Utilising Gradient-Based Proposals Within Sequential Monte Carlo Samplers for Training of Partial Bayesian Neural Networks

Anonymous authors
Paper under double-blind review

Abstract

Partial Bayesian neural networks (pBNNs) have been shown to perform competitively with fully Bayesian neural networks while only having a subset of the parameters be stochastic. Using sequential Monte Carlo (SMC) samplers as the inference method for pBNNs gives a non-parametric probabilistic estimation of the stochastic parameters, and has shown improved performance over parametric methods. In this paper we introduce a new SMC-based training method for pBNNs by utilising a guided proposal and incorporating gradient-based Markov kernels, which gives us better scalability on high dimensional problems. We show that our new method outperforms the state-of-the-art in terms of predictive performance and optimal loss. We also show that pBNNs scale well with larger batch sizes, resulting in significantly reduced training times and often better performance.

1 Introduction

Bayesian Neural Networks (BNNs) are a class of machine learning models that incorporate uncertainty quantification into deep learning. Previous research has shown the benefit Bayesian methods can bring to certain problems within deep learning Gal et al. (2017). However, computing the exact posterior distributions of BNNs is a difficult task as traditional methods such as Markov chain Monte Carlo (MCMC) Hastings (1970) are computationally poorly suited to exploring high dimensional spaces and dealing with large amounts of data. Parametric methods such as variational inference are better suited to these difficulties, but only give an approximation to the posterior distribution. These spaces have been found to be highly complex Izmailov et al. (2021a) and therefore variational methods often give a poor approximation of the posterior.

Sequential Monte Carlo (SMC) samplers Doucet et al. (2001) are an alternative to MCMC methods which also provide an empirical estimate of the posterior distribution. SMC samplers are instantly parallelisable Varsi et al. (2021b) and therefore can take advantage of the GPU resources commonly used in machine learning to speed up the training process. MCMC methods often require a warm-up period to adapt the hyperparameters, after which the chains can be parallelised. However, the hyperparameters must remain fixed after this warm-up period to obey stationarity. This means that SMC samplers can be more flexible than traditional MCMC methods when it comes to continual hyperparameter tuning. However, as a parametric Bayesian method, SMC samplers still struggle with similar issues to MCMC in terms of dimensionality and dataset size.

In recent years, partial Bayesian neural networks (pBNNs) have been proposed as a potential "model solution" to the high dimensionality issue Sharma et al. (2023). pBNNs only consider a subset of the parameters to be stochastic and despite this reduced dimensionality, have been shown to have comparable performance to that of fully Bayesian neural networks (BNNs) Sharma et al. (2023). More formally, let $f(\mathbf{x}; \boldsymbol{\theta}, \boldsymbol{\psi})$ be a neural network model governed by a set of deterministic parameters ($\boldsymbol{\psi}$) and a set of stochastic parameters ($\boldsymbol{\theta}$) initially generated from a prior distribution $q_0(\boldsymbol{\theta})$. If we have a dataset $(\mathbf{x}_n, \mathbf{y}_n)_{n=1}^N$ with a likelihood function $p(\mathbf{y}_n|\boldsymbol{\theta}, \boldsymbol{\psi})$, then our training procedure has two objectives; to learn the deterministic parameters from the dataset and to compute the posterior distribution $p(\boldsymbol{\theta}|\mathbf{y}_{1:N}, \boldsymbol{\psi})$. It has been shown in previous

work Sharma et al. (2023) that placing the uncertainty on the first layer of the pBNN empirically gives the best output. This may be because the majority of the randomness is aleatoric and originates from the data. Therefore it is difficult, when applying VI, to chose a parametric distribution to model the uncertainty correctly. This difficulty motivates non-parametric modeling of the uncertainty.

Using MCMC for the stochastic parameters would be theoretically invalid, as the target necessarily changes at each iteration due to the update in the deterministic parameters. SMC is a more flexible sampling approach and is therefore more suited this type of problem where the target is iteration dependent and leads us to the main motivation for studying and applying SMC samplers to pBNNs.

In this paper, we introduce gradient-based Markov kernels into the pBNN training process to help us more efficiently navigate the high dimensional spaces encountered in neural networks (NNs). More specifically, we introduce a new class of guided SMC samplers with gradient-based Markov kernels for training pBNNs. We then analyse both performance and runtime of the different proposal methods using multiple batch sizes. We demonstrate that we are able to use larger batch sizes without compromising performance, allowing for significantly reduced training times.

The paper is structured as follows. We provide a basic background of how SMC samplers work and how to apply them to a pBNN setting (Section 2). We then introduce the Guided Open Horizon SMC framework and how to incorporate gradient-based Markov kernels which will be compared to the random walk (RW) approach in later experiments (Section 3). The experimental set up, datasets and parameters are given (Section 4). We discuss the performance of the approaches on loss, classification accuracy (where appropriate) and runtime (Section 5). Finally, conclusions and further work suggestions are provided (Section 6).

2 Partial Bayesian Neural Network Training by Sequential Monte Carlo Samplers

2.1 Sequential Monte Carlo Samplers

SMC samplers are a subset of Bayesian inference algorithms which are often used when directly sampling from the posterior distribution is difficult. At the heart, SMC samplers sample from a sequence of distributions and approximate the distribution via a weighted set of samples (particles).

We aim to sample from a target distribution $\pi(\boldsymbol{\theta})$. We initially sample a set of J particles from a prior distribution with corresponding weights

$$\{\boldsymbol{\theta}_0^{(j)}, \mathbf{w}_0^{(j)}\}_{i=1}^{J} \sim q_0(\cdot). \tag{1}$$

Typically each particle is assigned an initial weight

$$\mathbf{w}_0^{(j)} = \frac{\pi(\boldsymbol{\theta}_0^{(j)})}{q_0(\boldsymbol{\theta}_0^{(j)})}. \tag{2}$$

After this initialisation, we start the main sampling loop, running for T iterations. At each new time step t, the weights are normalised via

$$\tilde{\mathbf{w}}_t^{(j)} = \frac{\mathbf{w}_t^{(j)}}{\sum_{j=1}^{J} \mathbf{w}_t^{(j)}}. \tag{3}$$

In order to avoid particle degeneracy, we employ resampling into the SMC process. Resampling happens when the effective sample size J_{eff} drops below a certain threshold. Typically this threshold is set to half the number of samples. i.e. we resample when

$$\frac{J}{2} > J_{\text{eff}} = \frac{1}{\sum_{i=1}^{J} (\tilde{\mathbf{w}}_t^{(j)})^2}. \tag{4}$$

Many different forms of resampling can be used such as stratified, residual Liu & Chen (1998) and systematic Kitagawa (1996). In our implementation we have used the multinomial resampling method Douc et al. (2005). If resampling occurs, the weights are then set to $\frac{1}{J}$.

Algorithm 1 SMC sampler for T iterations and J samples.

Sample $\{\boldsymbol{\theta}_0^{(j)}\}_{j=1}^{J}$ from $q_0(\cdot)$
Set initial weights $\mathbf{w}_0^{(j)}$ using equation 2
for $t = 1$ **to** T **do**
  for $j = 1$ **to** J **do**
    Normalise weights using equation 3
  end for
  Calculate J_{eff} using equation 4
  if $J_{\text{eff}} < J/2$ **then**
    Resample $[\boldsymbol{\theta}_t^{(1)}...\boldsymbol{\theta}_t^{(J)}]$ with probability $[\tilde{\mathbf{w}}_t^{(1)}...\tilde{\mathbf{w}}_t^{(J)}]$
    Reset all weights to $\frac{1}{J}$
  end if
  for $j = 1$ **to** J **do**
    $\boldsymbol{\theta}_t^{(j)} \sim q_t^{\theta}(\cdot|\boldsymbol{\theta}_{t-1}^{(j)})$
    Update sample weights $\mathbf{w}_t^{(j)}$ using equation 6
  end for
end for

New particles are proposed using a Markov kernel

$$\boldsymbol{\theta}_t^{(j)} \sim q_t^{\theta}(\cdot|\boldsymbol{\theta}_{t-1}^{(j)}) \tag{5}$$

which is a chosen method used to propagate the samples generated at $t-1$. The choice of this kernel will be covered in later sections.

Once these new samples have been generated, we weight them according to the following update rule:

$$\mathbf{w}_t^{(j)} = \mathbf{w}_{t-1}^{(j)} \frac{\pi(\boldsymbol{\theta}_t^{(j)})}{\pi(\boldsymbol{\theta}_{t-1}^{(j)})} \frac{L_t^{\theta}(\boldsymbol{\theta}_{t-1}^{(j)}|\boldsymbol{\theta}_t^{(j)})}{q_t^{\theta}(\boldsymbol{\theta}_t^{(j)}|\boldsymbol{\theta}_{t-1}^{(j)})}, \tag{6}$$

where $L_t^{\theta}(\boldsymbol{\theta}_{t-1}|\boldsymbol{\theta}_t)$ is denoted as the L-kernel, also known as the backward kernel. It is worth noting that if resampling has occurred at the current iteration, then when performing the weight update, the previous iteration weight is $\frac{1}{J}$.

A thorough overview of this concept and SMC samplers as a whole can be found in Del Moral et al. (2006). The pseudocode for an SMC sampler implementation can be found in Algorithm 1.

2.1.1 Gradient Based Proposals for SMC Samplers

The RW kernel is computationally efficient due its ability to propose new moves without gradient evaluations. However, it has been shown to struggle in high dimensional spaces. Gradient based Markov kernels such as the Metropolis adjusted Langevin algorithm (MALA) are also commonly used in the MCMC Roberts & Tweedie (1996) literature and have also been adapted for use in SMC methods in recent years Rosato et al. (2024).

MALA combines the Metropolis-Hastings acceptance criteria with Langevin dynamics, which utilises the gradient information to propose new states in a way that efficiently explores the target distribution. Langevin dynamics describes the evolution of a particle under both deterministic forces (gradient of the log-posterior) and stochastic forces (Gaussian noise). The proposed state is computed as:

$$\boldsymbol{\theta}' = \boldsymbol{\theta} + \frac{h^2}{2} \nabla \log \pi(\boldsymbol{\theta}) + h\mathbf{P} \tag{7}$$

where h is the step size, $\nabla \log \pi(\boldsymbol{\theta})$ is the gradient of the log-posterior at $\boldsymbol{\theta}$, and $\mathbf{P} \sim \mathcal{N}(0, I)$ is Gaussian noise. The proposed state $\boldsymbol{\theta}'$ is accepted with a probability given by the Metropolis-Hastings acceptance

Algorithm 2 SMC Sampler with Langevin Dynamics for K iterations and N samples.

Sample $\{\boldsymbol{\theta}_0^{(j)}\}_{j=1}^{J} \sim q_0(\cdot)$
Set initial weights $\mathbf{w}_0^{(j)}$ to equation 2
for $t = 1$ **to** T **do**
  for $j = 1$ **to** J **do**
    Normalise weights using equation 3
  end for
  Calculate J_{eff} using equation 4
  if $J_{\text{eff}} < J/2$ **then**
    Resample $[\boldsymbol{\theta}_t^{(1)} ... \boldsymbol{\theta}_t^{(J)}]$ with probability $[\tilde{\mathbf{w}}_t^{(1)} ... \tilde{\mathbf{w}}_t^{(J)}]$
    Reset all weights to $\frac{1}{J}$
  end if
  for $j = 1$ **to** J **do**
    $\mathbf{P} \sim \mathcal{N}(0, 1)$
    $\boldsymbol{\theta}_t^{(j)} = \boldsymbol{\theta}_{t-1}^{(j)} + \frac{h^2}{2} \nabla \log \pi(\boldsymbol{\theta}_{t-1}^{(j)}) + h\mathbf{P}$
    Update sample weights $\mathbf{w}_t^{(j)}$ using equation 6
  end for
end for

criterion:

$$\alpha(\boldsymbol{\theta}, \boldsymbol{\theta}') = \min\left(1, \frac{\pi(\boldsymbol{\theta}')q(\boldsymbol{\theta}|\boldsymbol{\theta}')}{\pi(\boldsymbol{\theta})q(\boldsymbol{\theta}'|\boldsymbol{\theta})}\right) \tag{8}$$

$$\boldsymbol{\theta} = \begin{cases} \boldsymbol{\theta}' & \text{if } u < \alpha(\boldsymbol{\theta}, \boldsymbol{\theta}'), \\ \boldsymbol{\theta} & \text{otherwise.} \end{cases} \tag{9}$$

where $q(\boldsymbol{\theta}'|\boldsymbol{\theta})$ is the proposal density of equation 7 and u is a random variable drawn from a uniform distribution $u \sim \mathcal{U}(0, 1)$. This step ensures that the Markov chain has the desired stationary distribution.

In our implementation of the SMC Sampler, we do not use the Metropolis criteria, instead relying on the weight update to decide how much each particle contributes to the mean and variance estimate which means we can forgo the accept/reject criteria normally implemented. Without the acceptance criteria the distribution we are targeting would not be the stationary one. However, if we choose an L-kernel which marginalises out the previous targeted distributions, we ensure that we draw samples from the posterior. This allows us to use the unadjusted Langevin dynamics as a Markov kernel within the sequential importance sampling framework. The weight update for an SMC Sampler with Langevin dynamics becomes equal to

$$\mathbf{w}_t^{(j)} = \mathbf{w}_{t-1}^{(j)} \frac{\pi(\boldsymbol{\theta}_t^{(j)})}{\pi(\boldsymbol{\theta}_{t-1}^{(j)})} \frac{L_t^P(-\mathbf{P}^{(j)*})}{q_t^P(\mathbf{P}_{t-1}^{(j)})} \tag{10}$$

where $\mathbf{P}_{t-1}$ is the Gaussian noise sampled at $t-1$ and $\mathbf{P}^*$ is this noise having undergone an update step. The use of this in an SMC context is given in Algorithm 2. The full justification of this proposal and details on the L-kernel associated with it can be found here Rosato et al. (2024) and for completeness a brief derivation is given in Appendix 1.

2.2 Stochastic Gradient and Open Horizon SMC

The SMC sampler in Algorithm 1 can be applied to sample the posterior distribution of the stochastic part of the pBNN with $\pi(\boldsymbol{\theta}, \boldsymbol{\psi}) = p(\mathbf{y}_n|\boldsymbol{\theta}, \boldsymbol{\psi})q_0(\boldsymbol{\theta}) \propto p(\boldsymbol{\theta} \mid \mathbf{y}_{1:N}, \boldsymbol{\psi})$. However, it remains to be shown how to learn the deterministic part of the pBNN, in particular, when there is a large number of data-points D. We introduce a scalable framework via Stochastic Gradient Sequential Monte Carlo (SGSMC) which uses the stochastic approximation method to approximate the marginal log-likelihood.

If we denote a batch size M where $1 \leq M \leq D$ then let $\mathbf{S}_M := \{\mathbf{S}_M(1), \mathbf{S}_M(2), \ldots, \mathbf{S}_M(M)\}$ be a set of batch indices. We can then approximate our log-likelihood with respect to $\boldsymbol{\psi}$ as

$$\log p(\mathbf{y}_{1:N}|\boldsymbol{\psi}) \approx \frac{N}{M} \log p(\mathbf{y}_{\mathbf{S}_M}|\boldsymbol{\psi}), \tag{11}$$

by only considering the subset of data $\mathbf{y}_{\mathbf{S}_M}$. Moreover, the SMC sampler also allows us to simultaneously compute an approximation of the gradient of this approximation to the log-likelihood

$$\frac{N}{M} \nabla_{\boldsymbol{\psi}} \log p(\mathbf{y}_{\mathbf{S}_M}|\boldsymbol{\psi}) = \frac{N}{M} \mathbb{E}_{p(\boldsymbol{\theta}|\mathbf{y}_{\mathbf{S}_M}, \boldsymbol{\psi})} \left[\nabla_{\boldsymbol{\psi}} \log p(\mathbf{y}_{\mathbf{S}_M}|\boldsymbol{\theta}; \boldsymbol{\psi})\right] \approx \frac{N}{M} \sum_{j=1}^{J} \tilde{\mathbf{w}}^{(j)} \nabla \log p(\mathbf{y}_{\mathbf{S}^{\mathbf{M}}}|\boldsymbol{\theta}^{(j)}; \boldsymbol{\psi}). \tag{12}$$

Note that this approximation is biased due to the presence of the normalised weights, $\tilde{\mathbf{w}}^{(j)}$.

This stochastic gradient is computed using the subdataset $\mathbf{y}_{\mathbf{S}_M}$, and the expectation is taken over the random batch indices. The SGSMC algorithm iteratively updates the particles and weights using the stochastic gradient approximation. At each iteration of the gradient optimisation, a subdataset is sampled, and then the SMC sampler is applied on this subdataset to estimate the gradient and update the pBNN deterministic parameters. However, the SGSMC algorithm when implemented naively is computationally demanding, and it does not really sample the posterior $p(\boldsymbol{\theta} \mid \mathbf{y}_{1:N}, \boldsymbol{\psi})$. More specifically, at each iteration, the algorithm *sequentially* loops over the data points in the subdata, and the posterior distribution we obtain is a crude approximate $p(\boldsymbol{\theta} \mid \mathbf{y}_{\mathbf{S}_M}, \boldsymbol{\psi})$ on this subdata. This motivated Zhao et al. (2024) to come up with a so-called open-horizon SMC (OHSMC) sampler to tackle these issues.

At its heart, OHSMC merges the loop of the stochastic optimisation for $\boldsymbol{\psi}$ and the loop of the SMC sampling in a principled way. Specifically, at each iteration of OHSMC, the algorithm randomly samples a subdataset, and then approximates the gradient equation 12 and the target posterior distribution concurrently. Crucially, the OHSMC can process the subdataset in parallel, while SGSMC sequentially loops over the elements in the subdataset. Empirically, OHSMC also provides better approximation to the target posterior distribution by linking the posterior distribution estimates across iterations. Unlike SGSMC, which independently restarts from the prior distribution at each step, OHSMC uses the posterior from the previous iteration as the starting point for the next. This warm-start strategy improves computational efficiency and maintains continuity in the posterior distribution estimation.

Both SGSMC and OHSMC significantly reduce the computational load by processing only subsets of the data, making them suitable for large datasets. The algorithms can be adapted to various latent variable models, enhancing their applicability across different domains. OHSMC offers practical advantages in implementation, particularly in environments like JAX and TensorFlow, by avoiding dynamic input size issues inherent in SGSMC. Overall, the SGSMC and OHSMC methods provide robust, scalable solutions for Bayesian inference in large datasets, balancing computational efficiency with accuracy. However, there are a few challenges in OHSMC which remain unsolved, laying the groundwork for our improvements.

3 Guided OHSMC

The main criticism of the original OHSMC is that it is based on a bootstrap version of Algorithm 1. Namely, they invoke a Markov kernel that leaves invariant with respect to the *previous* posterior distribution. We hereby adapt it in Algorithm 3 by making the Markov kernel leave invariant the *current* posterior distribution. This gives us a guided improvement of the original OHSMC by incorporating information from the target posterior distribution. Consequently, this guided version yields a more effective importance proposal leading to better statistical efficiency, as evidenced in the new weight update equation 6. Another improvement we deliver is better scalability in high-dimensional $\boldsymbol{\theta}$. Unlike Zhao et al. (2024) which essentially uses a RW Markov kernel Metropolis et al. (1953) Givens & Raftery (1996), we propose to use gradient-based Markov kernels, in particular, the unadjusted Langevin dynamics outlined in 2.1.1 to better explore the high-dimensional latent space Girolami & Calderhead (2011); Neal (2012); Liu & Liu (2001). Moreover, by using the unadjusted Langevin dynamics, it unlocks a cancellation of the forward and backward kernel

Algorithm 3 Guided open-horizon sequential Monte Carlo (GOHSMC)

Require: Training data $(\mathbf{x}_n, \mathbf{y}_n)_{n=1}^N$, number of samples J, initial parameters $\boldsymbol{\psi}_0$, prior distribution $q_0(\cdot)$, learning rate ϵ, and batch size M
Ensure: The MLE estimate $\boldsymbol{\psi}_t$
  Initialize samples $\{\boldsymbol{\theta}_0^{(j)}\}_{j=1}^J \sim q_0(\cdot)$
  Initialize weights according to equation 14
  for $t = 1, 2, \ldots$ until convergence **do**
    Draw sub dataset $\mathbf{y}_{\mathbf{S}_t^M} \subset \mathbf{y}_{1:N}$
    Calculate J_{eff} using equation 4
    if $J_{\text{eff}} < J/2$ **then**
      Resample $[\boldsymbol{\theta}_t^{(1)} ... \boldsymbol{\theta}_t^{(J)}]$ with probability $[\tilde{\mathbf{w}}_t^{(1)} ... \tilde{\mathbf{w}}_t^{(J)}]$
      Reset all weights to $\frac{1}{J}$
    end if
    for $j = 1$ **to** J **do**
      Propagate particles $\boldsymbol{\theta}_t^{(j)} \sim q_t^\theta(\cdot|\boldsymbol{\theta}_{t-1}^{(j)}; \boldsymbol{\psi}_{t-1})$
      Update weight $\mathbf{w}_t^{(j)}$ with equation 13
    end for
    Normalize weights using (3)
    $g(\boldsymbol{\psi}_{t-1}) = \frac{N}{M} \sum_{j=1}^J \tilde{\mathbf{w}}_t^{(j)} \nabla \log p(\mathbf{y}_{\mathbf{S}_t^M} | \boldsymbol{\theta}_t^{(j)}; \boldsymbol{\psi}_{t-1})$
    Update parameter $\boldsymbol{\psi}_t = \boldsymbol{\psi}_{t-1} + \epsilon g(\boldsymbol{\psi}_{t-1})$
  end for

evaluations in the weight update equation 6, thanks to the reversibility of the Langevin process Dai et al. (2022), so that the weight update is now tractable.

However, as our weight update is also conditional on the deterministic parameters, we have to alter it for the pBNN context

$$\mathbf{w}_t^{(j)} = \mathbf{w}_{t-1}^{(j)} \frac{\pi(\boldsymbol{\theta}_t^{(j)}|\boldsymbol{\psi}_{t-1})}{\pi(\boldsymbol{\theta}_{t-1}^{(j)}|\boldsymbol{\psi}_{t-2})} \frac{L_t^P(-\mathbf{P}^{(j)*})}{q_t^P(\mathbf{P}_{t-1}^{(j)})} \tag{13}$$

and

$$\mathbf{w}_0^{(j)} = \frac{\pi(\boldsymbol{\theta}_0^{(j)}|\boldsymbol{\psi}_0)}{q_0^\theta(\boldsymbol{\theta}_0^{(j)})} \tag{14}$$

where we note that there is no optimisation of the deterministic parameters, $\boldsymbol{\psi}_0$, upon initialisation.

4 Experiments

The following experiments were conducted using the JAX framework Bradbury et al. (2018) on an NVIDIA A100 GPU. The stochastic parameters were initialised from a standard normal distribution $\mathcal{N}(0, I)$ and the deterministic ones were initialised using JAX's standard technique, Xavier initialisation Glorot & Bengio (2010). To enhance generalisation, we did not apply any explicit regularization techniques such as dropout or weight decay, or any data augmentation techniques so as to focus on the core performance of the model.

As a baseline we also compared against two other common Bayesian approaches; Variational Inference (VI) Graves (2011) and Stochastic Gradient HMC (SGHMC) Chen et al. (2014). The full experimental set up and parameters are given in the appendix section B.

4.1 UCI Regression Datasets

The first experiments we have undertaken are on two common UCI regression datasets; The Red Wine Quality Dataset and the California Housing Dataset.

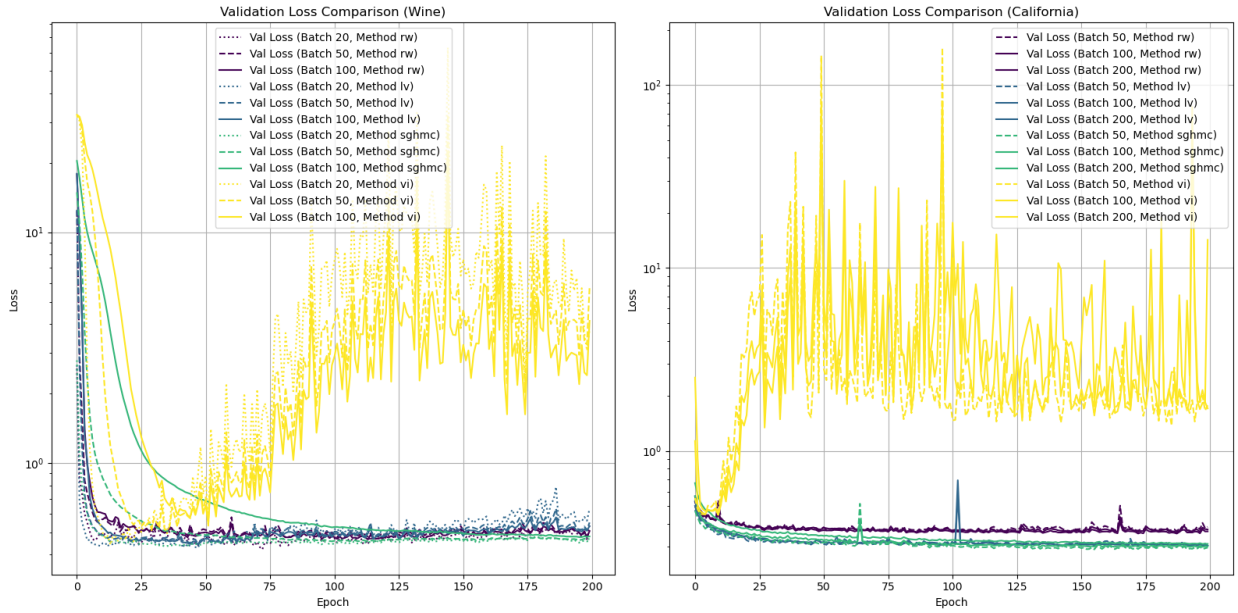

Figure 1: Validation loss comparison between different methods on different batch sizes for the Red Wine Quality and California Housing Dataset respectively, averaged over 5 runs.

The network architectures are simple feed forward networks with three dense layers, connected by a GeLU activation function Hendrycks & Gimpel (2023). The only difference between the two networks used in each experiment is the size of the first layer; the dimensionality of the first layers are 450 for CH and 240 for RWQ. The first layers were sampled by the SMC samplers while the rest of the layers were optimised by the Adam optimiser Kingma & Ba (2017) using a learning rate of 0.01. The RW scale and step size for the RW and Langevin kernels respectively were 0.01, while each method used 100 samples.

Each experiment was run for 200 epochs with the weights giving the best validation loss saved for use on the test dataset. We implemented a 60%, 30% and 10% train, validation and test split for both datasets and averaged the results over 5 runs. Three different batch sizes were chosen for each dataset; 20, 50, 100 for RWQ and 50, 100, 200 for CH. We gave larger batch sizes to the California runs due to its larger dataset. The results of this experiment can be seen in Table 1.

Table 1: Comparison of Test Loss on UCI Regression Datasets

Markov Kernel	**Red Wine Quality**			**California Housing**		
	Batch Size 20	**Batch Size 50**	**Batch Size 100**	**Batch Size 50**	**Batch Size 100**	**Batch Size 200**
Random Walk	0.4388 (0.0324)	0.4474 (0.0327)	0.4455 (0.0277)	0.3546 (0.0217)	0.3473 (0.0203)	0.3418 (0.0199)
Langevin	**0.4094 (0.093)**	**0.4096 (0.0146)**	**0.4083 (0.0144)**	0.2962 (0.0122)	0.3032 (0.0167)	**0.2989 (0.0156)**
SGHMC	0.4213 (0.181)	0.4211 (0.0177)	0.4401 (0.0312)	**0.2851 (0.0076)**	**0.2915 (0.0094)**	0.3047 (0.0206)
VI	0.4478 (0.130)	0.4433 (0.0147)	0.4461 (0.0245)	0.4332 (0.0126)	0.4252 (0.0142)	0.4173 (0.0170)

4.2 Classification

Model performance was evaluated on test set accuracy and loss. Training was conducted for 30 epochs and to ensure robustness, we performed 10 training runs with different random seeds for each kernel method, reporting the mean and standard deviation of the test accuracy and loss. For both datasets the weights were saved which corresponded with the best validation loss. These saved weights were then used at test time.

4.2.1 MNIST

For the first classification experiment, we evaluate the Markov Kernels on the MNIST dataset LeCun et al. (1998).

For the NN architecture we have chosen the LeNet-5 architecture Lecun et al. (1998). It consists of 2 convolutional layers followed by 2 dense layers and utilises the Tanh activation function. During training, the Adam optimiser Kingma & Ba (2015) with a learning rate of 0.002 was used for the deterministic parameters and we compare the results on three different batch sizes; 100, 500, 1000.

The first convolutional layer was chosen as the Bayesian layer which had a dimensionality of 160. Each SMC Sampler used 100 samples and the step size/RW variance scale was set to 0.01. The results from this experiment can be seen in Table 2.

Table 2: Comparison of Methods on MNIST with Different Batch Sizes

Markov Kernel	**MNIST**					
	Batch Size 100		**Batch Size 500**		**Batch Size 1000**	
	Test Loss (std)	**Test Accuracy (std)**	**Test Loss (std)**	**Test Accuracy (std)**	**Test Loss (std)**	**Test Accuracy (std)**
Random Walk	**0.0532 (0.0076)**	**98.43% (0.15)**	0.0422 (0.0024)	98.71% (0.09)	**0.0377 (0.0020)**	**98.81% (0.09)**
Langevin	0.0541 (0.0047)	98.42% (0.15)	**0.0412 (0.0039)**	**98.73% (0.10)**	0.0389 (0.0020)	98.79% (0.09)
SGHMC	0.0685 (0.0052)	97.88% (0.16)	0.0971 (0.0030)	96.99% (0.10)	0.1472 (0.0094)	95.63% (0.30)
VI	0.0780 (0.0046)	97.60% (0.13)	0.0530 (0.0038)	98.36% (0.07)	0.0511 (0.0034)	98.42% (0.06)

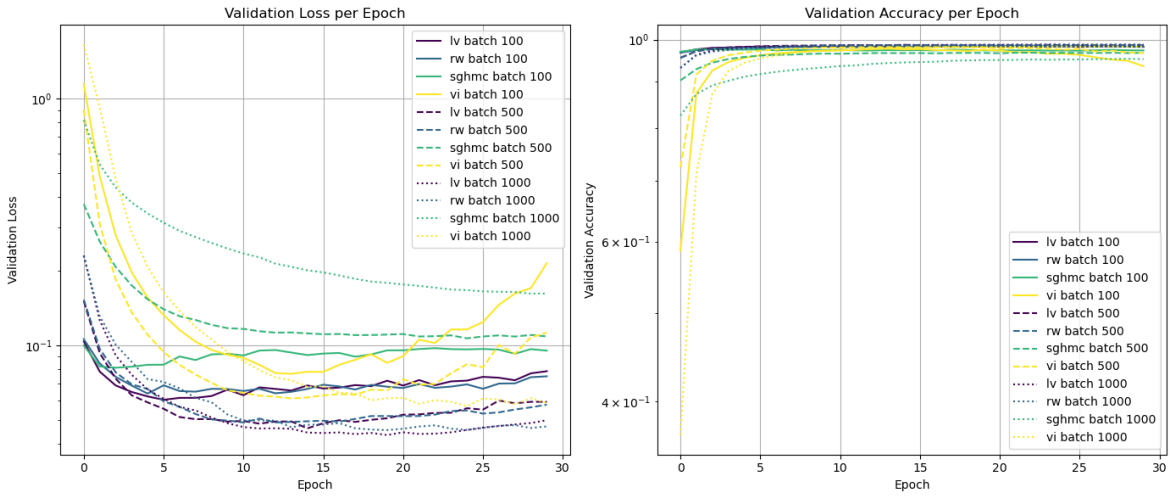

Figure 2: Validation loss and accuracy over the lifetime of each pBNN method and batch size for the MNIST Dataset, averaged over 10 runs.

4.2.2 FashionMNIST

For our second classification experiment, we evaluate the methods on another common image classification benchmark dataset in deep learning, the FashionMNIST dataset Xiao et al. (2017). This dataset consists of 70,000, 28 x 28, grayscale images depicted 10 different categories of fashion products. The training set has 60,000 data points and the test set has 10,000.

The NN architecture for this is a larger CNN than used in our MNIST experiment. It consists of a convolutional layer succeeded by a batch normalisation Ioffe & Szegedy (2015) layer and a max pooling Nagi et al. (2011) layer, this is then repeated with this second convolutional layer having a larger parameter size. This is then followed by 2 dense layers and the ReLU activation function Agarap (2019) was used.

The first convolutional layer was the chosen again as the Bayesian layer and had a dimensionality of 320. The SMC Samplers share the same set up as the first experiment except 50 samples were used and we tested it on the same 3 batch sizes; 100, 500 and 1000. The results from this experiment can be seen in Table 3.

Table 3: Comparison of Methods on FashionMNIST with Different Batch Sizes

Markov Kernel	**FashionMNIST**					
	Batch Size 100		**Batch Size 500**		**Batch Size 1000**	
	Test Loss (std)	**Test Accuracy (std)**	**Test Loss (std)**	**Test Accuracy (std)**	**Test Loss (std)**	**Test Accuracy (std)**
Random Walk	0.3121 (0.0089)	89.18% (0.47)	0.2913 (0.0088)	89.88 (0.0042)	0.2899 (0.0062)	90.11 (0.35)
Langevin	**0.2858** (0.0076)	**90.05% (0.38)**	**0.2794 (0.0076)**	**90.32% (0.23)**	**0.2758 (0.0074)**	**90.48% (0.43)**
SGHMC	0.3803 (0.0155)	86.71% (0.71)	0.4068 (0.0103)	85.83% (0.37)	0.4742 (0.0087)	83.48% (0.45)
VI	0.4322 (0.0239)	84.31% (1.08)	0.3798 (0.0179)	86.39% (0.84)	0.3581 (0.0154)	87.21% (0.62)

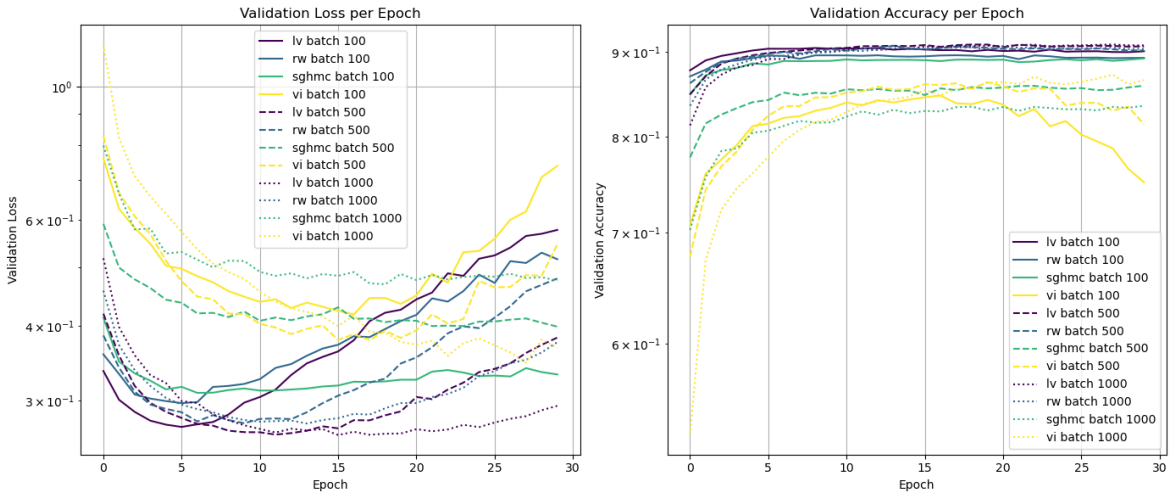

Figure 3: Validation loss and accuracy over the lifetime of each pBNN method and batch size for the FashionMNIST Dataset, averaged over 10 runs.

4.2.3 CIFAR10

For our final image classification experiment, we evaluate the methods on the CIFAR10 dataset Krizhevsky (2009). For this task we used the ResNet20 architecture He et al. (2015) with feature response Singh & Krishnan (2020), the same architecture used in Izmailov et al. (2021b). The network was trained for 200 epochs and we used a batch size of 100. We found that larger batch sizes led to sub optimal test metrics and therefore decided to compare only on the smaller batch size.

The first layer has a dimensionality of 448 parameters and we used a step size/random walk scale of 0.01. The rest of the parameters were trained using the AdamW optimiser Loshchilov & Hutter (2019) with a cosine annealing schedule where the initial value of the learning rate is set to of 0.01 and we used 10 samples for the SMC sampler. The results of these experiments can be found in table 4 and the validation loss plots can be found in figure 4.

Table 4: Comparison of Methods on CIFAR10 with Different Batch Sizes

Markov Kernel	**CIFAR10**	
	Test Loss (std)	**Test Accuracy (std)**
Random Walk	0.4024 (0.0204)	87.02% (0.86)
Langevin	**0.4007** (0.0120)	**87.27% (0.58)**

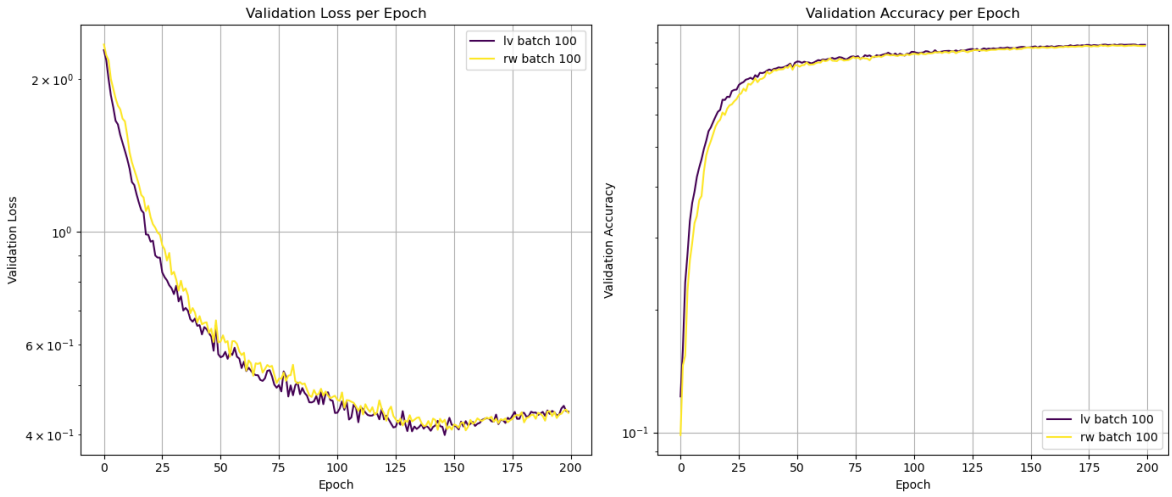

Figure 4: Validation loss and accuracy over the lifetime of each pBNN method and batch size for the CIFAR10 Dataset, averaged over 5 runs.

Table 5: OOD Performance Metrics

	Accuracy	F1-Score	Precision	Recall	Specificity	AUROC
SGHMC	0.8871 ± 0.0973	0.8667 ± 0.1238	0.9392 ± 0.0152	0.8244 ± 0.1945	**0.9497 ± 0.0025**	0.9682 ± 0.0333
RW	**0.9744 ± 0.0009**	**0.9751 ± 0.0009**	**0.9516 ± 0.0016**	**0.9997 ± 0.0005**	0.9491 ± 0.0017	**0.9994 ± 0.0006**
VI	0.9737 ± 0.0008	0.9743 ± 0.0008	0.9503 ± 0.0015	0.9996 ± 0.0005	0.9477 ± 0.0016	0.9978 ± 0.0016
LV	0.9704 ± 0.0055	0.9710 ± 0.0056	0.9495 ± 0.0022	0.9936 ± 0.0115	0.9471 ± 0.0026	0.9966 ± 0.0045

4.2.4 Out-of-distribution Analysis

We also tested all four methods on a benchmark out-of-distribution (OOD) problem. We trained the larger CNN outlined in the FashionMNIST experiment on the in distribution (ID) data (MNIST) data and then introduce the out of distribution (OOD) samples (FashionMNIST) at test time. The model is evaluated on its ability to distinguish between OOD samples ID samples (MNIST).

After training the network on MNIST, we use an energy-based Liu et al. (2021) detection method. We learn an energy threshold using a validation dataset of OOD samples. After this, we create a mixed dataset of ID and OOD samples and use the threshold learned at validation time to determine which data points are ID or OOD. The results of these experiments can be found in table 5 and the AUROC curves can be found in figure 5.

5 Discussion

Table 1 gives the test loss results for the regression datasets. We notice that Langevin consistently outperforms the RW proposal for each batch size and dataset, with the results being more pronounced on the housing dataset where the first layer dimensionality is larger than the wine dataset. The larger batch sizes seem to have minimal effect on the test loss (and in fact we see a slight boost in the test loss on the housing dataset) meaning that we can use fewer gradient calculations without effecting the performance of the model which also results in faster training times.

Table 2, 3 and 4 show the test loss and test accuracy for the MNIST, FashionMNIST and CIFAR10 datasets respectively. We can see that for both SMC methods, as the batch size increases the test accuracy increases and the test loss decreases. For the other two Bayesian methods, we see the opposite happens in that the test loss and accuracy decreases as the batch size increases. The RW Markov kernel marginally outperforms the Langevin Markov kernel in terms of test accuracy on the MNIST dataset, while Langevin performs better

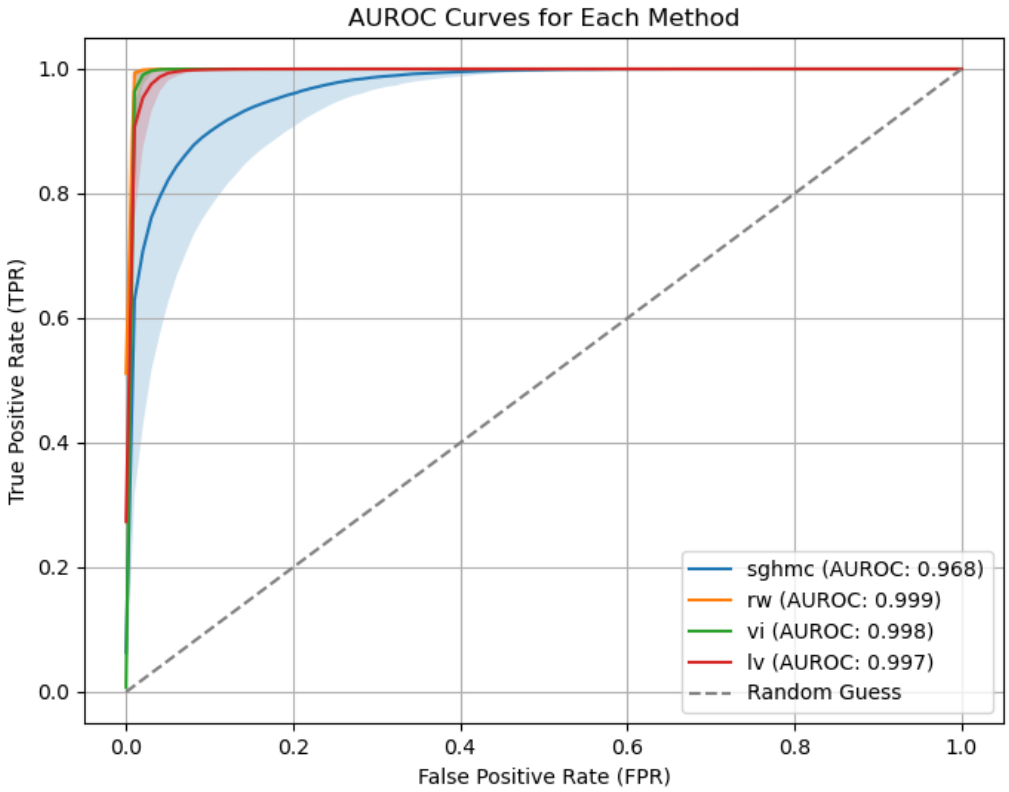

Figure 5: AUROC curves for each method averaged over 5 runs.

on the FashionMNIST and CIFAR10 datasets. The reason for the better performance for large batch sizes may be due to the fact that the larger batch gives a better approximation of the gradients over the full batch when scaled. The reason Langevin performs better on the larger neural networks and regression network could be due to the fact that the first layer sizes are larger than the MNIST architecture and Langevin is able to more efficiently navigate these higher dimensionality probability spaces.

On the OOD problem, we can see in table 5 that the VI, RW and Langevin have very similar results. This shows that despite the network not being fully bayesian, we see no drop off in performance in the unvertainty task.

We can see clearly in Figure 2 that on the FashionMNIST dataset, the validation loss starts to increase which may indicate an overfitting to the training data. This was the reason we saved the weights with the best validation loss as was done in Zhao et al. (2024). Another option would be to introduce an early stoppage routine LeCun et al. (2002) which is commonly employed in deep learning scenarios. The first option may be better for our scenario as often in stochastic settings, there may be a time period when the samples move to regions of higher loss before finding a better minimum, This can be seen clearly in validation loss plots in Figures 1 and 2. One other option could be to calculate the validation loss after each batch is processed and save the best weights from this, as was done in Zhao et al. (2024) however, this significantly increases the training time, especially for smaller batches. Larger batch sizes seemed to dampen the effect of overfitting to a certain extent, however if left for long enough without using one of the previously suggested methods, the effect may worsen to a similar extent as the smaller batch sizes.

Tables 6 and 7 also show the average run times in seconds with standard deviations for the regression and image classification datasets. As the batch size increases, the average training time decreases. We do see that

RW is faster than Langevin due to the fewer gradient calculations, but as discussed before, this can come at the cost of better test metric results. It is worth noting that Langevin uses the same number of gradient computations as a standard frequentist neural network. Therefore, if the architecture is set up optimally, it is possible to run pBNN's in a comparable run time to non-Bayesian neural networks. In appendix C we have provided results to show that when using a comparable runtime between RW and Langevin that the results are very similar to if they are run for the same amount of epochs.

Table 6: Comparison of Training Times on UCI Regression Datasets in seconds

Markov Kernel	**Red Wine Quality**			**California Housing**		
	Batch Size 20	**Batch Size 50**	**Batch Size 100**	**Batch Size 50**	**Batch Size 100**	**Batch Size 200**
Random Walk	245 (3)	105 (3)	54 (2)	1254 (7)	635 (6)	322 (3)
Langevin	**308 (5)**	**131 (4)**	**68 (3)**	**1616 (12)**	**817 (4)**	**416 (2)**

Table 7: Comparison of Training Times on Classification Datasets in seconds

Markov Kernel	**MNIST**			**Fashion MNIST**		
	Batch Size 100	**Batch Size 500**	**Batch Size 1000**	**Batch Size 100**	**Batch Size 500**	**Batch Size 1000**
Random Walk	1386 (11)	505 (6)	407 (7)	1204 (5)	516 (6)	475 (7)
Langevin	**1977 (15)**	**634 (8)**	**523 (9)**	**1745 (8)**	**751 (6)**	**718 (7)**

6 Conclusion

In this paper we introduce gradient based proposals into the SMC sampler architecture for use in the training of pBNNs. Specifically we have introduced Langevin dynamics as part of the Markov kernel and have demonstrated on four benchmark datasets, that we can outperform the current state-of-the-art SMC methods. This new proposal also allows us to use a larger batch size while gaining performance which in turn can reduce the training time of pBNNs. It is also worth noting that as the dimensionality of the first layer increased, the performance benefit of Langevin over RW was more pronounced. We also note that although only the first layer is stochastic, it performs comparatively on the uncertainty quantification tasks to full BNN techniques such as SGHMC and VI.

6.1 Limitations and Further Work

Memory and Runtime Limits Increased batch size comes at a memory cost. Further work on balancing the training time cost and memory cost would be very useful for future research. Our current codebase is not optimised to deal with larger Neural Networks such as ResNet He et al. (2015). Creating an optimised library with an automated memory and runtime balancing feature would be invaluable in future pBNN research. We have also not fully exploited the parallel nature of SMC samplers in our implementation. Using a parallel resampling scheme Varsi et al. (2021a) could potentially reduce runtime.

Other Gradient Based Proposals So far, we have only introduced Langevin dynamics as a gradient based proposal. However, there are other gradient based proposals that are also worth exploring such as Hamiltonian Monte Carlo (HMC) Neal (2011). One potential problem with HMC is the need to tune both the step size and the number of leapfrog steps. Two different approaches to solving this could be taken; first, the No U-Turn algorithm could be used as a Markov kernel Devlin et al. (2024). This algorithm has a U-turn termination criteria embedded into the algorithm which automatically tunes the number of leapfrog steps the algorithm should run for. Second, an adaptive HMC Markov kernel could be used instead Buchholz et al. (2020). A study to compare different gradient based Markov kernel methods would be interesting and useful. All of these methods would involve using multiple gradient evaluations due to the many step leapfrog process so there would be a greater computational overhead than the methods introduced in this paper. It would then be interesting to investigate whether the increased computational cost is offset by the increased performance of the more sophisticated approach.

References

Abien Fred Agarap. Deep learning using rectified linear units (relu), 2019.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

Alexander Buchholz, Nicolas Chopin, and Pierre E. Jacob. Adaptive tuning of hamiltonian monte carlo within sequential monte carlo, 2020.

Tianqi Chen, Emily Fox, and Carlos Guestrin. Stochastic gradient hamiltonian monte carlo. In *International conference on machine learning*, pp. 1683–1691. PMLR, 2014.

Chenguang Dai, Jeremy Heng, Pierre E. Jacob, and Nick Whiteley. An invitation to sequential Monte Carlo samplers. *Journal of the American Statistical Association*, 117(539):1587–1600, 2022.

Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential monte carlo samplers. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 68(3):411–436, 2006.

Lee Devlin, Matthew Carter, Paul Horridge, Peter L. Green, and Simon Maskell. The no-u-turn sampler as a proposal distribution in a sequential monte carlo sampler without accept/reject. *IEEE Signal Processing Letters*, 31:1089–1093, 2024. doi: 10.1109/LSP.2024.3386494.

Randal Douc, Olivier Cappé, and Eric Moulines. Comparison of resampling schemes for particle filtering, 2005. URL `https://arxiv.org/abs/cs/0507025`.

Arnaud Doucet, Nando de Freitas, and Neil J. Gordon. An introduction to sequential monte carlo methods. In Arnaud Doucet, Nando de Freitas, and Neil J. Gordon (eds.), *Sequential Monte Carlo Methods in Practice*, pp. 3–14. Springer, New York, NY, 2001.

Yarin Gal, Riashat Islam, and Zoubin Ghahramani. Deep bayesian active learning with image data, 2017.

Mark Girolami and Ben Calderhead. Riemann manifold langevin and hamiltonian monte carlo methods. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 73(2):123–214, 2011.

Geof H Givens and Adrian E Raftery. Local adaptive importance sampling for multivariate densities with strong nonlinear relationships. *Journal of the American Statistical Association*, 91(433):132–141, 1996.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256. JMLR Workshop and Conference Proceedings, 2010.

Alex Graves. Practical variational inference for neural networks. *Advances in neural information processing systems*, 24, 2011.

W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57 (1):97–109, 1970.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2023. URL `https://arxiv.org/abs/1606.08415`.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

Pavel Izmailov, Sharad Vikram, Matthew D. Hoffman, and Andrew Gordon Wilson. What are bayesian neural network posteriors really like? *CoRR*, abs/2104.14421, 2021a. URL `https://arxiv.org/abs/2104.14421`.

Pavel Izmailov, Sharad Vikram, Matthew D. Hoffman, and Andrew Gordon Wilson. What are bayesian neural network posteriors really like?, 2021b. URL https://arxiv.org/abs/2104.14421.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015. URL https://arxiv.org/abs/1412.6980.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

Genshiro Kitagawa. Monte carlo filter and smoother for non-gaussian nonlinear state space models. *Journal of Computational and Graphical Statistics*, 5(1):1–25, 1996. ISSN 10618600. URL http://www.jstor.org/stable/1390750.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical Report TR-2009, University of Toronto, Toronto, Canada, 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

Yann LeCun, Corinna Cortes, and CJ Burges. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist*, 1998.

Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pp. 9–50. Springer, 2002.

Jun S. Liu and Rong Chen. Sequential monte carlo methods for dynamic systems. *Journal of the American Statistical Association*, 93(443):1032–1044, 1998. ISSN 01621459, 1537274X. URL http://www.jstor.org/stable/2669847.

Jun S Liu and Jun S Liu. *Monte Carlo strategies in scientific computing*, volume 10. Springer, 2001.

Weitang Liu, Xiaoyun Wang, John D. Owens, and Yixuan Li. Energy-based out-of-distribution detection, 2021. URL https://arxiv.org/abs/2010.03759.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, 1953. doi: 10.1063/1.1699114.

Jawad Nagi, Frederick Ducatelle, Gianni A. Di Caro, Dan Cireşan, Ueli Meier, Alessandro Giusti, Farrukh Nagi, Jürgen Schmidhuber, and Luca Maria Gambardella. Max-pooling convolutional neural networks for vision-based hand gesture recognition. In *2011 IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*, pp. 342–347, 2011. doi: 10.1109/ICSIPA.2011.6144164.

Radford M. Neal. *MCMC using Hamiltonian dynamics*, pp. 113–162. Chapman and Hall/CRC, 2011. doi: 10.1201/b10905.

Radford M Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

G. O. Roberts and R. L. Tweedie. Exponential convergence of langevin distributions and their discrete approximations. *Bernoulli*, 2(4):341–363, 1996.

Conor Rosato, Joshua Murphy, Alessandro Varsi, Paul Horridge, and Simon Maskell. Enhanced smc^2: Leveraging gradient information from differentiable particle filters within langevin proposals, 2024. URL https://arxiv.org/abs/2407.17296.

Mrinank Sharma, Sebastian Farquhar, Eric Nalisnick, and Tom Rainforth. Do bayesian neural networks need to be fully stochastic?, 2023.

Saurabh Singh and Shankar Krishnan. Filter response normalization layer: Eliminating batch dependence in the training of deep neural networks, 2020. URL https://arxiv.org/abs/1911.09737.

Alessandro Varsi, Simon Maskell, and Paul G. Spirakis. An o(log2n) fully-balanced resampling algorithm for particle filters on distributed memory architectures. *Algorithms*, 14(12), 2021a. ISSN 1999-4893. doi: 10.3390/a14120342. URL https://www.mdpi.com/1999-4893/14/12/342.

Alessandro Varsi, Simon Maskell, and Paul G Spirakis. An o (log2 n) fully-balanced resampling algorithm for particle filters on distributed memory architectures. *Algorithms*, 14(12):342, 2021b.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

Zheng Zhao, Sebastian Mair, Thomas B. Schön, and Jens Sjölund. On feynman–kac training of partial bayesian neural networks, 2024.

A Langevin L-kernel derivation for section 3

The following work gives the derivation for the L-kernel where we utilise the reverse momentum in the weight update.

A.1 Proposal

Langevin dynamics for a single time step can be described via the following equation

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} + \frac{\epsilon^2}{2}\nabla \log \pi(\boldsymbol{\theta}_{t-1}) + \epsilon \mathbf{P}_{t-1} \tag{15}$$

This process is equivalent to the leapfrog integrator for a single timestep

$$\mathbf{P}_t = \mathbf{P}_{t-1} + \frac{\epsilon}{2}\nabla \log \pi(\boldsymbol{\theta}_{t-1}) \tag{16}$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} + \epsilon \mathbf{P}^* \tag{17}$$

And the proposal at the corresponding timestep can be written as $q_t(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1})$. We would like the proposal to reflect the stochastic momentum introduced and the Langevin dynamics. Therefore we must introduce a change of variables from $\mathbf{P}_{t-1}$ to $\boldsymbol{\theta}_t$. The change of variables can be described by

$$Y = g(X) \tag{18}$$

$$p_Y(y) = p_X(x)\left|\frac{dg(X)}{dX}\right|^{-1} \tag{19}$$

In our case $Y = \boldsymbol{\theta}_t$, $X = \mathbf{P}_{t-1}$ and $g(X) = f_{LMC}(\boldsymbol{\theta}_{t-1}, \mathbf{P}_{t-1})$, therefore our proposal can be rewritten from a proposal on $\boldsymbol{\theta}$ to a proposal on $\mathbf{P}$:

$$\begin{aligned} q_t^\theta(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}) &= q_t^\theta(f_{LMC}(\boldsymbol{\theta}_{t-1}, \mathbf{P}_{t-1})|\boldsymbol{\theta}_{t-1}) \\ &= q_t^P(\mathbf{P}_{t-1})\left|\tfrac{df_{LMC}(\boldsymbol{\theta}_{t-1}, \mathbf{P}_{t-1})}{d\mathbf{P}_{t-1}}\right|^{-1} \end{aligned} \tag{20}$$

Where f_{LMC} is the Langevin dynamics used to propagate our samples and $\left|\frac{df_{LMC}(\boldsymbol{\theta}_{t-1}, \mathbf{P}_{t-1})}{d\mathbf{P}_{t-1}}\right|^{-1}$ is the determinant of the transformation from $\boldsymbol{\theta}_{t-1} \rightarrow \boldsymbol{\theta}_t$ given our momentum $\mathbf{P}_{t-1}$. The initial momentum is usually sampled from a normal distribution

$$\mathbf{P}_{t-1} \sim \mathcal{N}(0, \boldsymbol{M}) \tag{21}$$

$\boldsymbol{M}$ is known as the mass matrix which governs the covariance of the distribution from which we pull our momentum. In our implementation we have set this mass matrix to be the identity matrix. Therefore the proposal can be given as

$$q_t^\theta(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}) = \mathcal{N}(\mathbf{P}_{t-1}; 0, \boldsymbol{M})\left|\tfrac{df_{LMC}(\boldsymbol{\theta}_{t-1}, \mathbf{P}_{t-1})}{d\mathbf{P}_{t-1}}\right|^{-1} \tag{22}$$

A.2 L-kernels

Langevin is a reversible process, therefore the momentum which would take us from $\boldsymbol{\theta}_t \rightarrow \boldsymbol{\theta}_{t-1}$ is the opposite of the one that takes us originally from $\boldsymbol{\theta}_{t-1} \rightarrow \boldsymbol{\theta}_t$ after the momentum update given by equation 16. Therefore, we can rewrite the L-kernel $L_t^\theta(\boldsymbol{\theta}_{t-1}|\boldsymbol{\theta}_t)$ using the same change of variables process used in the proposal (given by equation 19), but using the reverse momentum $\mathbf{P}_t$.

$$\begin{aligned} L_t^\theta(\boldsymbol{\theta}_{t-1}|\boldsymbol{\theta}_t) &= L_t^\theta(f_{LMC}(\boldsymbol{\theta}_t, -\mathbf{P}^*)|\boldsymbol{\theta}_t) \\ &= L_t^P(-\mathbf{P}^*) \left| \tfrac{df_{LMC}(\boldsymbol{\theta}_t, -\mathbf{P}^*)}{d\mathbf{P}^*} \right|^{-1} \\ &= \mathcal{N}(-\mathbf{P}^*; 0, \boldsymbol{M}) \left| \tfrac{df_{LMC}(\boldsymbol{\theta}_t, -\mathbf{P}^*)}{d\mathbf{P}^*} \right|^{-1} \end{aligned} \tag{23}$$

Due to the reversible nature of the Langevin dynamics, the determinants in both the proposal 22 and L-kernel 23 are equivalent, and therefore cancel in the final weight update.

A.3 pBNN context

The gradient of the Langevin dynamics is conditioned on the deterministic parameters which are the same for both the forwards and backwards moves.

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} + \frac{\epsilon^2}{2} \nabla \log \pi(\boldsymbol{\theta}_{t-1}|\boldsymbol{\psi}_{t-1}) + \epsilon \mathbf{P}_{t-1} \tag{24}$$

Therefore we can simply define our proposal and L-kernel in the context of a pBNN respectively as

$$q_t^\theta(f_{LMC}(\boldsymbol{\theta}_{t-1}, \mathbf{P}_{t-1})|\boldsymbol{\theta}_{t-1}, \boldsymbol{\psi}_{t-1}) = q_t^P(\mathbf{P}_{t-1}|\boldsymbol{\psi}_{t-1}) \left| \tfrac{df_{LMC}(\boldsymbol{\theta}_{t-1}, \mathbf{P}_{t-1})}{d\mathbf{P}_{t-1}} \right|^{-1} \tag{25}$$

However, the momentum is drawn from a Gaussian distribution independent of both $\boldsymbol{\theta}$ and $\boldsymbol{\psi}$. Therefore

$$\begin{aligned} q_t^\theta(f_{LMC}(\boldsymbol{\theta}_{t-1}, \mathbf{P}_{t-1})|\boldsymbol{\theta}_{t-1}, \boldsymbol{\psi}_{t-1}) &= q_t^P(\mathbf{P}_{t-1}) \left| \tfrac{df_{LMC}(\boldsymbol{\theta}_{t-1}, \mathbf{P}_{t-1})}{d\mathbf{P}_{t-1}} \right|^{-1} \\ &= \mathcal{N}(\mathbf{P}_{t-1}; 0, \boldsymbol{M}) \left| \tfrac{df_{LMC}(\boldsymbol{\theta}_{t-1}, \mathbf{P}_{t-1})}{d\mathbf{P}_{t-1}} \right|^{-1} \end{aligned} \tag{26}$$

$$L_t^\theta(f_{LMC}(\boldsymbol{\theta}_t, -\mathbf{P}^*)|\boldsymbol{\theta}_t, \boldsymbol{\psi}_{t-1}) = \mathcal{N}(-\mathbf{P}^*; 0, \boldsymbol{M}) \left| \tfrac{df_{LMC}(\boldsymbol{\theta}_t, -\mathbf{P}^*)}{d\mathbf{P}^*} \right|^{-1} \tag{27}$$

B Experiment Parameters and Set Up

For all of the SMC experiments, a step size/RW scale of 0.01 was used. However, in order to achieve optimal results on some of the experiments, the step size had to be altered for certain batches sizes when using SGHMC and VI. These hyperparameters are outlined in tables 8 and 9.

Experiment	Small Batch Size	Medium Batch Size	Large Batch Size
California Housing	0.001	0.001	0.001
Wine	0.01	0.01	0.01
MNIST	0.001	0.001	0.0005
FashionMNIST	0.001	0.001	0.0002
OOD	0.001	0.001	0.001

Table 8: Learning Rates for SGHMC Experiments

Experiment	Small Batch Size	Medium Batch Size	Large Batch Size
California Housing	0.001	0.001	0.001
Wine	0.001	0.001	0.001
MNIST	0.001	0.001	0.0002
FashionMNIST	0.001	0.001	0.0002
OOD	0.001	0.001	0.001

Table 9: Learning Rates for VI Experiments

C Comparable Runtime Comparison

We decided also to run the MNIST and FashionMNIST image classification experiments as before but this time compare the different proposals using a comparable runtime. In order to do this we let the RW proposal run again for 30 epochs and then adjusted the Langevin method so it ran for a similar amount of time as the random walk proposal, resulting in it training for fewer epochs. The results can be found in tables 10 and 11 and the validation loss and accuracy curves for this runtime comparison can be found in figures 6 and 7.

Table 10: Comparison of Methods on MNIST with Different Batch Sizes

Method	Batch Size 100			Batch Size 500			Batch Size 1000		
	Test Loss (std)	Test Acc (std)	Runtime (std)	Test Loss (std)	Test Acc (std)	Runtime (std)	Test Loss (std)	Test Acc (std)	Runtime (std)
Random Walk	**0.0517 (0.0046)**	**98.45% (0.09)**	1005.86s (9.70)	**0.0422 (0.0024)**	98.71% (0.09)	442.87s (10.60)	**0.0377 (0.0020)**	**98.81% (0.09)**	384.19s (10.98)
Langevin	0.0522 (0.0042)	98.42% (0.18)	**1065.61s (14.78)**	0.0426 (0.0025)	**98.72% (0.09)**	**392.05s (13.23)**	0.0378 (0.0037)	98.80% (0.12)	**340.09s (13.20)**

Table 11: Comparison of Methods on FashionMNIST with Different Batch Sizes

Method	Batch Size 100			Batch Size 500			Batch Size 1000		
	Test Loss (std)	Test Acc (std)	Runtime (std)	Test Loss (std)	Test Acc (std)	Runtime (std)	Test Loss (std)	Test Acc (std)	Runtime (std)
Random Walk	0.3157 (0.0107)	88.96% (0.48)	890.66s (48.64)	0.2922 (0.0068)	89.76% (0.46)	425.65s (9.43)	0.2899 (0.0062)	90.11% (0.35)	393.46s (10.84)
Langevin	**0.2815 (0.0078)**	**90.17% (0.29)**	**932.94s (10.52)**	**0.2789 (0.0038)**	**90.30% (0.18)**	**451.03s (9.89)**	**0.2763 (0.0066)**	**90.38% (0.30)**	**422.78s (10.73)**

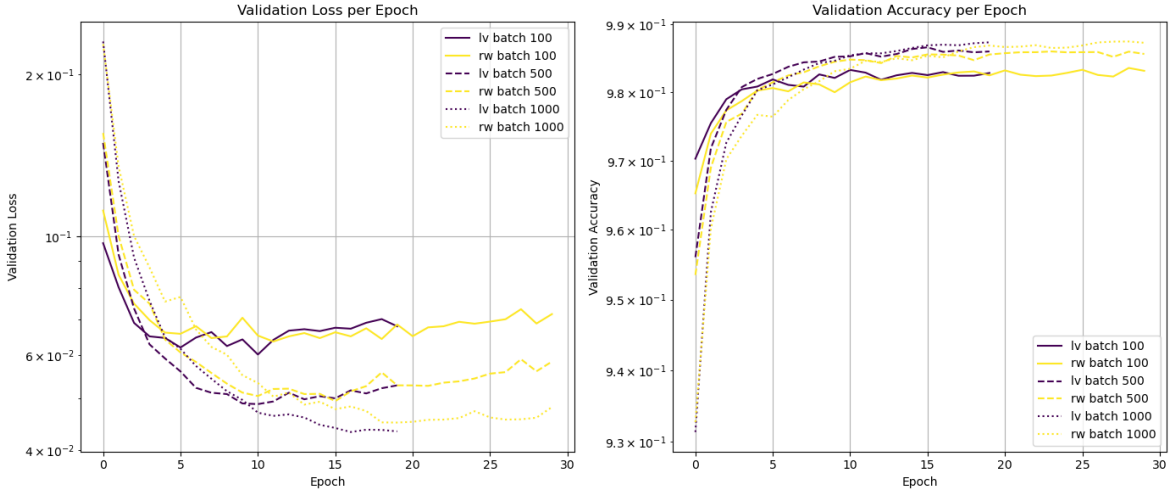

Figure 6: Validation loss comparison between different methods for a fixed runtime on the MNIST dataset.

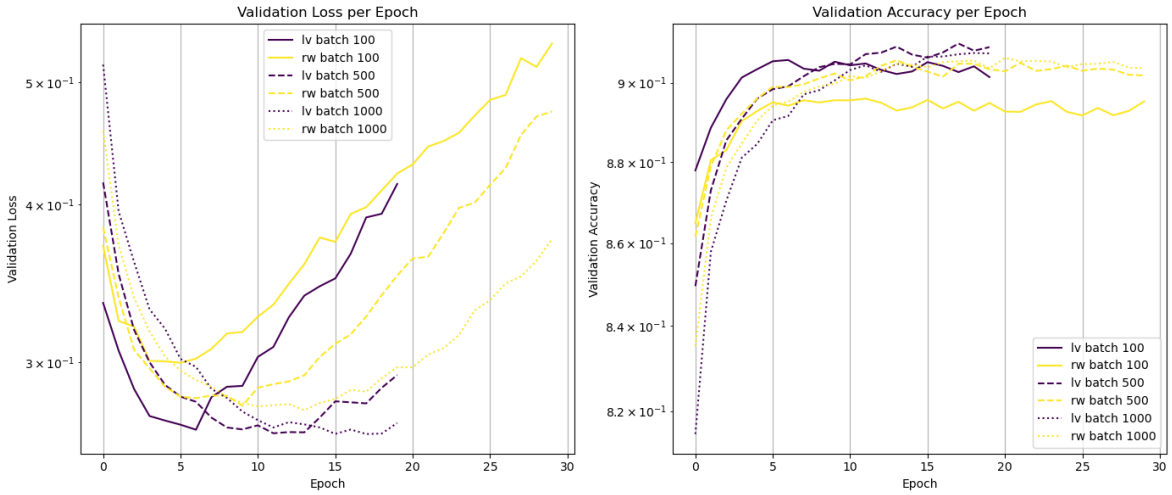

Figure 7: Validation loss comparison between different methods for a fixed runtime on the FashionMNIST dataset.