Comparative Study of BNN Implementations

Pranav Narendra Gopalkrishna
231052045
Paulo Rauber
MSc. Artificial Intelligence

***Abstract*—Deep learning using neural networks (NNs) lacks transparency about the model's confidence in its predictions. Bayesian neural networks (BNNs) are a means to achieve a level of transparency and interpretability in deep learning by using Bayesian inference (BI) to quantify the model's predictive uncertainty (i.e. uncertainty about its predictions). The link between BI and BNNs is not easy, and implementing BNNs in practice needs significant mathematical and algorithmic depth. Furthermore, there exist both exact and approximate approaches to BNNs, the former offering accuracy, the latter offering efficiency. In this study, the two approaches are explored in theoretical and practical detail, using (for reasons discussed later) Hamiltonian Monte Carlo (HMC) as the exact BI method, and Variational Inference (VI) as the approximate BI method. The implementations of each are applied to four synthetic regression problems and are evaluated by observing (1) the average accuracy of their predictions and (2) the quality of the quantification of their predictive uncertainty.**

***Index Terms*—Bayesian Inference (BI), Markov Chain Monte Carlo (MCMC), Variational Inference (VI), Hamiltonian Monte Carlo (HMC), Neural Network (NN), Bayesian Neural Network (BNN)**

I. Introduction

A BNN is a kind of NN that can quantify its confidence about its predictions, or more precisely, its predictive uncertainty. Furthermore, using the framework of BI, it can explain – mathematically – how this uncertainty is quantified. To elaborate, a BNN is a stochastic neural network (SNN) trained using BI (Jospin et al., 2020). An SNN is an NN with stochastic elements (e.g. stochastic weights or stochastic activations) that, through its stochasticity, simulates a number of models according to some probability distribution of models; thus, SNNs are a special case of ensemble learning. Thus, a BNN is an SNN whose models are distributed by a posterior for the parameterisations of a more generalised model.

There are practical reasons to explore BNNs. Firstly, BNNs help distinguish between epistemic uncertainty (uncertainty due to lack of knowledge) and aleatoric uncertainty (uncertainty due to the model's inherent stochasticity, i.e. uncertainty due to factors that cannot be accounted for practically); thus, BNNs help avoid overfitting, since when making predictions, the model gives a high epistemic uncertainty for points outside the expected distribution of data points (which is based on the training data points) instead of a prediction that is likely to be incorrect (Jospin et al., 2020). Secondly, every deep learning model has assumptions about the distribution of its parameters; these assumptions are implicit in the initialisation of its weights and measures (if any) to avoid overfitting, such as regularisation and drop-out. A BNN makes assumptions about its parameters explicit through its prior, thus making the model more transparent and interpretable.

Thus, BNNs are a means to achieve transparency, interpretability and generalisability in deep learning. In light of the practical value of BNNs as well as their theoretical and practical complexity, the goals of this study are as follows: (1) bridging the gap between BNN's theory and its implementation, (2) implementing two fundamentally distinct approaches to quantifying the uncertainty of an NN's predictions, (3) evaluating the results of the aforementioned approaches, and (4) making the theory and practice of BNNs accessible and reproducible to someone new to BI and BNNs.

As indicated, the focus shall be on two fundamentally distinct types of BNNs: (1) BNN via an exact BI method, and (2) a BNN via an approximate BI method. Due to being theoretically well-founded and practically well-established, HMC (a class of MCMC methods) and VI are used for the exact and approximate BI method respectively; the meaning of "exact" and "approximate" in the context of BI and the theoretical and practical details of the methods used shall be discussed in later sections.

II. Related Work

Jospin et al. (2020) give an in-depth introduction to BI for deep learning, covering a wide range of topics from BI algorithms to performance metric for BNNs. BI itself (from the mathematics to the algorithms) is covered more deeply by Martin et al. (2021) in their textbook "Bayesian Modelling and Computation in Python"; notably, the textbook also introduces MCMC methods (Metropolis-Hastings and HMC) and VI. On the other hand, Chandra and Simmons (2023) give a practical tutorial for implementing MCMC using Python, starting from MCMC for a single-valued parameter and progressing to linear regression via MCMC and finally to BNNs via MCMC.

Using four evaluation metrics (validity of the confidence intervals, distance to the HMC reference, distance to the target

posterior and similarities between the algorithms), Brian and Da Veiga (2022) evaluate the performance of a wide range of approximation methods for BNNs on synthetic regression tasks. In practice, where parameter spaces tend to be high-dimensional (due to complex models, e.g. deep learning models) and datasets tend to be large (due to complex problems, e.g. image analysis), approximate BI methods are key areas of study. Yao et al. (2019) focus on evaluating the quality of uncertainty quantification using empirical comparison of 8 state-of-the-art approximate BI methods and 2 non-Bayesian frameworks. On the other hand, Foong et. al (2019) focus not only on evaluating approximate BI methods but also exploring pathologies arising due to approximation.

III. Relevant Concepts

A. Bayesian Inference (BI)

BI is the process of inferring a generative model[1] that explains the observed data. BI consists of (1) proposing a generative model parameterised by θ, (2) making judgements about the model's parameterisations prior to considering the observed data D, (3) considering D by measuring the likelihood of the model generating D for the given parameterisation, and (4) measuring the plausibility — or more precisely, the posterior probability — of the model's parameterisations, given the prior judgements and D. Hence, note that the generative model and prior judgements about it are set before the inference (e.g. using assumptions and/or knowledge apart from D) and the likelihood is based on the generative model. Hence, BI seeks the posterior probability of the model's parameterisations based on D. Mathematically, BI is based on Bayes' theorem:

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$$

θ	A parameterisation of the model
D	The observed data
P	The probability measure
$P(\theta\|D)$	The posterior
$P(D\|\theta)$	The likelihood
$P(\theta)$	The prior
$P(D)$	The evidence

The evidence term $P(D)$ is used to normalise the numerator $P(D|\theta)P(\theta)$; without this, we only have $P(D|\theta)P(\theta) = P(\theta, D)$, i.e. the joint probability of θ and D rather than the conditional probability of θ given D. The evidence term can be interpreted as the total probability of the observed data being generated by the model for any hypothetical parameterisation. Mathematically, if Θ is the set of all hypothetical parameterisations, then $P(D) = \int_{\theta\in\Theta} P(D|\theta)P(\theta)d\theta$. In practice, however, the $P(D)$ is often intractable and thus BI methods (e.g. MCMC and VI) have been developed to estimate the posterior through just the unnormalised posterior, i.e. $P(D|\theta)P(\theta)$.

[1]A model that generates data based on a well-defined random process.

B. Exact vs. Approximate BI

A exact BI method (e.g. MCMC) is one that is theoretically guaranteed to converge to posterior. An approximate BI method (e.g. VI) is one that uses some approximation of the posterior that is not theoretically guaranteed to converge to the posterior; the goal of such a method is to minimise the distance between the approximation and the posterior.

C. Markov Chain Monte Carlo (MCMC)

It is key to first understand MCMC, since HMC is a class of MCMC methods. MCMC is a class of methods for sampling from a target distribution. Hence, it is an exact BI method when the target distribution is the posterior $P(\theta|D)$, which is the probability distribution of the model parameter θ as conditioned by the observed data D.

MCMC works when we know the prior $P(\theta)$ and likelihood $P(D|\theta)$ and thereby know the unnormalised posterior $P(D|\theta)P(\theta) \propto P(\theta|D)$. Using the unnormalised posterior, we can calculate the probability α of a proposed sample θ' of θ being drawn from the posterior. In MCMC, α is used as the acceptance probability of θ', i.e. the probability of accepting θ' to be a sample from the posterior. To make proposals for new samples, MCMC sets only one condition: the probability of proposing a new sample must depend on the current sample. The motivation for this condition is that when a sample has a high probability of being from the posterior, there must be a method, using this sample as the starting point, to find other samples with a high probability of being from the posterior.

For example, Metropolis-Hastings (an MCMC method) proposes new samples using a random walk starting from the current sample, under the assumption that when the current sample has a high probability of being from the posterior, so do the samples in its vicinity[2]. In effect, MCMC results in a Markov chain of Monte Carlo samples (hence its name), with each state of the Markov chain being a sample from the posterior. More precisely, the transition probabilities of MCMC's Markov chain, which are based on the acceptance probability α, are such that (1) the Markov chain is theoretically guaranteed to converge to a steady-state (i.e. a state where long-range transition probabilities are stable), and (2) the steady-state probabilities (i.e. long-range transition probabilities) are theoretically guaranteed to correspond to the probabilities of sampling from the target distribution. To summarise, MCMC goes through the following steps:

1) Set the current sample with an initial value θ_0
2) Using a proposal method, propose a new sample θ'
3) Calculate the acceptance probability α of θ'

[2]Seeking higher probability samples does not preclude the inclusion of lower probability samples; indeed, for an accurate estimation of the target distribution, we need all kinds of samples. But the proportion of each kind of sample drawn corresponds to its probability with respect to the target distribution.

4) If θ' is accepted, set θ' as the current sample
5) Repeat from step 2 for a fixed number of iterations

D. Hamiltonian Monte Carlo (HMC)

HMC is a class of MCMC methods that uses gradients of the negative log-probability of the posterior to generate new proposed states, i.e. new samples proposed to be from the target distribution (the reason for using the negative log-probability of the posterior is explained in appendix A). The gradients of the negative log-probability of the posterior evaluated at a given state (i.e. a given sample) give information about the posterior density function's geometry. Owing to the geometric interpretation of HMC, states (i.e. samples) are also called positions.

As with MCMC, only the unnormalised posterior $P(D|\theta)P(\theta)$ is needed for HMC, because the gradients of the negative log-probability are equal for the posterior $P(\theta|D)$ and the unnormalised posterior $P(D|\theta)P(\theta)$, as show below:

$$
\begin{aligned}
&-\frac{\delta}{\delta\theta}\log P(\theta|D) = -\frac{\delta}{\delta\theta}\log\frac{P(D|\theta)P(\theta)}{P(D)} \text{ (by Bayes' rule)}\\
&= -\frac{\delta}{\delta\theta}(\log P(D|\theta)P(\theta) - P(D))\\
&= -\frac{\delta}{\delta\theta}\log P(D|\theta)P(\theta) + \frac{\delta}{\delta\theta}P(D)\\
&= -\frac{\delta}{\delta\theta}\log P(D|\theta)P(\theta) \text{ (since } P(D) \text{ is a constant)}
\end{aligned}
$$

HMC tries to avoid the random walk behaviour typical of Metropolis-Hastings by using gradients to propose new positions (i.e. new samples) that is both far from the current position (i.e. current sample) and with high acceptance probability (Martin et al., 2021). This allows HMC to scale well to higher dimensions and, in principle, to more complex geometries compared to other MCMC methods (Martin et al., 2021). Intuitively, we can think of HMC as a Metropolis-Hasting algorithm with a superior sample proposal distribution (Martin et al., 2021). The HMC algorithm as well as a deeper explanation of HMC using a physical analogy is given in appendix B.

To put HMC in practice, there must be an efficient way to accurately solve Hamilton's equations in the context of BI. Each of Hamilton's equations is a differential equation (DE), and solving a DE means integrating it, i.e. finding the functions whose differentials, which are present in the DE, satisfy the DE. To this end, symplectic integrators (discussed in appendix B) are used, and a well-established symplectic integrator is the leapfrog integrator (which is also used by the HMC implementation for this dissertation). The details of leapfrog integration can be found in appendix C.

E. Variational Inference (VI)

VI is a method of analytically approximating the target distribution. More precisely, VI approximates the target distribution p using a distribution q_ϕ — called the variational distribution — parameterised by ϕ. To approximate p using q_ϕ, ϕ is optimised to minimise the distance between p and q_ϕ; a well-established measure of distance is the Kullback-Leibler divergence (KL-divergence). Hence, when applied to BI, VI is an approximate BI method, the target distribution being the posterior.

In theory, MCMC converges to the posterior, but in practice, this convergence may be inefficient or even infeasible due to the posterior's complexity arising from the model's complexity. After all, converging to the posterior through samples is a statistical inference problem that has a need for sufficient sample quantity and quality; the higher the posterior's complexity, the higher the need. VI, on the other hand, simplifies the statistical inference problem to an optimisation problem (Ganguly and Earp, , 2021), gaining efficiency while losing the theoretical guarantee of convergence, since in general, there is no theoretical guarantee that a known distribution can converge to an unknown distribution.

A well-established method to perform VI is with the use of the Evidence Lower Bound (ELBO), which is derived from the KL-divergence between the variational distribution and the posterior (see appendix D). However, the focus shall be on the method used in *torchbnn*, a PyTorch-based implementation of BNNs (Kim, 2020; Lee et al., 2021), since this method can be more easily applied to a neural network. This implementation is an adversarial BNN based on the paper by Lee et al. (2021), which, while not a VI method explicitly, shall now be shown to be a VI method implicitly, i.e. in essence.

Given an ANN, the variational distribution is taken as a distribution whose parameters are functions of the existing weights and/or biases of the ANN. For example, the variational distribution can be a normal distribution parameterised by (μ_q, σ_q), where μ_q and σ_q are the mean and standard deviation of the ANN's weights. Given a well-defined prior, the training is done using (1) a point estimation loss chosen as per the given problem (e.g. mean squared error, for regression problems), and (2) the KL-divergence between the variational distribution and the prior. Hence, the sum of the point estimation loss and KL-divergence serves as a measure of distance between the variational distribution and the posterior.

IV. Methodology

The methodological focus is on four key areas: (1) well-motivated synthetic regression problems, (2) a BNN via an exact BI method, (3) a BNN via an approximate BI method, and (4) experiments for evaluating the performance of the aforementioned BNNs (which are presented in section **V. Results**). The implementations are in Python, using both TensorFlow and PyTorch. Note that for both kinds of BNNs, the training parameters (namely the learning rate, number of epochs and batch size) were chosen by trail-and-error based

on which values produced sufficiently high levels of average accuracy (i.e. after having averaged the samples/predictions).

A. Synthetic Regression Problems

The focus is not on the predictive accuracy of the models but rather the models' ability to quantify the uncertainty about the process (real-life or synthetic) that is generating the observed data. Hence, the problems need the be such that:

- Basic NN models can accurately train for them
- The data is noisy enough to cause uncertainty
- The data has complexities leading to areas of uncertainty

The four synthetic regression problems used by Brian and Da Veiga (2022) are such that each problem meets some or all of the above requirements; note that the problems that meet only specific requirements help focus on specific aspects of the BNNs' performance. The problems have been changed without changing their essence and are as follows (note that $\mathcal{N}(\mu, \sigma)$ denotes a normal distribution with mean μ and standard deviation σ, whereas $\mathcal{U}([a, b])$ denotes a uniform distribution over the interval $[a, b]$):

Synthetic Problem A

Outputs	$y_i = \cos 2x_i + \sin x_i + \epsilon_i$
Error term	$\epsilon_i \sim \mathcal{N}(0, 0.25)$
Train inputs	$x_i \sim \mathcal{U}([-3, 3])$
Test inputs	$x_i \sim \mathcal{U}([-3, 3])$

Synthetic Problem B

Outputs	$y_i = 0.1x_i^3 - x + \epsilon_i$
Error term	$\epsilon_i \sim \mathcal{N}(0, 0.25)$
Train inputs	$x_i \sim \mathcal{U}([-4, 1] \bigcup [3, 4])$
Test inputs	$x_i \sim \mathcal{U}([-4, 4])$

Synthetic Problem C

Outputs	$y_i = -(1 + x_i)\sin(1.2x_i) + \epsilon_i$
Error term	$\epsilon_i \sim \mathcal{N}(0, 0.5)$
98% train inputs	$x_i \sim \mathcal{U}([-6, 2] \bigcup [2, 6])$
2% train inputs	$x_i \sim \mathcal{U}([-2, 2])$
Test inputs	$x_i \sim \mathcal{U}([-6, 6])$

Synthetic Problem D

Outputs	$y_i = f(x_i, w) + \epsilon_i$
Weights	$w \sim \mathcal{N}(0, I_d)$
Error term	$\epsilon_i \sim \mathcal{N}(0, 500)$
Training inputs	$x_i \sim \mathcal{U}([-10, 6] \bigcup [6, 10] \bigcup [14, 18])$
Test inputs	$x_i \sim \mathcal{U}([-12, 22])$

The function f in synthetic problem D denotes a feed-forward neural network with three hidden layers with 100 neurons each (thus, 20501 parameters in total); the network's weights w are sampled once from a standard multivariate Gaussian distribution $\mathcal{N}(0, I_d)$. For reproducibility, the PyTorch seed for pseudo-random number generation was set as 3 using the instruction: *torch.manual_seed(3)* (wherein *torch* is the name by which PyTorch was imported).

Problems A and C introduce complexity in their functional forms, problems B and D introduce gaps in the data, and problem C introduces data sparsity within the interval $[-2, 2]$. In each problem, the noise terms ϵ_i are normally distributed so that the outputs are normally distributed, hence making normal priors and likelihoods viable in the later BIs; such an approach is motivated by the fact that normal distributions are easy to work with mathematically. The standard deviations of the noise terms were chosen by trial and error. Hence, the problems introduce uncertainty through noise, shape, gaps and sparsity.

B. HMC Implementation

1) Functional Model: An artificial neural network (ANN), implemented using TensorFlow's Keras API, is used as the generative model, i.e. functional model for BI. The ANN's architecture is meant to simple enough to allow for as efficient sampling as possible, yet complex enough to allow for the accurate modelling of each synthetic regression problem. The ANN's architecture is as follows (note that the parameters include bias terms, 100 per layer):

Layer	Shape	Parameters
Input	(1, 100)	200
Hidden	(100, 100)	10100
Output	(100, 1)	101

The ANN is trained for 500 epochs and a batch size of 32, using a learning rate of 0.01. The ANN's weights are initialised by random values and are trained using stochastic gradient descent with mean squared error loss. To improve the ANN's performance, (1) per training loop, only the weights of the epoch with the lowest loss are saved, and (2) 5 training loops are run for the same regression problem, with the weights reinitialised at the start of every loop. In the end, the best performing weights (i.e. the weights with the lowest loss) are chosen.

2) HMC Sampler: The HMC sampler used is the *HamiltonianMonteCarlo* kernel from the TensorFlow Probability MCMC Package. The number of samples to be drawn is set as 20000, since this was sufficient to lead to a more than 50% acceptance rate. The number of burn-in steps (i.e. the steps for which the Markov chain is considered to be converging to its steady-state) is set as 2500.

To discretise the Hamiltonian equations used to explore the posterior, this HMC sampler uses the leapfrog integrator, which has two hyperparameters: (1) step-size Δt (i.e. the size of the discrete time steps[3] for which the Hamiltonian equations are solved, thereby simulating the sampler's trajectory when exploring the posterior) and (2) the number of leapfrog steps L; hence, the length of the sampler's simulated trajectory

[3]"Time" here is fictitious, referring to the physical analogy.

from the current sample to the proposed sample is $T = L \cdot \Delta t$.

Based on Neal (2012, p. 135), if the step-size is too large, the discretisation errors of the simulated trajectories get so large that the proposed samples have a low acceptance rate, i.e. there are to too many sub-optimal proposals. If the step-size is too small, the simulated trajectory's length T gets so small that the exploration of the posterior is too inefficient, possibly more inefficient than even a random walk. To avoid such issues, the simple step-size adaptation policy is used, wherein the step-size is not fixed before sampling but rather multiplicatively increased or decreased during sampling (TensorFlow, 2023), based on the logarithm of the acceptance probabilities; this policy is based on equation 19 of Andrieu and Thoms (2008). To achieve this policy in code, the *HamiltonianMonteCarlo* kernel is wrapped in the *SimpleStepSizeAdaptation* kernel from the TensorFlow Probability MCMC Package. The number of adaptation steps is given as 0.8 times the number of burn-in steps.

The simulated trajectory's length $T = L \cdot \Delta t$ is key to the HMC exploring the state space systematically rather than by a random walk (Neal, 2011, p. 137). Having chosen the step-size Δt, the number of leapfrog steps L must now be chosen. Choosing the number of leapfrog steps L for a complex problem has to be done through trial-and-error; in the case of exploring the posterior of the weights of a neural network, which is a high-dimensional, non-convex and thus multi-modal distribution, a high number of leapfrog steps such as $L = 100$ may be suitable (Neal, 2011, p. 137) and is hence chosen.

Note that the HMC sampler starts with its initial position as the trained weights of the ANN, because it is reasonable to assume that if the trained weights of the ANN lead to accurate predictions, then this parameterisation of the ANN may be considered as a high probability sample of the posterior, making it a practical initial position for the HMC sampler. With a high probability initial position and an efficient sample proposal method (as given by HMC), HMC is likely to lead to more representative samples of the posterior with very few (if any) burn-in samples. However, the burn-in is set as 2500 (12.5% of the total number of samples) to allow for some adaptation, if necessary.

---

3) BI Components: To perform HMC sampling, the likelihood $P(D|\theta)$ and prior $P(\theta)$ must be defined so as to define the unnormalised posterior $P(D|\theta)P(\theta) \propto P(\theta|D)$ (as seen before, the HMC sampler only needs the gradients of the unnormalised posterior with respect to the position θ).

For basic architectures as used in Bayesian regression, it is standard to use a multivariate normal prior with a zero mean and a diagonal covariance σI_d on the coefficients of the ANN. In other words, the prior assumption is that $\theta \sim \mathcal{N}(0, \sigma I_d)$. Hence, the prior density of a parameterisation θ is given by:

$$P(\theta) = \mathcal{N}(0, \sigma I_d)(\theta)$$

Here, I_d is an identity matrix of d dimensions, where d is the dimensionality of θ. There is no theoretical argument that makes such a prior superior to any other (Jospin et al., 2020), but in practice, if lacking more well-motivated priors, a zero mean normal prior is preferred due to (1) its mathematical properties and (2) the ease of taking its logarithm (logarithms are used in machine learning to prevent arithmetic overflow). Furthermore, such a prior serves to make the ANN to keep its weights within a range unless there is overwhelming evidence to the contrary. In particular, a multivariate normal prior with mean 0 and diagonal covariance σI_d is equivalent to a weighted l_2 regularisation when training a point estimate ANN (Jospin et al., 2020).

A regression problem models the output (i.e. the target) y_i as $y_i = f_\theta(x_i) + \epsilon_i$, where x_i is the input, f_θ is the functional model parameterised by θ, and $epsilon_i$ is the error term. Here, the error term is taken to be distributed by $\mathcal{N}(0, \tau)$. The likelihood of the observed data D given the parameterisation θ presupposes (1) the observed data $D = (x, y)$, where $x = (x_1, x_2...x_n)$ and $y = (y_1, y_2...y_n)$ (here, we consider the inputs as one-dimensional) (2) the functional model f_θ as parameterised by θ, and (3) the distribution of error terms. Note that when taking the likelihood, D and θ are taken as constant. Hence, the target y is also distributed by a normal distribution, its standard deviation being the same as the standard deviation for the error term. More precisely, given $\hat{y_i} = f_\theta(x_i)$, we have that $y_i \sim \mathcal{N}(f_\theta(x_i), \tau)$ (note that τ is assumed to be known, and is assigned the exact value of the standard deviation of the given synthetic regression problem's error term ϵ_i). Hence, the likelihood density for $y = (y_1, y_2...y_n)$ is:

$$\mathcal{N}(f_\theta(x), \tau)(y) = \prod_{i=1}^{n} \mathcal{N}(f_\theta(x_i), \tau)(y_i)$$

The above product would lead to arithmetic overflow if there are a substantial number of data points, i.e. if n is to large, and thus, in practice, the log-likelihood density is used:

$$\log \prod_{i=1}^{n} \mathcal{N}(f_\theta(x_i), \tau)(y_i) = \sum_{i=1}^{n} \log \mathcal{N}(f_\theta(x_i), \tau)(y_i)$$

The *HamiltonianMonteCarlo* kernel from the TensorFlow Probability MCMC Package takes in a callable for the target log-probability, which becomes the log-probability of the unnormalised posterior for BI. Hence, this callable is to return the following:

$$\log P(D|\theta)P(\theta) = \log P(D|\theta) + \log P(\theta)$$

$$= \log \mathcal{N}(f_\theta(x), \tau)(y) + \log \mathcal{N}(0, \sigma I_d)(\theta)$$

$$= \sum_{i=1}^{n} \log \mathcal{N}(f_\theta(x_i), \tau)(y_i) + \log \mathcal{N}(0, \sigma I_d)(\theta)$$

4) Distributed Parameter: For reference, the ANN weights:

Index	Layer	Is Bias?	Shape
0	Input	False	(1, 100)
1	Input	True	(100)
2	Hidden	False	(100, 100)
3	Hidden	True	(100)
4	Output	False	(100, 1)
5	Output	True	(100)

Instead of distributing the weights of the ANN as a whole, only the hidden weights (index 2) were chosen as the parameter to be distributed because it has 10000 out of the ANN's 10401 parameters, i.e. $\sim 96.145\%$ of the ANN's parameters. Moreover, distributing only one array of weights makes implementing and applying the prior and likelihood far easier, since prior and likelihood functions need to only work with only one array rather than a list of arrays with non-matching shapes.

C. VI Implementation

The implementation of VI is done using PyTorch and the package *torchbnn* (Lee et al., 2021), which is an implementation of BNNs using PyTorch. The BNN architecture is given below:

Layer	Type	Shape	Parameters
Input	Linear	(1, 100)	200
Hidden	Bayes' Linear	(100, 100)	10100
Output	Linear	(100, 1)	101

For parity in the architectures of the HMC implementation and the VI implementation, only hidden layer's weights are to be distributed. More precisely, only the hidden layer is a Bayes' linear layer. In *torchbnn*, a Bayes' linear layer is one with a normal prior $\mathcal{N}(\mu, \sigma)$ and with a normal variational distribution q_ϕ parameterised by $\phi = (\mu_q, \sigma_q)$, wherein μ_q and σ_q are the mean and standard deviation of the layer's weights.

The model is trained using mean squared error (MSE) for the point estimation loss and KL-divergence between the variational distribution and the prior. The sum of the MSE and KL-divergence serves as a measure of distance between the variational distribution and the posterior. The number of epochs and batch size vary across experiments, while the learning rate is fixed at 0.01. As with HMC BNN, to improve the VI BNN's performance, (1) per training loop, only the weights of the epoch with the lowest loss are saved, and (2) 5 training loops are run for the same regression problem, with the weights reinitialised at the start of every loop. In the end, the best performing weights (i.e. the weights with the lowest loss) are chosen.

V. Results

In the given graphs, the dots represent the synthetic data points, the blue area represents the range of variation in the model's predictions, and the orange area represents the 95% confidence interval of the distribution of predictions. In each graph, the orange area is so close to the curve of average predictions that they are indistinguishable, which means most of the predictions are very close to the average.

The first set of experiments has the following training parameters for both models: 500 epochs with batch size 32 and learning rate 0.01. These values were chosen as they achieved sufficient predictive accuracy for an ANN (without Bayesian layers). The models were trained for 100 data points per problem. The results are presented in **Table I**, where each row represents result for one problem, starting from problem A at the top. Focusing on HMC BNN, the predictions show greater variation and hence more uncertainty when there are gaps or sparsity in the data, as seen in the intervals $[1, 3]$ for problem B and $[-2, 2]$ for problem B. In problem A, the predictions vary most at inflection points of the average prediction, where there is the greatest complexity in the data's distribution.

Turning to VI BNN, the poor average predictive accuract reveals that the number of epochs and/or batch size were insufficient for the model to properly train for the data. Nonetheless, it can observed that, unlike HMC BNN, the predictions do not show greater variation and hence uncertainty in gaps or sparsity, but instead show greater variation at the edges of the interval for which the data is defined, suggesting uncertainty about how the data extends beyond the given interval. In order to more effectively capture VI BNN's quantification of uncertainty, the VI BNN has been trained for 1500 epochs. The results are presented in **Table II** (shown only for the test data), where the average predictive accuracy is comparable to that of HMC BNN. Again, the predictions do not show greater variation and hence uncertainty in gaps or sparsity. Only in problem A is there greater variation in the noisier part of the data, in the interval $[-3, -1]$. Moreover, training for more epochs (e.g. 2000 epochs) tends to reduce the variation in predictions overall (see appendix E).

Observe that for both models in problem D, despite the presence of a gap in the data in the interval $[10, 14]$, both models show very low predictive uncertainty. To explore the cause, two experiments were run: (1) all the VI BNN's layers were set as Bayesian linear layers, and (2) a hidden Bayesian layer was added to the VI BNN in (1). The results of these experiments are given in appendix E.

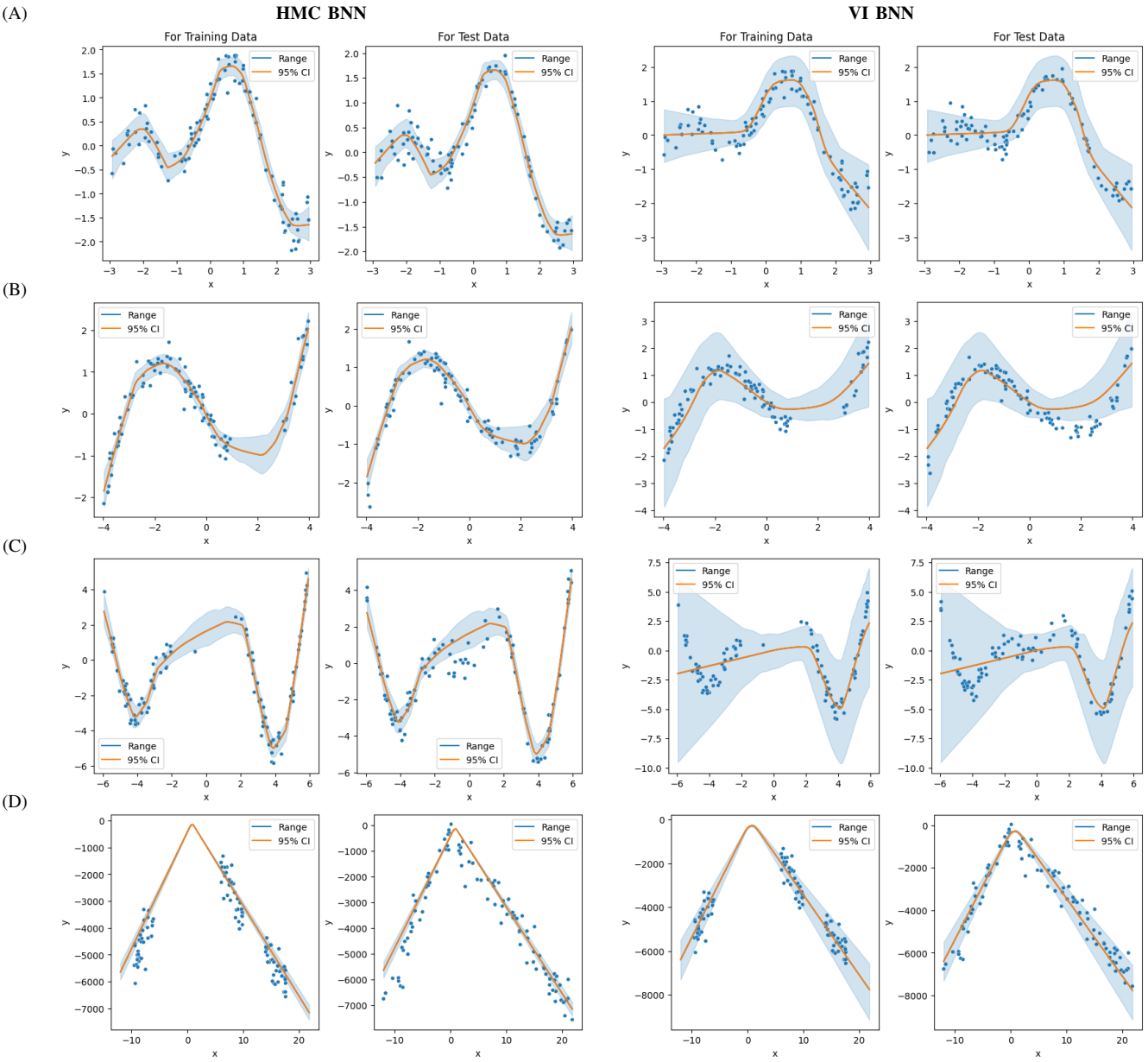

TABLE I
HMC BNN & VI BNN PERFORMANCE (500 EPOCHS)

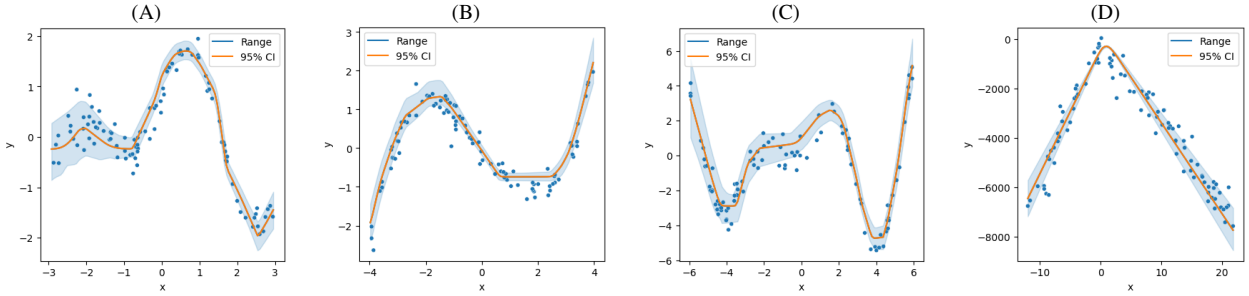

TABLE II
VI BNN PERFORMANCE (1500 EPOCHS, TEST DATA ONLY)

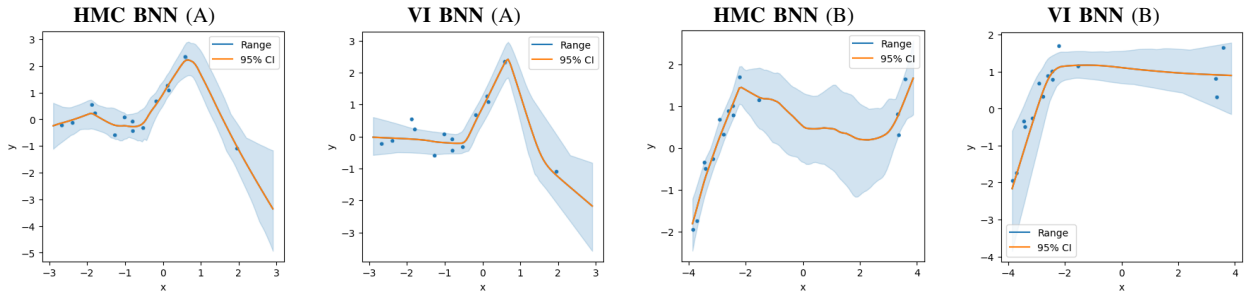

TABLE III
HMC BNN & VI BNN PERFORMANCE (500 & 1500 EPOCHS, SPARSE DATA)

In the next set of experiments, the models were trained for sparser data, i.e. 15 data points per problem instead of 100, without purposefully added gaps in the data. To achieve a comparable level of predictive accuracy, the functional model for HMC BNN and the VI BNN were trained for 500 and 2000 epochs respectively (batch size and learning rate being the same as before). The experiment was focused on problems A and B. The goal here was to test whether the models show greater predictive uncertainty due to the data's sparsity. The results are presented in **Table III**. In problem A and for both models, greater variation in predictions is observed where data is sparser, i.e. in the interval $[2, 3]$. In problem B, HMC BNN shows a much greater variation in predictions in the interval $[-2, 3]$, where there are no data points, which corresponds to a much greater uncertainty about its predictions in the absence of data. However, the variation in the predictions of VI BNN in problem B does not show the same correspondence between variation in the predictions and uncertainty about the predictions.

VI. CONCLUSION

Given the domain of synthetic regression problems with single-valued inputs, the performance of HMC BNN aligns reasonably well with expectations about the predictive uncertainty of the model, which suggests that taking HMC samples from a trained ANN is a promising approach to quantifying a model's predictive uncertainty. However, since only one layer of the model was distributed, the HMC samples are likely not representative of the distribution of potential parameterisations of the ANN as a whole, since ANNs are often sensitive to the initialisation of their weights. In other words, the distribution of the hidden layer's weights while keeping other layers constant would not sufficiently explore the multi-modal posterior of the ANN's weights as a whole. Hence, valuable further work to more rigorously study the effectiveness of HMC in deep learning would be to develop an HMC BNN implementation that can sample for the weights of the whole model, as done by Brian and Da Veiga (2022).

The VI BNN implementation used, which is based on adversarial BNN, shows less consistent and overall poorer performance. This may be a deficiency in the particular approximate BI method used, and hence, no broader conclusion about the effectiveness of approximate BI methods can be made. Hence, further work would be valuable in exploring the effectiveness of other approximate BI methods, such as Bayes-by-Backpropagation, optimisation using the Evidence Lower Bound (ELBO) and Monte Carlo drop-out.

REFERENCES

[1] Andrieu, C. and Thoms, J. (2008). A tutorial on adaptive MCMC. *Springer*. 18, pp. 343–373. doi:10.1007/s11222-008-9110-y.

[2] Betancourt, M. (2018). A Conceptual Introduction to Hamiltonian Monte Carlo. *arXiv preprint arXiv:1701.02434*.

[3] Brian, S. and Da Veiga, S. (2022). Benchmarking Bayesian neural networks and evaluation metrics for regression tasks. *arXiv preprint arXiv:2206.06779*.

[4] Carroll, C. (2019). *Hamiltonian Monte Carlo from scratch*. [online] Available from: https://colindcarroll.com/2019/04/11/hamiltonian-monte-carlo-from-scratch [Accessed 21 August 2024].

[5] Chandra, R. and Simmons, J. (2023). Bayesian neural networks via MCMC: a Python-based tutorial. *arXiv preprint arXiv:2304.02595*.

[6] Cook, J. D. (2020). *Leapfrog integrator*. [online] Available from: https://www.johndcook.com/blog/2020/07/13/leapfrog-integrator/.

[7] Foong, A. Y. K., Burt, D. R., Li, Y. and Turner, R. E. (2019). On the Expressiveness of Approximate Inference in Bayesian Neural Networks. *arXiv preprint arXiv:1909.00719*.

[8] Ganguly, A. and Earp, S. W. F. (2021). An Introduction to Variational Inference. *arXiv preprint arXiv:2108.13083*.

[9] janosh.dev (2019). *Training BNNs with HMC*. [online] Available from: https://janosh.dev/posts/hmc-bnn [Accessed 21 August 2024].

[10] Jospin, L. V., Laga, H., Boussaid, F., Buntine, W. and Bennamoun, M. (2020). Hands-on Bayesian Neural Networks – A Tutorial for Deep Learning Users. *arXiv preprint arXiv:2007.06823*.

[11] Kim, H. (2020). *Bayesian-Neural-Network-Pytorch*. [online] Available from: https://github.com/Harry24k/bayesian-neural-network-pytorch [Accessed 21 August 2024].

[12] Lee, S., Kim, H. and Lee, J. (2021). GradDiv: Adversarial Robustness of Randomized Neural Networks via Gradient Diversity Regularization. *arXiv preprint arXiv:2107.02425*.

[13] Martin. O. A., Kumar, R. and Lao, J. Bayesian Modeling and Computation in Python. Boca Ratón, Florida, USA: CRC Press, Taylor & Francis Group. ISBN: 978-0-367-89436-8.

[14] Murphy, K. P. (2012). '3. Generative models for discrete data'. *Machine Learning: A Probabilistic Perspective*. London, England: The MIT Press. pp. 65-87.

[15] Neal, R. (2011). 'Chapter 5: MCMC using Hamiltonian dynamics'. In: Brooks, S., Gelman, A., Jones, G., and Meng, X. *Handbook of Markov Chain Monte Carlo*. Boca Ratón, Florida, USA: CRC Press, Taylor & Francis Group. pp. 113-162.

[16] Nowling, R. J. (2016). *Deriving the Leapfrog Integrator*. [online] Available from: https://rnowling.github.io/math/2016/11/11/deriving-leapfrog.html [Accessed 23 August 2024].

[17] TensorFlow. (2023). *tfp.mcmc.SimpleStepSizeAdaptation*. [online] Available from: https://www.tensorflow.org/probability/api_docs/python/tfp/mcmc/SimpleStepSizeAdaptation [Accessed 21 August 2024].
[18] Wikipedia. (2024). *Leapfrog integration*. [online] Available from: https://en.wikipedia.org/wiki/Leapfrog_integration [Accessed 23 August 2024].
[19] Yao, J., Pan, W., Ghosh, S. and Doshi-Velez, F. (2019). Quality of Uncertainty Quantification for Bayesian Neural Network Inference. *arXiv preprint arXiv:1906.09686*.

APPENDIX

Appendix A: Use of Negative Log-Probability in HMC

Let $\theta = (\theta_1, \theta_2 ... \theta_k)$ be a parameterisation of a given generative model. Let m be the momentum, where $m = (m_1, m_2 ... m_k)$ (each m_i being the momentum in the ith dimension). Also, note that $P(\theta, m) = P(\theta|D)P(m)$, which means $\log P(\theta, m) = \log(P(\theta|D)P(m)) = \log P(\theta|D) + \log P(m)$. Hence:

$$-\log P(\theta, m) = -\log P(\theta|D) - \log P(m)$$

This is of the form $H(\theta, m) = K(\theta) + V(\theta)$, where $H(\theta, m) = -\log P(\theta, m)$ (Hamiltonian), $K(m) = -\log P(m)$ ("kinetic energy"), and $V(\theta) = -\log P(\theta|D)$ ("potential energy"). Thus, we have Hamilton's equations:

$$\frac{d\theta}{dt} = \frac{\delta H}{\delta m}, \quad \frac{dm}{dt} = -\frac{\delta H}{\delta \theta}$$

Hamilton's equations describe the state of a physical system with one body of a fixed mass moving in a space of k dimensions ($k \geq 1$). Now, even though Hamilton's equations were meant to deal with physical spaces, we can accurately extend their application to analogous simulated or abstract spaces, such as a high-dimensional parameter space of a Bayesian model.

In the context BI, the "position" of the sample can be thought of as the location of a particle, and the "momentum" provides the force needed to move the particle through the parameter space of the distributions. As shown above, the Hamiltonian for BI has the negative log-probability of the posterior. However, since the negative log-probability preserves the distribution of the posterior's probability mass as well as the posterior's modes (albeit that the modes are represented by minima instead of maxima), samples from the the negative log-probability of the posterior follow the same distribution as samples the posterior. Hence, Hamilton's equations provide a valid means to explore the posterior.

Appendix B: Reasoning for HMC's Acceptance Criterion

The HMC algorithm is given below is drawn mostly from Carroll (2019) with changes made to more closely reflect the mathematics behind the algorithm. The names and symbols used are explained in the table after the algorithm:

Algorithm 1 Run HMC Sampler

H **input** $n, -\log p, \theta_0, T, \Delta t$
  samples $\leftarrow$ array of size n
  $g \leftarrow \text{grad}(-\log p)$
  for $i \in (0, 1, 2 ... n-1)$ **do**
    $m_0 \leftarrow \text{draw}(\mathcal{N}(0, \sigma I))$
    $\theta_T, m_T \leftarrow \text{leapfrog}(\theta_0, m, g, T, \Delta t)$
    $H_{\text{old}} \leftarrow -\log(p)(\theta_0) - \log \mathcal{N}(0, \sigma I)(m_0)$
    $H_{\text{new}} \leftarrow -\log(p)(\theta_T) - \log \mathcal{N}(0, \sigma I)(m_T)$
    $\alpha \leftarrow e^{H_{\text{old}} - H_{\text{new}}}$
    if $\text{draw}(\mathcal{U}(0,1)) < \alpha$ **then**
      samples$[i] \leftarrow \theta_T$
    else
      samples$[i] \leftarrow \theta_0$
    end if
  end for
return samples

n	Number of samples to draw
p	Target probability distribution
$-\log p$	Negative log-probability of p
$T, \Delta t$	Number and size of leapfrog steps
θ_0, m_0	Current position and momentum
θ_T, m_T	Proposed position and momentum
α	Acceptance probability of proposal
H_{old}	Hamiltonian function for θ_0, m_0
H_{new}	Hamiltonian function for θ_T, m_T
$\mathcal{N}(\mu, \sigma)$	Normal distribution
$\mathcal{U}([a, b])$	Uniform distribution
grad	Returns callable gradient
draw	Function to draw from given distribution
leapfrog	Leapfrog integrator

HMC can be explained using a physical analogy, wherein $-\log p$ defines the contours of a force field (e.g. a gravitational field) along which a body — the sampler, in our case — can travel, with each position representing a sample. Starting from the current position θ_0, the leapfrog integrator is a symplectic integrator that uses the gradient of $-\log p$ and the current momentum m_0 to simulate the exact trajectory of the sampler along the contours of $-\log p$ for $L = \frac{T}{\Delta t}$ discrete time steps. Thus, a symplectic integrator discretises the Hamiltonian equations used to explore the posterior. If the momentum is well-chosen, the trajectory travels through positions with the same or similar acceptance probabilities as the current position.

However, despite its accuracy, a symplectic integrator is likely to introduce at least some errors in the calculation of trajectories (Betancourt, 2018). Furthermore, the momentum may be sub-optimal. To account for such errors, a proposed sample is accepted based on the Metropolis criterion, i.e. a proposed sample is accepted with the acceptance probability $\alpha = \min(1, e^{H_{\text{old}} - H_{\text{new}}})$ (Neal, 2011). The mathematical

details and reasoning are given in the following paragraphs.

Let θ be the current position (i.e. the current state of the Markov chain sampling the Bayesian model's parameterisations). Let m be the current momentum. Likewise, let θ^* and m^* be the proposed position and momentum. Also, let $H(\theta, m)$ be the Hamiltonian for position θ and momentum m. Then, we have the following cases:

- $e^{H(\theta,m)-H(\theta^*,m^*)} \geq 1 \implies H(\theta^*, m^*) \geq H(\theta, m)$
- $e^{H(\theta,m)-H(\theta^*,m^*)} \leq 1 \implies H(\theta^*, m^*) \leq H(\theta, m)$

$H(\theta^*, m^*) \geq H(\theta, m)$ means the proposed state is from a region in the posterior of equal or higher probability mass, which means it should always be accepted, because we want to sample more from regions of equal or higher probability mass. $H(\theta^*, m^*) < H(\theta, m)$ means the proposed state is from a region of lower probability mass, which means it should be accepted only probabilistically, with the probability of accepting it being proportional to its closeness to the current state (in terms of probability density), because we want there to be a lower but non-zero chance of sampling from a sparser region after sampling from a denser region, with the condition that the lower the density, the lower the chance.

Appendix C: Leapfrog Integration

Leapfrog integration is a method for numerically any integrating differential equations (DE) of the form $\frac{d^2x}{dt^2} = f(x)$, i.e. $x'' = f(x)$ (Wikipedia, 2024). Here, x is a function of t. In general, integrating a DE means solving it, i.e. finding the functions for the differentials so as to satisfy the DE. In the case of leapfrog integration, integrating a DE of the form $x'' = f(x)$ means solving for the function x (note that f is given). More precisely, for discrete steps along t, the leapfrog integrator tries to approximate the values for x across for a given number of steps. Typically, x is position and t is time; this analogy can be extended to any use-case.

The leapfrog integrator is a numerical method for solving DEs of the form: $x'' = f(x)$. Here, note the following: (1) x is a function of t (typically, x is position and t is time), (2) we are dealing with a k-dimensional space, where $k \geq 1$, and hence, (3) if $x(t)$ is the position, it would be an array of coordinates. The leapfrog integrator is used when there is no analytical equation for $x(t)$ that satisfies the DE, or at least when the analytical equation is too complex to derive. The idea is to use a numerical integration algorithm to compute values of $x(t)$ on discrete time steps Δt.

In the following equations, we can see how the values are approximated for x. Note that here, we are using the analogy (which can be accurately translated to posterior sampling) of x as position, v as velocity, a as acceleration and t as time. The complete derivation is based on the derivation presented by Nowling (2016), with the notation modified according to the algorithm presented by Cook (2020). The leapfrog integration algorithm can be summarised as follows:

1) Calculate a for the current time step
2) Calculate x for the next time step
3) Calculate a for the next time step
4) Calculate v for the next time step

Mathematically, the above steps are:

1) $a(t) = -f(x(t))$
2) $x(t + \Delta t) = x(t) + v(t)\Delta t + \frac{1}{2}a(t)\Delta t^2$
3) $a(t + \Delta t) = -f(x(t + \Delta t))$
4) $v(t + \Delta t) = v(t) + \frac{1}{2}(a(t) + a(t + \Delta t))\Delta t$

Here, x, v and a are related as follows:

- $x''(t) = a(t) = f(x)$
- $v''(t) = a'(t)$
- $x''(t) = v'(t) = a(t)$
- $x'(t) = v(t)$

The above relationships can be easily understood if one considers x as position, v as velocity, a as acceleration, and t as time. Note that position, velocity and acceleration are all represented by vectors if the number of spatial dimensions are greater than one.

Appendix D: VI with Evidence Lower Bound (ELBO)

KL-divergence is a well-established measure of distance used in VI. If used, the optimisation function would based on the KL-divergence between $q_\phi(\theta)$ and $p = P(\theta|D)$. Given that Θ is the set of all hypothetical parameterisations of a given model, the KL-divergence between $q_\phi(\theta)$ and $P(\theta|D)$ is given by the following:

$$KL(q_\phi(\theta)||P(\theta|D)) = \int_{\theta\in\Theta} q_\phi(\theta) \log \frac{q_\phi(\theta)}{P(\theta|D)} d\theta$$

However, $P(\theta|D)$ is the posterior being approximated and thus cannot be in the optimisation function. But there exists a function derived from KL-divergence, namely the Evidence Lower Bound (ELBO), that uses only the variational distribution q_ϕ and the joint distribution $P(\theta, D) = P(D|\theta)|P(\theta)$, i.e. the unnormalised posterior. Note that these distributions are known, since q_ϕ and the prior $P(\theta)$ are chosen and since the likelihood $P(D|\theta)$ is based on a chosen generative model. Mathematically, ELBO is given as follows (Jospin et al., 2020):

$$\int_{\theta\in\Theta} q_\phi(\theta) \log \frac{p(\theta, D)}{q_\phi(\theta)} d\theta$$

ELBO is derived such that maximising it achieves the same optimisation as minimising KL-divergence. In practice, the above integral is estimated through numerical methods, e.g. averaging Monte Carlo samples drawn from the variational and joint distributions and plugging them into the ELBO formula[4] (Martin et al., 2021). For the derivation of ELBO, check *conceptual-notes/bayesian-inference/sampling-methods/variational-inference-vi* in *github.com/pranigopu/mastersProject*, the repository associated with this paper.

Appendix E: Results of Further Experiments

1) VI BNN Trained for 2000 Epochs: The following set of graphs are to demonstrate that increasing the number of training epochs for the VI BNN implementation used tends to decrease the variation in its predictions, which means it likely does not quantify predictive uncertainty more effectively with more training epochs.

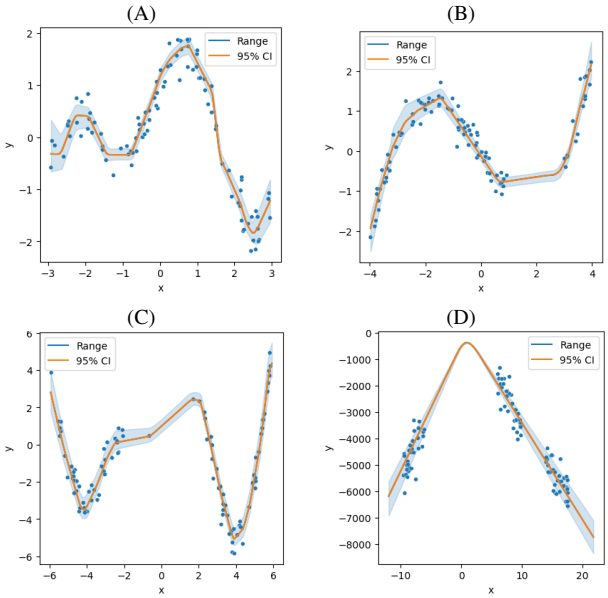

TABLE IV
VI BNN PERFORMANCE (2000 EPOCHS, TRAINING DATA ONLY)

2) VI BNN Experiments for Problem D: The following graphs are the result of running VI BNN with all its layers set as Bayesian linear layers. Two architectures were tried, with one and two hidden layers respectively.

[4]Here, the integral becomes a summation.

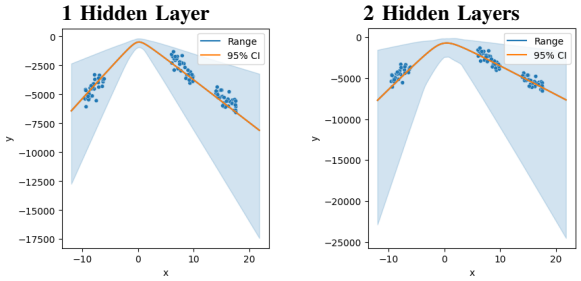

TABLE V
VI BNN FOR PROBLEM D (FULL BAYESIAN, TRAINING DATA ONLY)

As can be observed, the quantification of the predictive uncertainty for the gap in the data does not improve. This could be an issue with the particular form of VI BNN used, in which case uncertainty quantification may improve if more complex architectures were used for HMC BNN.