

For the analysis and computations in this study, Google Colab was used, providing a Python 3 environment powered by the Google Compute Engine backend. The virtual machine configuration included 12.67 GB of RAM and 107.72 GB of disk space. This setup provided sufficient computational power for data processing and model training throughout the research.

Chapter 4

Results and Discussion

This chapter presents the results of the comparative analysis between the HMC variants—standard HMC, NUTS, and SGHMC—in the context of BNNs (Section 4.1). The findings are then interpreted and discussed in Section 4.2, concerning the research aims and objectives, highlighting their implications for practical applications and future research directions.

4.1 Results

The results are presented as follows: Subsection 4.1.1 discusses performance evaluation and computational efficiency, Subsection 4.1.2 presents the posterior predictions generated by the three models, and Subsection 4.1.3 details the outcomes of the scalability experiment.

4.1.1 Performance Evaluation and Computational Efficiency

Two trials were performed, utilising 250 tuning steps in the first trial and increasing this to 500 in the second trial. The primary objectives of the trials were to determine: (1) whether an increase in tuning steps correlates with improved accuracy and (2) which model demonstrates superior computational efficiency. After each trial, metrics including MSE, RMSE, MAE, and model training time were computed to assess the models. Lower training times indicate better computational efficiency, while values close to zero for MSE, RMSE, and MAE signify enhanced prediction accuracy and overall model performance. Table 4.1 shows the results of these two trials.

Table 4.1: Results of the Two Trials

Trial	Model	MSE	RMSE	MAE	Training time (in seconds)
1	BNN-HMC	0.1332	0.3650	0.2819	2232.23
1	BNN-NUTS	0.1333	0.3651	0.2814	1236.63
1	BNN-SGHMC	0.1009	0.3177	0.2415	483.63
2	BNN-HMC	0.1331	0.3649	0.2815	3984.24
2	BNN-NUTS	0.1330	0.3647	0.2814	3848.67
2	BNN-SGHMC	0.0925	0.3042	0.2303	1005.53

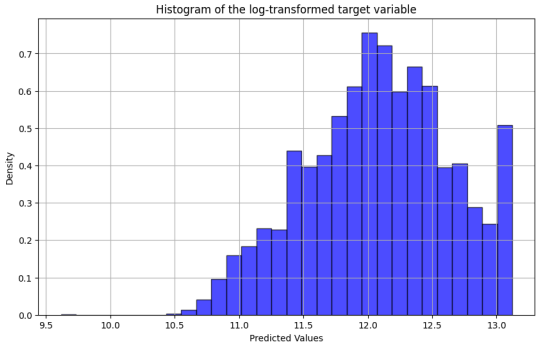
Key Observations:

- In the first trial, the BNN with the SGHMC sampler exhibited the lowest MSE, RMSE, and MAE scores compared to the other two BNN models. The performance of the BNN models with the HMC and NUTS samplers was comparable, with their MSE, RMSE, and MAE scores closely aligned.
- In the second trial, the BNN with the SGHMC sampler again achieved the lowest MSE, RMSE, and MAE scores, reinforcing its superior performance relative to the other models. The NUTS and HMC models continued to show similar performance metrics.
- Notably, increasing the number of tuning steps did not yield significant improvements in model performance across the evaluated metrics. However, the BNN-SGHMC's scores slightly improved with an increase in the number of tuning steps.
- Regarding computational efficiency, the BNN model trained with the SGHMC sampler required less than half the training time compared to the models using NUTS and HMC samplers. The BNN-NUTS also had significantly lower training times compared to the BNN-HMC in both trials.
- The SGHMC model converged more quickly than the other two, achieving lower values for the evaluation metrics, indicating higher prediction accuracy and enhanced computational efficiency.
- The BNN with NUTS and BNN with HMC displayed no significant differences in predictive accuracy. Although the NUTS model had slightly lower metric scores for MSE, RMSE, and MAE on average, these differences were minimal and practically negligible.

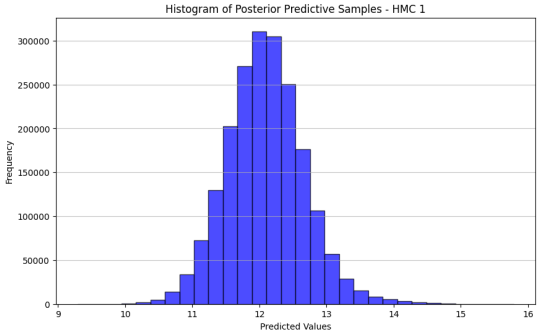
The findings indicate that the BNN with the SGHMC sampler consistently outperformed both the BNN-HMC and BNN-NUTS models when it comes to predictive accuracy, as shown by

lower MSE, RMSE, and MAE scores in both trials. Additionally, the BNN-SGHMC model demonstrated superior computational efficiency, with training times being about more than half that of the other samplers and having quicker convergence, highlighting its potential as the most effective HMC variant for optimising posterior density estimation in BDL models. In contrast, the HMC and NUTS models showed similar performance in terms of predictive accuracy, with only marginal differences in their metrics. However, the BNN-NUTS was found to be more computationally efficient compared to the BNN-HMC model.

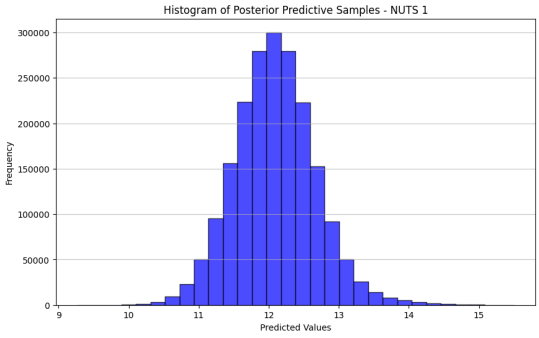
4.1.2 Posterior Predictions for the Three Models



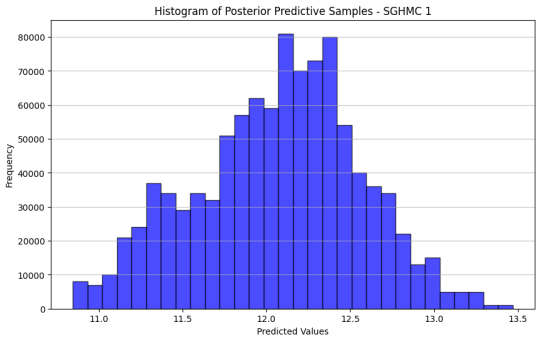
(a) Log-Transformed Target Varaible



(b) BNN-HMC Predictive Posterior Samples



(c) BNN-NUTS Predictive Posterior Samples

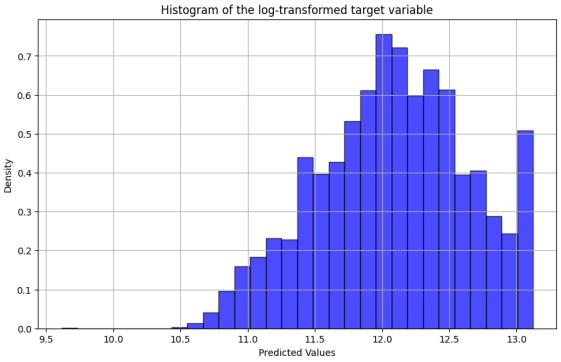


(d) BNN-SGHMC Predictive Posterior Samples

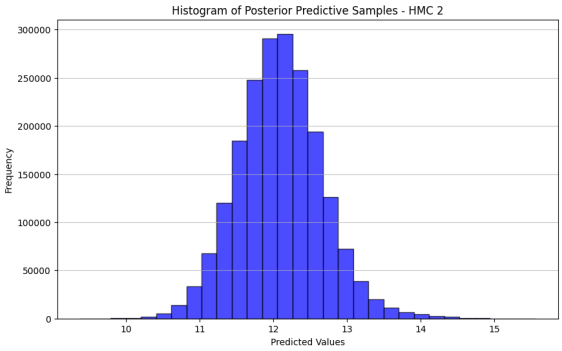
Figure 4.1: Histograms of the Log-Transformed Target Variable and Posterior Predictive Samples for the Three BNN Models (Trial 1)

Figure 4.1 shows the histograms of the log-transformed target variable (median house value) and the posterior predictions for the BNN-HMC, BNN-NUTS and BNN-SGHMC for the first trial with 250 tuning steps. The histogram of the log-transformed target variable shows a distribution with a peak around 12.0 and a range approximately from 10.0 to 13.0. The distribution exhibits a slight leftward skew, with a higher density of values between 11.5 and 12.5. All three

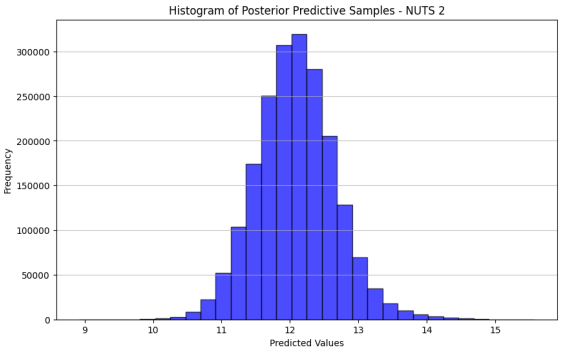
models (BNN-HMC, BNN-NUTS, and BNN-SGHMC) predict distributions with peaks around 12.0, aligning well with the log-transformed target variable. The BNN-HMC and BNN-NUTS models have wider ranges (9.0 to 15.0) and symmetric distributions, capturing more variability. In contrast, the BNN-SGHMC model has a narrower range (11.0 to 13.5) and a slight left-skewed distribution, which closely aligns with the target variable's range and skewness. Essentially, each model shows a good fit, with BNN-SGHMC being most closely aligned to the target variable.



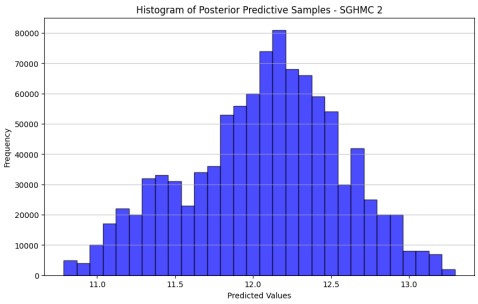
(a) Log-Transformed Target Variable



(b) BNN-HMC Predictive Posterior Samples



(c) BNN-NUTS Predictive Posterior Samples



(d) BNN-SGHMC Predictive Posterior Samples

Figure 4.2: Histograms of the Log-Transformed Target Variable and Posterior Predictive Samples for the Three BNN Models (Trial 2)

Figure 4.2 again shows that all three models—BNN-HMC, BNN-NUTS, and BNN-SGHMC—predict distributions that peak around 12.0, matching the log-transformed target variable. BNN-HMC and BNN-NUTS again have wider, symmetric distributions (9.0 to 15.0), capturing more variability. BNN-SGHMC, with a narrower range (11.0 to 13.5) and slight left-skewness, closely aligns with the target variable's distribution. These results again show that the increase in tuning steps does not significantly improve the predictions of each model.

The histograms indicate that while BNN-HMC and BNN-NUTS exhibit wider and symmetric distributions, capturing greater variability, the BNN-SGHMC model has a narrower range and a slight left-skewed distribution that closely matches the target variable's characteristics. This suggests that, although all models fit the data well, the BNN-SGHMC offers a more precise estimation of the target variable's distribution. Additionally, the findings reinforce that adding more tuning steps does not significantly enhance any of the models' predictive performance, indicating that the SGHMC method achieves its superior performance without the need for extensive tuning, which could streamline its application in practice.

4.1.3 Scalability

To evaluate the scalability of the three HMC variants, three experiments were conducted using progressively larger subsets of the dataset. In the first experiment, 25% of the data was utilised, which was reduced through stratified sampling to ensure balanced representation across classes. The second and third experiments employed 50% and 75% of the original dataset, respectively, also obtained through stratified sampling. For each of these experiments, an 80-20 data split was applied to create training and testing sets, and the Min-Max scaler was utilised to standardise the predictor variables in the test and training sets. The number of tuning steps was set to 250.

The resulting shapes of the datasets after the splits are as follows:

1. 25% of the Original Dataset: The training sets (X-train and y-train) comprised 1,000 rows each, while the testing sets (X-test and y-test) contained 250 rows each.

2. 50% of the Original Dataset: The training sets consisted of 2,000 rows each, with the testing sets having 500 rows each.

3. 75% of the Original Dataset: The training sets increased to 3,000 rows each, and the testing sets expanded to 750 rows each.

These experiments were designed to evaluate the performance of the three HMC variants under varying dataset sizes, providing insights into their scalability and adaptability. By systematically increasing the amount of data used, we aimed to determine whether the models could generalize effectively to larger datasets, thereby ensuring their robustness in real-world applications.

Table 4.2: Model Performance Evaluation

Data size	Model	MSE	RMSE	MAE	Training time (in seconds)
25%	BNN-HMC	0.1515	0.3893	0.3096	418.93
25%	BNN-NUTS	0.1516	0.3893	0.3106	403.71
25%	BNN-SGHMC	0.1506	0.3881	0.3087	110.69
50%	BNN-HMC	0.1420	0.3769	0.2944	918.06
50%	BNN-NUTS	0.1422	0.3771	0.2943	814.89
50%	BNN-SGHMC	0.1316	0.3627	0.2785	227.53
75%	BNN-HMC	0.1267	0.3560	0.2778	939.06
75%	BNN-NUTS	0.1270	0.3564	0.2775	1741.21
75%	BNN-SGHMC	0.0899	0.2998	0.2318	350.30

Table 4.2 displays the results of the scalability experiment.

The scalability experiments conducted with 25%, 50%, and 75% of the original reduced dataset yielded several key observations:

Experiment 1: 25% of the Original Reduced Dataset

- The BNN-SGHMC exhibited the shortest training times, which were approximately three times shorter than those of the BNN-HMC and BNN-NUTS models.
- All three models produced similar scores for MSE, RMSE, and MAE, indicating comparable predictive performance when using only 25% of the dataset.
- Despite the comparable predictive performance, the BNN-SGHMC demonstrated superior computational efficiency with faster convergence, while the BNN-NUTS was also more computationally efficient than the BNN-HMC.

Experiment 2: 50% of the Original Reduced Dataset

- The BNN-SGHMC achieved the lowest MSE, RMSE, and MAE scores, outperforming the BNN-HMC and BNN-NUTS models regarding predictive accuracy.
- Regarding training times, the BNN-SGHMC continued to maintain lower times compared to the other models, demonstrating its ability to scale efficiently as the dataset size increased.

- The BNN-HMC and BNN-NUTS models exhibited similar computational efficiency, with the BNN-NUTS showing a marginally shorter training time.

Experiment 3: 75% of the Original Reduced Dataset

- Once again, the BNN-SGHMC achieved the lowest MSE, RMSE, and MAE scores, confirming its superior performance across all metrics.
- The BNN-HMC and BNN-NUTS models continued to exhibit similar scores for the three evaluation metrics, showing that they behave almost identically in terms of predictive accuracy.
- The BNN-SGHMC's training time remained significantly shorter, around three times faster than both the BNN-HMC and BNN-NUTS models, demonstrating its scalability and computational efficiency even with larger datasets.

The results of the three scalability experiments suggest that, with smaller datasets (25% of the original), all three models perform similarly in terms of predictive accuracy, but the BNN-SGHMC consistently converges faster. As the dataset size increases, the BNN-SGHMC continues to outperform both the BNN-HMC and BNN-NUTS in terms of both training time and predictive accuracy, making it the most efficient and scalable model. The BNN-HMC and BNN-NUTS perform almost identically across all metrics, but the BNN-NUTS tends to have slightly better computational efficiency. Notably, as the dataset size increases, the MSE, RMSE, and MAE scores decrease for all three models, indicating improved accuracy with more data. Overall, these results demonstrate that the BNN-SGHMC is the most scalable model, balancing both predictive performance and computational efficiency across different dataset sizes.

4.2 Discussion

This section presents a summary of the findings (Subsection 4.2.1) and then discusses the interpretation of the research results (Subsection 4.2.2), highlighting their relevance to the research aims and objectives (Subsection 4.2.3). It also explores the implications of these findings for real-world applications (Subsection 4.2.4), addresses the limitations encountered during the study (Subsection 4.2.5), and outlines potential directions for future research (Subsection 4.2.6).

4.2.1 Summary of Findings

This research compared the performance, computational efficiency, and scalability of three HMC variants—standard HMC, NUTS, and SGHMC—in BNNs for prediction tasks. The

results demonstrated that SGHMC achieved better predictive accuracy than both HMC and NUTS while also being the most computationally efficient and scalable. NUTS and HMC showed comparable performance to each other but fell behind SGHMC in both accuracy and efficiency.

4.2.2 Interpretation of Results

The results indicate that SGHMC is a strong contender for tasks requiring both high predictive accuracy and computational efficiency. By leveraging stochastic gradients, SGHMC not only reduced training time but also provided the most accurate posterior predictions, outperforming both HMC and NUTS.

The comparable performance between NUTS and HMC aligns with prior research by Hoffman et al. (2014), as both methods require significant computational resources. NUTS' adaptive tuning helped reduce some manual effort, but its computational cost still placed it behind SGHMC, particularly for larger datasets. This highlights that while NUTS eliminates manual tuning challenges, it does not necessarily improve accuracy or efficiency compared to SGHMC.

4.2.3 Relevance to Research Aims and Objectives

The main aim of this research was to evaluate the computational efficiency, scalability and performance of different HMC variants in BNNs for prediction tasks. This research successfully addressed these aims, demonstrating that SGHMC is the most accurate and efficient method, while NUTS and HMC offered comparable but less optimal performance. These results provide clear recommendations on the most suitable HMC variant depending on the task's requirements, contributing to a deeper understanding of optimising BI in DL models.

4.2.4 Implications

The practical implications of these findings are important for practitioners and researchers working in BDL. SGHMC, with its superior accuracy and efficiency, should be the preferred method when handling large datasets or real-time prediction tasks where both speed and precision are critical. Its ability to outperform both NUTS and HMC in terms of predictive accuracy makes it an ideal choice for applications where the accuracy of posterior predictions is paramount, such as in autonomous systems or medical diagnostics. SGHMC's scalability suggests it can handle complex, large-scale problems without the trade-offs typically seen in traditional HMC methods. In contrast, NUTS and HMC, with their similar performance, may be more suited to smaller datasets or cases where computational resources are less constrained.

4.2.5 Limitations

Several limitations of this study should be considered. First, the experiments were conducted on a particular dataset, which can restrict the generalisability of the findings to other datasets with different characteristics, such as higher dimensionality or more noise. Additionally, the focus on three specific HMC variants might overlook other promising methods or hybrid approaches that could offer better performance in certain scenarios. Lastly, while the study compared computational efficiency, it did not account for hardware limitations (Subsection 3.8), which may also impact the real-world applicability of these methods. Furthermore, the randomness introduced by SGHMC's stochastic gradients could impact reproducibility across different hardware environments or implementations, which may require further investigation to ensure consistency in results.

4.2.6 Future Directions

Future research should focus on expanding the comparison to include other HMC variants, such as Riemannian Manifold HMC (RMHMC), Stochastic Variance Reduced Gradient HMC (SVRG-HMC), Generalized HMC (gHMC), Adaptive HMC (aHMC), and hybrid approaches that may offer better trade-offs between accuracy and efficiency. Additionally, studies could investigate the performance of these three variants on more complex datasets with varying structures, including those with higher dimensionality or different types of noise, to validate the findings in broader contexts. Another area worth exploring is how various hardware environments affect the efficiency and scalability of these methods, particularly for SGHMC, which relies heavily on stochastic gradients.

Chapter 5

Conclusion

This research investigates the integration of BI with DL models, specifically focusing on BDL to address overfitting and uncertainty in predictions. By utilising posterior probability distributions for uncertainty quantification, BDL enhances decision-making and reliability. Despite various posterior density estimation techniques, such as HMC and its variants, a comprehensive comparison of their performance, efficiency, and scalability within BDL remains lacking.

We conducted a comparative analysis of three BNN models using standard HMC, NUTS, and SGHMC for regression/prediction tasks, employing the California Housing Prices dataset. The results revealed that while all models achieved fairly low mean MSE, RMSE, and MAE scores, their training times increased significantly with larger datasets. Notably, the BNN with the SGHMC sampler outperformed the others in performance and computational efficiency as the dataset size grew. Although NUTS and standard HMC showed comparable results, the advantages of SGHMC became evident with larger datasets.

This study recommends the BNN model with SGHMC for practitioners, especially when handling larger datasets, due to its superior predictive accuracy and efficiency. By providing a thorough analysis of HMC variants in BNNs, this research not only fills a critical gap in existing literature but also sets the stage for future studies on other advanced HMC variants and hybrid approaches, further enhancing the reliability and efficiency of BDL models across diverse applications.

Bibliography

Abdullah, A. A., Hassan, M. M., and Mustafa, Y. T. (2022). A review on bayesian deep learning in healthcare: Applications and challenges. *IEEE Access*, 10:36538–36562. 12, 13

Abdullah, A. A., Hassan, M. M., and Mustafa, Y. T. (2024). Leveraging bayesian deep learning and ensemble methods for uncertainty quantification in image classification: A ranking-based approach. *Heliyon*, 10(2). 1

Abiodun, O. I., Jantan, A., Omolara, A. E., Dada, K. V., Mohamed, N. A., and Arshad, H. (2018). State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11). 7

Altartouri, H. (2021). Improving the classification of protein sequence functions by reducing the heterogeneity of datasets. 27

Arakawa, A., Hayashi, T., Taniguchi, M., Mikawa, S., and Nishio, M. (2021). Hamiltonian monte carlo method for estimating variance components. *Animal Science Journal*, 92(1):e13575. 2

Berrar, D. (2019). Bayes' theorem and naive bayes classifier. 14, 15

Bou-Rabee, N. and Sanz-Serna, J. M. (2017). Randomized hamiltonian monte carlo. 19

Büyükkaya, K., Karsavuran, M. O., and Aykanat, C. (2024). Stochastic gradient descent for matrix completion: Hybrid parallelization on shared-and distributed-memory systems. *Knowledge-Based Systems*, 283:111176. 10, 11

Bykov, K., Höhne, M. M.-C., Creosteanu, A., Müller, K.-R., Klauschen, F., Nakajima, S., and Kloft, M. (2021). Explaining bayesian neural networks. *arXiv preprint arXiv:2108.10346*. 13

Chang, D. T. (2021). Bayesian neural networks: Essentials. *arXiv preprint arXiv:2106.13594*. 15

Chen, T., Fox, E., and Guestrin, C. (2014). Stochastic gradient hamiltonian monte carlo. In *International conference on machine learning*, pages 1683–1691. PMLR. 2, 19, 20, 21

Choudhury, A. and Kosorok, M. R. (2020). Missing data imputation for classification problems. *arXiv preprint arXiv:2002.10709*. 24, 25

Clare, M. C. and Piggott, M. D. (2022). Bayesian neural networks for the probabilistic forecasting of wind direction and speed using ocean data. *Trends in Renewable Energies Offshore*, pages 533–540. 6

Cobb, A. D. and Jalaian, B. (2021). Scaling hamiltonian monte carlo inference for bayesian neural networks with symmetric splitting. In *Uncertainty in Artificial Intelligence*, pages 675–685. PMLR. 2

Dong, S., Wang, P., and Abbas, K. (2021). A survey on deep learning and its applications. *Computer Science Review*, 40:100379. 12

Elfil, M. and Negida, A. (2017). Sampling methods in clinical research; an educational review. *Emergency*, 5(1). 25, 26

Ellison, A. M. (2004). Bayesian inference in ecology. *Ecology letters*, 7(6):509–520. 15

Etz, A. and Vandekerckhove, J. (2018). Introduction to bayesian inference for psychology. *Psychonomic bulletin & review*, 25:5–34. 14, 15

Ferri Borredà, P. (2024). *Deeep Continual Multimodal Multitask Modells for Out-of-Hospital Emergency Medical Call Incidents Triage Support in the Presence of Dataset Shifts*. PhD thesis, Universitat Politècnica de València. 8

Fissha, Y., Ikeda, H., Toriya, H., Adachi, T., and Kawamura, Y. (2023). Application of bayesian neural network (bnn) for the prediction of blast-induced ground vibration. *Applied sciences*, 13(5):3128. 5, 23

Ghiassi, M. and Saidane, H. (2005). A dynamic architecture for artificial neural networks. *Neurocomputing*, 63:397–413. 7

Havasi, M., Hernández-Lobato, J. M., and Murillo-Fuentes, J. J. (2018). Inference in deep gaussian processes using stochastic gradient hamiltonian monte carlo. *Advances in neural information processing systems*, 31. 2, 22, 23

He, G., Zhao, Y., and Yan, C. (2024). Uncertainty quantification in multiaxial fatigue life prediction using bayesian neural networks. *Engineering Fracture Mechanics*, page 109961. 6

Hoffman, M. D., Gelman, A., et al. (2014). The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623. 21, 53

Jadhav, A., Pramod, D., and Ramanathan, K. (2019). Comparison of performance of data imputation methods for numeric dataset. *Applied Artificial Intelligence*, 33(10):913–933. 24, 25

James, G., Witten, D., Hastie, T., Tibshirani, R., et al. (2013). *An introduction to statistical learning*, volume 112. Springer. 24

Jha, R., Jha, N. N., and Lele, M. M. (2023). Stochastic gradient descent algorithm for the predictive modelling of grate combustion and boiler dynamics. *ISA transactions*, 136:571–589. 10, 11

Jones, G. L. and Qin, Q. (2022). Markov chain monte carlo in practice. *Annual Review of Statistics and Its Application*, 9:557–578. 17

Jospin, L. V., Laga, H., Boussaid, F., Buntine, W., and Bennamoun, M. (2022). Hands-on bayesian neural networks—a tutorial for deep learning users. *IEEE Computational Intelligence Magazine*, 17(2):29–48. 1, 13, 14, 15, 16, 17, 18, 19

Kazemi, R. and Mosleh, A. (2012). Improving default risk prediction using bayesian model uncertainty techniques. *Risk Analysis: An International Journal*, 32(11):1888–1900. 2

Kianiharchegani, E. (2023). *Data-Driven Exploration of Coarse-Grained Equations: Harnessing Machine Learning*. PhD thesis, The University of Western Ontario (Canada). 1

Krenker, A., Bešter, J., and Kos, A. (2011). Introduction to the artificial neural networks. *Artificial Neural Networks: Methodological Advances and Biomedical Applications. InTech*, pages 1–18. 7

Kumari, P. (2024). *Different Approaches of Quantitative Structure Retention Relationship of Small Molecules in Liquid Chromatography (QSRR)*. PhD thesis, Universite de Liege (Belgium). 27

Lambert, B. (2018). A student's guide to bayesian statistics. *A Student's Guide to Bayesian Statistics*, pages 1–520. 15

learn Developers, S. (2018). The california housing dataset. Accessed: 2025-01-27. 29

Lederer, J. (2021). Activation functions in artificial neural networks: A systematic overview. *arXiv preprint arXiv:2101.09957*. 8, 9

Li, Z., Zhang, T., Cheng, S., Zhu, J., and Li, J. (2019). Stochastic gradient hamiltonian monte carlo with variance reduction for bayesian inference. *Machine Learning*, 108:1701–1727. 6, 23

Little, R. J. and Rubin, D. B. (2002). Statistical analysis with missing data. john wiley & sons. *New York*. 24

Luengo, D., Martino, L., Bugallo, M., Elvira, V., and Särkkä, S. (2020). A survey of monte carlo methods for parameter estimation. *EURASIP Journal on Advances in Signal Processing*, 2020:1–62. 18

Ma, N., Chen, L., Hu, J., Perdikaris, P., and Braham, W. W. (2021). Adaptive behavior and different thermal experiences of real people: A bayesian neural network approach to thermal preference prediction and classification. *Building and Environment*, 198:107875. 1

Mbuvha, R. and Marwala, T. (2020). Bayesian inference of covid-19 spreading rates in south africa. *PloS one*, 15(8):e0237126. 14

Minar, M. R. and Naher, J. (2018). Recent advances in deep learning: An overview. *arXiv preprint arXiv:1807.08169*. 12

Mullachery, V., Khera, A., and Husain, A. (2018). Bayesian neural networks. *arXiv preprint arXiv:1801.07710*. 13

Mustapha, A., Mohamed, L., and Ali, K. (2020). An overview of gradient descent algorithm optimization in machine learning: Application in the ophthalmology field. In *Smart Applications and Data Analysis: Third International Conference, SADASC 2020, Marrakesh, Morocco, June 25–26, 2020, Proceedings 3*, pages 349–359. Springer. 10, 11

Ndirangu, G. I. (2023). Curating nba play-by-play data for predicting shooting success. Master's thesis, San Diego State University. 27

Neal, R. M. (2012). *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media. 43

Nishio, M. and Arakawa, A. (2019). Performance of hamiltonian monte carlo and no-u-turn sampler for estimating genetic parameters and breeding values. *Genetics Selection Evolution*, 51:1–12. 21

Nishio, M. and Arakawa, A. (2022). Performance of the no-u-turn sampler in multi-trait variance component estimation using genomic data. *Genetics Selection Evolution*, 54(1):51. 2

Nugent, C. (2017). California housing prices. https://www.kaggle.com/datasets/camnugent/california-housing-prices. Accessed: 2024-06-17. 28

Numpy (2024). numpy.sqrt. https://numpy.org/doc/stable/reference/generated/numpy.sqrt.html. Accessed: 2024-06-15. 44

Nwankpa, C., Ijomah, W., Gachagan, A., and Marshall, S. (2018). Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*. 9

Pandas (2024). pandas.isnull. https://pandas.pydata.org/docs/reference/api/pandas.isnull.html. Accessed: 2024-06-12. 34

Pandas-Authors (2024). pandas.series.duplicated. Accessed: 2025-02-03. 34

PyMC (2024). Learn pymc bayesian modeling. https://www.pymc.io/projects/docs/en/stable/learn.html. Accessed: 2024-06-14. 43

Ramchoun, H. and Ettaouil, M. (2020). New prior distribution for bayesian neural network and learning via hamiltonian monte carlo. *Evolving Systems*, 11(4):661–671. 6

Robert, C. P., Marin, J.-M., and Rousseau, J. (2010). Bayesian inference. *arXiv preprint arXiv:1002.2080*. 15

Salimans, T., Kingma, D., and Welling, M. (2015). Markov chain monte carlo and variational inference: Bridging the gap. In *International conference on machine learning*, pages 1218–1226. PMLR. 17

Sarker, I. H. (2021). Deep learning: a comprehensive overview on techniques, taxonomy, applications and research directions. *SN Computer Science*, 2(6):420. 12

Scikit, S. (2024). Minmaxscaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html. Accessed: 12 Jun 2024. 38

Scikit-learn (2024). Labelencoder. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html. Accessed: 2024-06-12. 38

Seoni, S., Jahmunah, V., Salvi, M., Barua, P. D., Molinari, F., and Acharya, U. R. (2023). Application of uncertainty quantification to artificial intelligence in healthcare: A review of last decade (2013–2023). *Computers in Biology and Medicine*, page 107441. 2

Setia, M. S. (2016). Methodology series module 5: Sampling strategies. *Indian journal of dermatology*, 61(5):505–509. 26

Sharma, S., Sharma, S., and Athaiya, A. (2017). Activation functions in neural networks. *Towards Data Sci*, 6(12):310–316. 8, 9

Song, B., Sunny, S., Li, S., Gurushanth, K., Mendonca, P., Mukhia, N., Patrick, S., Gurudath, S., Raghavan, S., Tsusennaro, I., et al. (2021). Bayesian deep learning for reliable oral cancer image classification. *Biomedical Optics Express*, 12(10):6422–6430. 12, 13

Suzuki, K. (2011). *Artificial neural networks: methodological advances and biomedical applications*. BoD–Books on Demand. 7

Tatachar, A. V. (2021). Comparative assessment of regression models based on model evaluation metrics. *International Research Journal of Engineering and Technology (IRJET)*, 8(09):2395–0056. 23

Tensorflow (2024). Api documentationg. https://www.tensorflow.org/api_docs. Accessed: 2024-06-14. 43

Tirink, C. (2022). Comparison of bayesian regularized neural network, random forest regression, support vector regression and multivariate adaptive regression splines algorithms to predict body weight from biometrical measurements in thalli sheep. *Kafkas Üniversitesi Veteriner Fakültesi Dergisi*, 28(3). 5

Van de Schoot, R. and Miocević, M. (2020). *Small sample size solutions: A guide for applied researchers and practitioners*. Taylor & Francis. 27

Venkatesh, R., Balasubramanian, C., and Kaliappan, M. (2019). Development of big data predictive analytics model for disease prediction using machine learning technique. *Journal of medical systems*, 43(8):272. 14

Wahle, J. (2021). No-u-turn sampling for phylogenetic trees. *bioRxiv*, pages 2021–03. 21

Wang, H. (2018). California housing data (1990). Accessed: 2025-01-27. 29

Wang, H. and Yeung, D.-Y. (2020). A survey on bayesian deep learning. *ACM computing surveys (csur)*, 53(5):1–37. 12, 13

Wang, T., Liu, Z., and Mrad, N. (2020). A probabilistic framework for remaining useful life prediction of bearings. *IEEE Transactions on Instrumentation and Measurement*, 70:1–12. 2

Westbury, C. F. (2010). Bayes' rule for clinicians: an introduction. *Frontiers in psychology*, 1:2149. 14, 15

Yu, H., Seno, A. H., Sharif Khodaei, Z., and Aliabadi, M. F. (2022). Structural health monitoring impact classification method based on bayesian neural network. *Polymers*, 14(19):3947. 13, 17

Zhang, S. (2012). Nearest neighbor selection for iteratively knn imputation. *Journal of Systems and Software*, 85(11):2541–2552. 24

Appendix A

Python Code

```
### import modules
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns


####################################   EDA

### import data
df = pd.read_csv('housing.csv')
df.head()

### checking for missing values
df.isnull().sum()

### Checking for duplicates
df.duplicated().sum()

### column info
df.info()

###summary statistics
df.describe()


### Visualisations

##1. Ocean proximity bar chart
ax = sns.countplot(x=df['ocean_proximity'],
                   order=df['ocean_proximity'].value_counts(ascending=False).index);

abs_values = df['ocean_proximity'].value_counts(ascending=False).values

ax.bar_label(container=ax.containers[0], labels=abs_values)

plt.title('Distribution of Ocean proximity')

plt.show()


##2. Pairplot to visualize relationships between variables
sns.pairplot(df)
plt.title('Pairplot')
plt.show()
```

```
##3.Correlation heatmap for numerical variables
corr = df.drop(columns=['ocean_proximity']).corr()
plt.figure(figsize=(10, 8))
sns.heatmap(corr, annot=True, cmap='coolwarm', fmt=".2f", linewidths=0.5)
plt.title('Correlation Heatmap')
plt.show()

## 4. # Distribution plot of numerical columns by target variable
plt.figure(figsize=(10, 6))
sns.histplot(df, x='median_house_value', kde=True)
plt.title('Distribution of the median house value feature')
plt.show()

## 5. # Plot KDE plot using seaborn
sns.kdeplot(df['median_house_value'], shade=True)
plt.title('Kernel Density Estimation (KDE) Plot of the median house value feature')
plt.xlabel('Data Values')
plt.ylabel('Density')
plt.show()

## 6. Boxplots for predictor features
numerical_features = df.columns[:-1]

# Set up the matplotlib figure
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(12, 8))

# Flatten the axes array for easy iteration
axes = axes.flatten()

# Plot boxplots for each predictor feature
for i, feature in enumerate(numerical_features):
    sns.boxplot(y=df[feature], ax=axes[i])
    axes[i].set_title(feature)

# Adjust layout
plt.tight_layout()
plt.show()
```

```
## 7. KDE plots of predictor features

new_features = df.columns[:-1]
# Set up the matplotlib figure
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(12, 8))

# Flatten the axes array for easy iteration
axes = axes.flatten()

# Plot histograms for each predictor feature
for i, feature in enumerate(new_features):
    sns.histplot(data=df, x=feature, ax=axes[i], kde = True)
    axes[i].set_title(feature)

# Adjust layout

plt.tight_layout()
plt.title('Distribution of numerical variables')
plt.show()


## Detecting outliers
from scipy import stats

# Calculate Z-scores for numeric columns
z_scores = df.select_dtypes(include=np.number).apply(stats.zscore, nan_policy='omit')

# Define a threshold (e.g., 3)
threshold = 3

# Create a boolean mask for outliers
outlier_mask = (z_scores.abs() > threshold)

# Get a list of outliers for each feature
outliers_by_feature = {}
for col in outlier_mask.columns:
    outliers_by_feature[col] = df.index[outlier_mask[col]].tolist()

#(Returns row numbers)
print(outliers_by_feature)
```

```
############################ Data Pre-peocessing


### Mean Imputation

df2 = df.copy()
# Calculate the mean of the 'feature' column, ignoring NaN values
mean_value = df2['total_bedrooms'].mean()

# Fill NaN values in the 'feature' column with the mean value
df2['total_bedrooms'].fillna(mean_value, inplace=True)

### checking for missing values after imputation
df2.isnull().sum()

### summary stats of new mean imputation df
df2.describe()

### New mean imputation df distribution plot

# Distribution plot of numerical columns by target variable
plt.figure(figsize=(10, 6))
sns.histplot(df2, x='total_bedrooms', kde=True)
plt.title('Distribution of the median total bedrooms feature after mean imputation')
plt.show()

### target distribution plot
# Distribution plot of numerical columns by target variable
plt.figure(figsize=(10, 6))
sns.histplot(df, x='total_bedrooms', kde=True)
plt.title('Distribution of the median total bedrooms feature in the original dataset')
plt.show()

### KNN imputation
df3 = df.drop(columns=['ocean_proximity']).copy()
from sklearn.impute import KNNImputer
# Create a KNNImputer object
imputer = KNNImputer()
df_imputed = pd.DataFrame(imputer.fit_transform(df3), columns=df3.columns)
df_imputed.isnull().sum()


### summary stats of new kNN imputation df
df_imputed.describe()


### New knn imputatio df distribution plot
# Distribution plot of numerical columns by target variable
plt.figure(figsize=(10, 6))
sns.histplot(df_imputed, x='total_bedrooms', kde=True)
plt.title('Distribution of the median total bedrooms feature after KNN imputation')
plt.show()
```

```
## creating new dataframe from knn imoutations
df_new = df_imputed.copy()
df_new['ocean_proximity'] = df['ocean_proximity']
df_new.head()

## Stratified Sampling

from sklearn.model_selection import train_test_split
# Define the feature to stratify by
stratify_feature = df_new['ocean_proximity']

# Perform stratified sampling
df_sampled, _ = train_test_split(df_new, test_size=(df_new.shape[0] - 5000) / df_new.shape[0], stratify=strati

# Verify the sampled data
print(df_sampled.shape)


#### Transforming ocean proimity feature to numerical feature

df_sampled['ocean_proximity'] =df_sampled['ocean_proximity'].map({'<1H OCEAN':0, 'INLAND':1, 'NEAR OCEAN':2,
                                                                  'NEAR BAY':3,'ISLAND':4})
df_sampled.head()
```

```
#### Feature Selection
df_sampled.drop('median_house_value', axis=1)


## 1. RFE
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.feature_selection import RFE
from sklearn.linear_model import LinearRegression

# Assuming df_sampled is your DataFrame
X = df_sampled.drop('median_house_value', axis=1)
y = df_sampled['median_house_value']

# Define model
model = LinearRegression()

# Define RFE
rfe = RFE(model, n_features_to_select=5)
fit = rfe.fit(X, y)

# Print features
print("Num Features: %s" % (fit.n_features_))
print("Selected Features: %s" % (fit.support_))
print("Feature Ranking: %s" % (fit.ranking_))

# Create a DataFrame for plotting
features = X.columns
ranking_df = pd.DataFrame({'Feature': features, 'Ranking': fit.ranking_})
ranking_df = ranking_df.sort_values(by='Ranking')

# Plot feature ranking
plt.figure(figsize=(12, 8))
plt.barh(ranking_df['Feature'], ranking_df['Ranking'], color='skyblue')
plt.xlabel('Ranking')
plt.title('Feature Ranking using RFE')
plt.gca().invert_yaxis()  # To have the most important feature at the top
plt.show()
```

```
## 2. LASSO Regression
from sklearn.linear_model import Lasso

# Define model
model1 = Lasso(alpha=0.01)
model1.fit(df_sampled.drop('median_house_value', axis=1), df_sampled['median_house_value'])

# Get feature importances
importance = model1.coef_

# Create a DataFrame for feature importances
features = df_sampled.drop('median_house_value', axis=1).columns
feature_importances_df = pd.DataFrame({
    'Feature': features,
    'Importance': importance
})

# Sort the DataFrame by importance
feature_importances_df = feature_importances_df.sort_values(by='Importance', ascending=False)

# Plot feature importances
plt.figure(figsize=(10, 6))
plt.barh(feature_importances_df['Feature'], feature_importances_df['Importance'], color='skyblue')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importances from Lasso Regression')
plt.gca().invert_yaxis()  # To display the highest importance at the top
plt.show()
```

```
## 3. Random Forest Regressor
from sklearn.ensemble import RandomForestRegressor

# Define model
model2 = RandomForestRegressor(n_estimators=100)
model2.fit(df_sampled.drop('median_house_value', axis=1), df_sampled['median_house_value'])

# Get feature importances
importance2 = model2.feature_importances_

# Create a DataFrame for feature importances
features = df_sampled.drop('median_house_value', axis=1).columns
feature_importances_df = pd.DataFrame({
    'Feature': features,
    'Importance': importance2
})

# Sort the DataFrame by importance
feature_importances_df = feature_importances_df.sort_values(by='Importance', ascending=False)

# Plot feature importances
plt.figure(figsize=(10, 6))
plt.barh(feature_importances_df['Feature'], feature_importances_df['Importance'], color='skyblue')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importances from RandomForestRegressor')
plt.gca().invert_yaxis()  # To display the highest importance at the top
plt.show()
```

```
##################################### Model Fitting

## data splitting (80-20 split)

#X =  df_sampled.drop('median_house_value', axis=1)
X = df_sampled[['longitude', 'latitude','housing_median_age', 'median_income', 'ocean_proximity']]
y =  df_sampled['median_house_value']

## log transforming target variable
y_log_transformed = np.log(y)
y = y_log_transformed

### Plot log-transformed target varibale
plt.figure(figsize=(10, 6))
plt.hist(y, bins=30, density=True, alpha=0.7, color='blue', edgecolor='black')
plt.title('Histogram of the log-transformed target variable')
plt.xlabel('Predicted Values')
plt.ylabel('Density')
plt.grid()
plt.show()

X_train, X_test,y_train, y_test = train_test_split(X,y,test_size=0.2, random_state=42)

# Print the shapes of the splits
print("Training feature set shape:", X_train.shape)
print("Test feature set shape:", X_test.shape)
print("Training target set shape:", y_train.shape)
print("Test target set shape:", y_test.shape)


##Standardising the data
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Code for trial 1 with 250 tuning steps. The same code applies to 500 tuning steps.

```
########## Trial 1 (250 steps)

### 1. HMC 1 with histogram
import pymc as pm
import arviz as az
from sklearn.metrics import mean_squared_error, mean_absolute_error
from math import sqrt
import time
import matplotlib.pyplot as plt

# Define the Bayesian Neural Network model
with pm.Model() as model:
    # Input layer
    X_data = pm.MutableData('X_data', X_train)
    y_data = pm.MutableData('y_data', y_train)

    # Hidden layer 1
    weights_1 = pm.Normal('weights_1', mu=0, sigma=1, shape=(X_train.shape[1], 10))
    bias_1 = pm.Normal('bias_1', mu=0, sigma=1, shape=(10,))
    layer_1 = pm.Deterministic('layer_1', pm.math.dot(X_data, weights_1) + bias_1)

    # Hidden layer 2
    weights_2 = pm.Normal('weights_2', mu=0, sigma=1, shape=(10, 10))
    bias_2 = pm.Normal('bias_2', mu=0, sigma=1, shape=(10,))
    layer_2 = pm.Deterministic('layer_2', pm.math.dot(layer_1, weights_2) + bias_2)

    # Output layer
    weights_out = pm.Normal('weights_out', mu=0, sigma=1, shape=(10, 1))
    bias_out = pm.Normal('bias_out', mu=0, sigma=1, shape=(1,))
    output = pm.Deterministic('output', pm.math.dot(layer_2, weights_out) + bias_out)

    # Likelihood
    sigma = pm.HalfNormal('sigma', sigma=1)
    y_pred = pm.Normal('y_pred', mu=output.flatten(), sigma=sigma, observed=y_data)

    # Measure training time
    start_time = time.time()
    trace = pm.sample(1000, tune=250, cores=2, return_inferencedata=True)
    training_time = time.time() - start_time
    print(f'Training time: {training_time:.2f} seconds')

# Posterior predictive sampling
with model:
    pm.set_data({'X_data': X_test, 'y_data': y_test})
    start_time = time.time()
    posterior_predictive = pm.sample_posterior_predictive(trace)
    prediction_time = time.time() - start_time
    print(f'Prediction time: {prediction_time:.2f} seconds')

# Predictions
y_pred_samples = posterior_predictive.posterior_predictive['y_pred']
y_pred_mean = y_pred_samples.mean(axis=(0,1))
```

```
# Predictions
y_pred_samples = posterior_predictive.posterior_predictive['y_pred']
y_pred_mean = y_pred_samples.mean(axis=(0,1))

# Evaluation metrics
mse = mean_squared_error(y_test, y_pred_mean)
rmse = sqrt(mse)
mae = mean_absolute_error(y_test, y_pred_mean)
print(f'MSE: {mse}')
print(f'RMSE: {rmse}')
print(f'MAE: {mae}')

# Histogram of predictions
plt.figure(figsize=(10, 6))
plt.hist(y_pred_samples.values.flatten(), bins=30, alpha=0.7, color='blue', edgecolor='black')
plt.title('Histogram of Posterior Predictive Samples - HMC 1')
plt.xlabel('Predicted Values')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show()
```

```
### 2. NUTS 1 with histogram

import pymc as pm
import arviz as az
from sklearn.metrics import mean_squared_error, mean_absolute_error
from math import sqrt
import time

# Define the Bayesian Neural Network model
with pm.Model() as model:
    # Input layer
    X_data = pm.MutableData('X_data', X_train)
    y_data = pm.MutableData('y_data', y_train)

    # Hidden layer 1
    weights_1 = pm.Normal('weights_1', mu=0, sigma=1, shape=(X_train.shape[1], 10))
    bias_1 = pm.Normal('bias_1', mu=0, sigma=1, shape=(10,))
    layer_1 = pm.Deterministic('layer_1', pm.math.dot(X_data, weights_1) + bias_1)

    # Hidden layer 2
    weights_2 = pm.Normal('weights_2', mu=0, sigma=1, shape=(10, 10))
    bias_2 = pm.Normal('bias_2', mu=0, sigma=1, shape=(10,))
    layer_2 = pm.Deterministic('layer_2', pm.math.dot(layer_1, weights_2) + bias_2)

    # Output layer
    weights_out = pm.Normal('weights_out', mu=0, sigma=1, shape=(10, 1))
    bias_out = pm.Normal('bias_out', mu=0, sigma=1, shape=(1,))
    output = pm.Deterministic('output', pm.math.dot(layer_2, weights_out) + bias_out)

    # Likelihood
    sigma = pm.HalfNormal('sigma', sigma=1)
    y_pred = pm.Normal('y_pred', mu=output.flatten(), sigma=sigma, observed=y_data)

    # Measure training time
    start_time = time.time()
    trace = pm.sample(1000, tune=250, cores=2, return_inferencedata=True, step=pm.NUTS())
    training_time = time.time() - start_time
    print(f'Training time: {training_time:.2f} seconds')

# Posterior predictive sampling
with model:
    pm.set_data({'X_data': X_test, 'y_data': y_test})
    start_time = time.time()
    posterior_predictive = pm.sample_posterior_predictive(trace)
    prediction_time = time.time() - start_time
    print(f'Prediction time: {prediction_time:.2f} seconds')

# Predictions
y_pred_samples = posterior_predictive.posterior_predictive['y_pred']
y_pred_mean = y_pred_samples.mean(axis=(0,1))
```

```
# Evaluation metrics
mse = mean_squared_error(y_test, y_pred_mean)
rmse = sqrt(mse)
mae = mean_absolute_error(y_test, y_pred_mean)

print(f'MSE: {mse}')
print(f'RMSE: {rmse}')
print(f'MAE: {mae}')

# Histogram of predictions
plt.figure(figsize=(10, 6))
plt.hist(y_pred_samples.values.flatten(), bins=30, alpha=0.7, color='blue', edgecolor='black')
plt.title('Histogram of Posterior Predictive Samples - NUTS 1')
plt.xlabel('Predicted Values')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show()
```

```
### 3. ### SGMC 1 with histogram
import numpy as np
import pandas as pd
import tensorflow as tf
import tensorflow_probability as tfp  # Import TensorFlow Probability
from sklearn.metrics import mean_squared_error, mean_absolute_error
import matplotlib.pyplot as plt  # Import Matplotlib for plotting
import time  # Import the time module

# Define the BNN model
class BayesianNN(tf.keras.Model):
    def __init__(self):
        super(BayesianNN, self).__init__()
        self.dense1 = tf.keras.layers.Dense(64, activation='relu')
        self.dense2 = tf.keras.layers.Dense(64, activation='relu')
        self.output_layer = tf.keras.layers.Dense(1)

    def call(self, inputs):
        x = self.dense1(inputs)
        x = self.dense2(x)
        return self.output_layer(x)

# SGHMC implementation
class SGHMC:
    def __init__(self, model, learning_rate=0.01, mass=1.0, friction=0.1):
        self.model = model
        self.learning_rate = learning_rate
        self.mass = mass
        self.friction = friction
        self.momentum = [tf.zeros_like(var) for var in model.trainable_variables]

    def update(self, x, y):
        with tf.GradientTape() as tape:
            y_pred = self.model(x)
            loss_fn = tf.keras.losses.MeanSquaredError()
            loss = loss_fn(y, y_pred)

        grads = tape.gradient(loss, self.model.trainable_variables)

        if len(self.momentum) != len(self.model.trainable_variables):
            self.momentum = [tf.zeros_like(var) for var in self.model.trainable_variables]

        for i, (var, grad) in enumerate(zip(self.model.trainable_variables, grads)):
            self.momentum[i] = (self.friction * self.momentum[i]
                                - self.learning_rate * (grad / self.mass))
            var.assign_add(self.momentum[i])

# Initialize model and optimizer
model = BayesianNN()
optimizer = SGHMC(model)
```

```
# Training loop
epochs = 250
batch_size = 32

# Measure training time
start_time = time.time()  # Start time for training
for epoch in range(epochs):
    for i in range(0, len(X_train), batch_size):
        x_batch = X_train[i:i + batch_size]
        y_batch = y_train.values[i:i + batch_size]
        optimizer.update(x_batch, y_batch)
end_time = time.time()  # End time for training
training_time = end_time - start_time  # Calculate training time

# Make predictions
start_time = time.time()  # Start time for testing
y_pred = model(X_test).numpy()
end_time = time.time()  # End time for testing
testing_time = end_time - start_time  # Calculate testing time

# Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test, y_pred)

# Output the results
print(f'Mean Squared Error (MSE): {mse:.4f}')
print(f'Root Mean Squared Error (RMSE): {rmse:.4f}')
print(f'Mean Absolute Error (MAE): {mae:.4f}')
print(f'Training Time: {training_time:.4f} seconds')  # Output training time
print(f'Testing Time: {testing_time:.4f} seconds')    # Output testing time

# Generate samples from the posterior predictive distribution
num_samples = 1000  # Number of samples to draw
predictions = []

for _ in range(num_samples):
    y_sample = model(X_test)  # Sample from the model
    predictions.append(y_sample.numpy())

# Convert predictions to a numpy array for easier processing
predictions = np.array(predictions)

# Histogram of predictions
plt.figure(figsize=(10, 6))
plt.hist(predictions.flatten(), bins=30, alpha=0.7, color='blue', edgecolor='black')
plt.title('Histogram of Posterior Predictive Samples - SGHMC 1')
plt.xlabel('Predicted Values')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show()
```

Scalability codes for the three trials. The same code applies for 50% and 75% data sizes.

```
#################### Scalability
### 25% of original data

sampled_25 = df_sampled.sample(frac=0.25, random_state=42)

## data splitting (80-20 split)

X_25 = sampled_25[['longitude', 'latitude','housing_median_age', 'median_income', 'ocean_proximity']]
y_25 = sampled_25['median_house_value']
y_25 = np.log(y_25)

X_train25, X_test25,y_train25, y_test25 = train_test_split(X_25,y_25,test_size=0.2, random_state=42)

## Scaling the data
scaler = MinMaxScaler()
X_train25 = scaler.fit_transform(X_train25)
X_test25 = scaler.transform(X_test25)

print(X_train25.shape)
print(X_test25.shape)
print(y_train25.shape)
print(y_test25.shape)
```

```
## HMC 25%
import pymc as pm
import arviz as az
from sklearn.metrics import mean_squared_error, mean_absolute_error
from math import sqrt
import time
import matplotlib.pyplot as plt

#Define the Bayesian Neural Network model
with pm.Model() as model:
    # Input layer
    X_data = pm.MutableData('X_data', X_train25)
    y_data = pm.MutableData('y_data', y_train25)

    # Hidden layer 1
    weights_1 = pm.Normal('weights_1', mu=0, sigma=1, shape=(X_train25.shape[1], 10))
    bias_1 = pm.Normal('bias_1', mu=0, sigma=1, shape=(10,))
    layer_1 = pm.Deterministic('layer_1', pm.math.dot(X_data, weights_1) + bias_1)

    # Hidden layer 2
    weights_2 = pm.Normal('weights_2', mu=0, sigma=1, shape=(10, 10))
    bias_2 = pm.Normal('bias_2', mu=0, sigma=1, shape=(10,))
    layer_2 = pm.Deterministic('layer_2', pm.math.dot(layer_1, weights_2) + bias_2)

    # Output layer
    weights_out = pm.Normal('weights_out', mu=0, sigma=1, shape=(10, 1))
    bias_out = pm.Normal('bias_out', mu=0, sigma=1, shape=(1,))
    output = pm.Deterministic('output', pm.math.dot(layer_2, weights_out) + bias_out)

    # Likelihood
    sigma = pm.HalfNormal('sigma', sigma=1)
    y_pred = pm.Normal('y_pred', mu=output.flatten(), sigma=sigma, observed=y_data)

    # Measure training time
    start_time = time.time()
    trace = pm.sample(250, tune=250, cores=2, return_inferencedata=True)
    training_time = time.time() - start_time
    print(f'Training time: {training_time:.2f} seconds')

#Posterior predictive sampling
with model:
    pm.set_data({'X_data': X_test25, 'y_data': y_test25})
    start_time = time.time()
    posterior_predictive = pm.sample_posterior_predictive(trace)
    prediction_time = time.time() - start_time
    print(f'Prediction time: {prediction_time:.2f} seconds')

# Predictions
y_pred_samples = posterior_predictive.posterior_predictive['y_pred']
y_pred_mean = y_pred_samples.mean(axis=(0,1))
```

```
    # Output layer
    weights_out = pm.Normal('weights_out', mu=0, sigma=1, shape=(10, 1))
    bias_out = pm.Normal('bias_out', mu=0, sigma=1, shape=(1,))
    output = pm.Deterministic('output', pm.math.dot(layer_2, weights_out) + bias_out)

    # Likelihood
    sigma = pm.HalfNormal('sigma', sigma=1)
    y_pred = pm.Normal('y_pred', mu=output.flatten(), sigma=sigma, observed=y_data)

    # Measure training time
    start_time = time.time()
    trace = pm.sample(250, tune=250, cores=2, return_inferencedata=True)
    training_time = time.time() - start_time
    print(f'Training time: {training_time:.2f} seconds')

#Posterior predictive sampling
with model:
    pm.set_data({'X_data': X_test25, 'y_data': y_test25})
    start_time = time.time()
    posterior_predictive = pm.sample_posterior_predictive(trace)
    prediction_time = time.time() - start_time
    print(f'Prediction time: {prediction_time:.2f} seconds')

# Predictions
y_pred_samples = posterior_predictive.posterior_predictive['y_pred']
y_pred_mean = y_pred_samples.mean(axis=(0,1))

# Evaluation metrics
mse = mean_squared_error(y_test25, y_pred_mean)
rmse = sqrt(mse)
mae = mean_absolute_error(y_test25, y_pred_mean)
print(f'MSE: {mse}')
print(f'RMSE: {rmse}')
print(f'MAE: {mae}')
```

```
### NUTS 25%

import pymc as pm
import arviz as az
from sklearn.metrics import mean_squared_error, mean_absolute_error
from math import sqrt
import time

# Define the Bayesian Neural Network model
with pm.Model() as model:
    # Input layer
    X_data = pm.MutableData('X_data', X_train25)
    y_data = pm.MutableData('y_data', y_train25)

    # Hidden layer 1
    weights_1 = pm.Normal('weights_1', mu=0, sigma=1, shape=(X_train25.shape[1], 10))
    bias_1 = pm.Normal('bias_1', mu=0, sigma=1, shape=(10,))
    layer_1 = pm.Deterministic('layer_1', pm.math.dot(X_data, weights_1) + bias_1)

    # Hidden layer 2
    weights_2 = pm.Normal('weights_2', mu=0, sigma=1, shape=(10, 10))
    bias_2 = pm.Normal('bias_2', mu=0, sigma=1, shape=(10,))
    layer_2 = pm.Deterministic('layer_2', pm.math.dot(layer_1, weights_2) + bias_2)

    # Output layer
    weights_out = pm.Normal('weights_out', mu=0, sigma=1, shape=(10, 1))
    bias_out = pm.Normal('bias_out', mu=0, sigma=1, shape=(1,))
    output = pm.Deterministic('output', pm.math.dot(layer_2, weights_out) + bias_out)

    # Likelihood
    sigma = pm.HalfNormal('sigma', sigma=1)
    y_pred = pm.Normal('y_pred', mu=output.flatten(), sigma=sigma, observed=y_data)

    # Measure training time
    start_time = time.time()
    trace = pm.sample(250, tune=250, cores=2, return_inferencedata=True, step=pm.NUTS())
    training_time = time.time() - start_time
    print(f'Training time: {training_time:.2f} seconds')
```

```
# Posterior predictive sampling
with model:
    pm.set_data({'X_data': X_test25, 'y_data': y_test25})
    start_time = time.time()
    posterior_predictive = pm.sample_posterior_predictive(trace)
    prediction_time = time.time() - start_time
    print(f'Prediction time: {prediction_time:.2f} seconds')

# Predictions
y_pred_samples = posterior_predictive.posterior_predictive['y_pred']
y_pred_mean = y_pred_samples.mean(axis=(0,1))

# Evaluation metrics
mse = mean_squared_error(y_test25, y_pred_mean)
rmse = sqrt(mse)
mae = mean_absolute_error(y_test25, y_pred_mean)

print(f'MSE: {mse}')
print(f'RMSE: {rmse}')
print(f'MAE: {mae}')
```

```
#### SGHMC 25%
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.metrics import mean_squared_error, mean_absolute_error
import time  # Import the time module

# Define the BNN model
class BayesianNN(tf.keras.Model):
    def __init__(self):
        super(BayesianNN, self).__init__()
        self.dense1 = tf.keras.layers.Dense(64, activation='relu')
        self.dense2 = tf.keras.layers.Dense(64, activation='relu')
        self.output_layer = tf.keras.layers.Dense(1)

    def call(self, inputs):
        x = self.dense1(inputs)
        x = self.dense2(x)
        return self.output_layer(x)

# SGHMC implementation
class SGHMC:
    def __init__(self, model, learning_rate=0.01, mass=1.0, friction=0.1):
        self.model = model
        self.learning_rate = learning_rate
        self.mass = mass
        self.friction = friction
        self.momentum = [tf.zeros_like(var) for var in model.trainable_variables]

    def update(self, x, y):
        with tf.GradientTape() as tape:
            y_pred = self.model(x)
            loss_fn = tf.keras.losses.MeanSquaredError()
            loss = loss_fn(y, y_pred)

        grads = tape.gradient(loss, self.model.trainable_variables)

        if len(self.momentum) != len(self.model.trainable_variables):
            self.momentum = [tf.zeros_like(var) for var in self.model.trainable_variables]

        for i, (var, grad) in enumerate(zip(self.model.trainable_variables, grads)):
            self.momentum[i] = (self.friction * self.momentum[i]
                                - self.learning_rate * (grad / self.mass))
            var.assign_add(self.momentum[i])

# Initialize model and optimizer
model = BayesianNN()
optimizer = SGHMC(model)

# Training loop
epochs = 250
batch_size = 32
```

```
# Measure training time
start_time = time.time()  # Start time for training
for epoch in range(epochs):
    for i in range(0, len(X_train25), batch_size):
        x_batch = X_train25[i:i + batch_size]
        y_batch = y_train25.values[i:i + batch_size]
        optimizer.update(x_batch, y_batch)
end_time = time.time()  # End time for training
training_time = end_time - start_time  # Calculate training time

# Make predictions
start_time = time.time()  # Start time for testing
y_pred = model(X_test25).numpy()
end_time = time.time()  # End time for testing
testing_time = end_time - start_time  # Calculate testing time

# Calculate evaluation metrics
mse = mean_squared_error(y_test25, y_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test25, y_pred)

# Output the results
print(f'Mean Squared Error (MSE): {mse:.4f}')
print(f'Root Mean Squared Error (RMSE): {rmse:.4f}')
print(f'Mean Absolute Error (MAE): {mae:.4f}')
print(f'Training Time: {training_time:.4f} seconds')  # Output training time
print(f'Testing Time: {testing_time:.4f} seconds')    # Output testing time
```

```
### 50% of original data
sampled_50 = df_sampled.sample(frac=0.5, random_state=42)

## data splitting (80-20 split)

X_50 = sampled_50[['longitude', 'latitude','housing_median_age', 'median_income', 'ocean_proximity']]
y_50 = sampled_50['median_house_value']
y_50 = np.log(y_50)

X_train50, X_test50,y_train50, y_test50 = train_test_split(X_50,y_50,test_size=0.2, random_state=42)

## Scaling the data
scaler = MinMaxScaler()
X_train50 = scaler.fit_transform(X_train50)
X_test50 = scaler.transform(X_test50)

print(X_train50.shape)
print(X_test50.shape)
print(y_train50.shape)
print(y_test50.shape)
```

```
### 75% of original data
sampled_75 = df_sampled.sample(frac=0.75, random_state=42)

## data splitting (80-20 split)

X_75 = sampled_75[['longitude', 'latitude','housing_median_age', 'median_income', 'ocean_proximity']]
y_75= sampled_75['median_house_value']
y_75 = np.log(y_75)

X_train75, X_test75,y_train75, y_test75 = train_test_split(X_75,y_75,test_size=0.2, random_state=42)

## Scaling the data
scaler = MinMaxScaler()
X_train75 = scaler.fit_transform(X_train75)
X_test75 = scaler.transform(X_test75)

print(X_train75.shape)
print(X_test75.shape)
print(y_train75.shape)
print(y_test75.shape)
```