Comparative Study of BNN Implementations

Pranav Narendra Gopalkrishna
231052045
Paulo Rauber
MSc. Artificial Intelligence

***Abstract*—To be filled**

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

I. Introduction

The focus is on two kinds of BNNs: (1) BNN via an exact Bayesian inference method, and (2) a BNN via an approximate Bayesian inference method. Due to their sound theories and practical successes, HMC (a class of MCMC methods) and VI are used for an exact and approximate Bayesian inference method respectively.

II. Related Work

Jospin et al. give an in-depth introduction to Bayesian inference for deep learning, covering a wide range of topics from Bayesian inference algorithms to performance metric for BNNs. Bayesian inference 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 Bayesian inference 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 Bayesian inference methods and 2 non-Bayesian frameworks. On the other hand, Foong et. al (2019) focus not only on evaluating approximate Bayesian inference methods but also exploring pathologies arising due to approximation.

III. Relevant Concepts

A. Bayesian Inference

Bayesian inference is the process of inferring a generative model[1] that explains the observed data. Bayesian inference 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, Bayesian inference seeks the posterior probability of the model's parameterisations based on D. Mathematically, Bayesian inference 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 Bayesian inference 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 Bayesian Inference

A exact Bayesian inference method (e.g. MCMC) is one that is theoretically guaranteed to converge to posterior. An approximate Bayesian inference 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 Bayesian inference 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'
4) If θ' is accepted, set θ' as the current sample
5) Repeat from step 2 for a fixed number of iterations

[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.

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 the appendix). 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 behavior 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 better sample proposal distribution (Martin et al., 2021). The HMC algorithm is given below (Carroll, 2019):

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

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**
      $\text{samples}[i] \leftarrow \theta_T$
    else
      $\text{samples}[i] \leftarrow \theta_0$
    end if
  end for
return samples

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, 2012). The mathematical details and reasoning are given in the appendix.

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 Bayesian inference, VI is an approximate Bayesian inference 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). However, the focus shall be on the method used in *torchbnn*, a PyTorch-based implementation of BNNs (Lee et al., 2022), since this method can be more easily applied to a neural network. 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 Bayesian inference method, (3) a BNN via an approximate Bayesian inference method, and (4) methods of evaluating the performance of the aforementioned BNNs. The implementations are in Python.

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	$y_i = 0.1x_i^3 + \epsilon_i$
Error term	$\epsilon_i \sim \mathcal{N}(0, 0.25)$
Train inputs	$x_i \sim \mathcal{U}([-4, 1] \bigcup [1, 4])$
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	$\epsilon_i \sim \mathcal{N}(0, 0.5)$
95% train inputs	$x_i \sim \mathcal{U}([-6, 2] \bigcup [2, 6])$
5% 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	$x_i \sim \mathcal{U}([-12, 22])$

Note that 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 $\mathcal{N}(0, I_d)$.

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 Bayesian inferences; 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) is used as the generative model, i.e. functional model. 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 50 epochs and a batch size of 2, 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) more than one 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 10000, since this was sufficient to lead to more than 1000 accepted sample proposals. 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 5000.

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 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 (tfp.mcmc.SimpleStepSizeAdaptation), 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, 2012, p. 137). Having chosen the step-

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

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, 2012, p. 137) and is hence chosen.

3) Bayesian Inference 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 $y_i \sim \mathcal{N}(f_\theta(x_i), \tau)$. 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 Bayesian inference. 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 the package *torchbnn* (Lee et al., 2022), 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 model is trained for 3000 epochs with a batch size of 100.

REFERENCES

[1] Jospin, L. V., Laga, H., Boussaid, F., Buntine, W. and Bennamoun, M. (2020). "Hands-on Bayesian Neural Networks – A Tutorial for Deep Learning Users". Available at: https://arxiv.org/pdf/2007.06823.

[2] Martin, O. A., Kumar, R. and Lao, J. (2021). "Bayesian Modeling and Computation in Python". Available at: https://bayesiancomputationbook.com.

[3] Chandra, R. and Simmons, J. (2023) "Bayesian neural networks via MCMC: a Python-based tutorial". Available at: https://arxiv.org/pdf/2304.02595.

[4] Brian, S. and Da Veiga, S. (2022). "Benchmarking Bayesian neural networks and evaluation metrics for regression tasks". Available at: https://arxiv.org/pdf/2206.06779.

[5] Yao, J., Pan, W., Ghosh, S. and Doshi-Velez, F. (2019). "Quality of Uncertainty Quantification for Bayesian Neural Network Inference". Available at: https://arxiv.org/pdf/1906.09686.

[6] Foong, A. Y. K., Burt, D. R., Li, Y. and Turner, R. E. (2019). "On the Expressiveness of Approximate Inference in Bayesian Neural Networks". Available at: https://arxiv.org/pdf/1909.00719.

[7] Betancourt, M. (2018). "A Conceptual Introduction to Hamiltonian Monte Carlo". Available at: https://arxiv.org/pdf/1701.02434.

[8] Ganguly, A. and Earp, S. W. F. (2021). "An Introduction to Variational Inference". Available at: https://arxiv.org/pdf/2108.13083.

[9] Neal, R. (2012). "Chapter 5: MCMC using Hamiltonian dynamics". *Handbook of Markov Chain Monte Carlo*. Available at: https://arxiv.org/pdf/1206.1901.

[10] Carroll, C. (2019). "Hamiltonian Monte Carlo from scratch". Available at: https://colindcarroll.com/2019/04/11/hamiltonian-monte-carlo-from-scratch.

[11] "tfp.mcmc.SimpleStepSizeAdaptation". *Tensorflow Documentation*.

[12] Andrieu, C. and Thoms, J. (2008). "A tutorial on adaptive MCMC". Available at: https://link.springer.com/article/10.1007/s11222-008-9110-y

[13] ee, S., Kim, H. and Lee, J. (2022). "Graddiv: Adversarial robustness of randomized neural networks via gradient diversity regularization". *IEEE Transactions on Pattern Analysis and Machine Intelligence*. IEEE.

APPENDIX

Appendix A: VI with KL-Divergence

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 (the derivation is in the appendix):

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

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 (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). 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).

[4] Here, the integral becomes a summation.