Learning Hyperparameters for Neural Network Models Using Hamiltonian Dynamics

by

Kiam Choo

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Learning Hyperparameters for Neural Network Models Using Hamiltonian Dynamics

Kiam Choo

Master of Science

Graduate Department of Computer Science

University of Toronto

2000

We consider a feedforward neural network model with hyperparameters controlling groups of weights. Given some training data, the posterior distribution of the weights and the hyperparameters can be obtained by alternately updating the weights with hybrid Monte Carlo and sampling from the hyperparameters using Gibbs sampling. However, this method becomes slow for networks with large hidden layers. We address this problem by incorporating the hyperparameters into the hybrid Monte Carlo update. However, the region of state space under the posterior with large hyperparameters is huge and has low probability density, while the region with small hyperparameters is very small and very high density. As hybrid Monte Carlo inherently does not move well between such regions, we reparameterize the weights to make the two regions more compatible, only to be hampered by the resulting inability to compute good stepsizes. No definite improvement results from our efforts, but we diagnose the reasons for that, and suggest future directions of research.

Dedication

I dedicate this thesis to my family, who have accepted my wanderings over the years. I especially dedicate this to my mother, whose strength and will I carry on.

Acknowledgements

I thank Prof. Radford Neal for his invaluable guidance on this thesis. I also thank Faisal Qureshi for his helpful suggestions and friendship during this thesis. Thanks also to the other inhabitants of the Artificial Intelligence Laboratory who have helped me in some way to complete this thesis.

Contents

Chapter 1

Introduction

1.1 Overview

A feedforward neural network is a nonlinear model that maps an input to an output. It can be viewed as a nonparametric model in the sense that its parameters cannot easily be interpreted to provide insight into the problem that it is being used for. Nevertheless, feedforward neural networks are powerful as with sufficient hidden units they can learn to approximate any nonlinear mapping arbitrarily closely (Cybenko, 1989). Partly because of this flexibility, they have become widespread tools used by many practitioners in the sciences and engineering. These practitioners typically use well-established learning techniques like backpropagation (Rumelhart *et al.*, 1986) or its variants. But despite the multitude of learning methods already in existence, learning for feedforward networks remains an area of active research.

A recent approach to feedforward neural net learning is Bayesian learning (Buntine and Weigend, 1991; MacKay, 1991, 1999; Neal, 1996; Müller and Insua, 1998). This new approach can be viewed as a response to the problem of incorporating prior knowledge into neural networks. However, the computational problems in Bayesian learning are complex, and none of the existing techniques are perfect. In the interests of computational tractability, both the works of MacKay (1991; 1999) and Buntine and Weigend (1991) assume Gaussian approximations to the posterior distribution over network weights.

A more general and flexible approach is to sample from the posterior distribution of the weights, as has been done by Neal (1996) and Müller and Insua (1998). Neal obtains samples by alternating hybrid Monte Carlo updates of the weights with Gibbs sampling updates of the hyperparameters. Müller and Insua also alternately update the weights and Gibbs-sample the hyperparameters, but in addition, they observe that, given all weights except for the hidden-to-output ones, the posterior distribution of the latter is simply Gaussian when the data noise is Gaussian. While the other weights still need to be updated by a more complicated Metropolis step, this does allow them to sample directly from the Gaussian distribution of the hidden-to-output weights. However, as will be described later, both methods are expected to become slow for large networks, possibly to the point where they become unusable.

This thesis addresses the above inefficiency for large networks. Specifically, it is concerned with improving on the hybrid Monte Carlo technique used by Neal so that both parameters and hyperparameters are updated using hybrid Monte Carlo.

1.2 The Neural Network Learning Problem

The rest of this thesis is about feedforward neural networks only, so we drop the "feedforward" for simplicity.

In this section, we define the neural network learning problem that underlies this thesis.

Given a set of inputs $X = \{\mathbf{x}^c\}_{c=1}^{N_c}$ and targets $Y = \{\mathbf{y}^c\}_{c=1}^{N_c}$, a neural network can be used to model the relationship between them so that:

$$\mathbf{f}(\mathbf{x}^c; W) \approx \mathbf{y}^c \tag{1.1}$$

where $\mathbf{f}(\cdot; W)$ is the function computed by the neural network with weights W. This modeling is achieved by "training" the weights W using the training data consisting of the inputs X and the targets Y. Once training is complete, the neural net can be used to predict targets given previously unseen values of inputs.

Conventionally, the learning process is viewed as an optimization problem where the weights are learned using some kind of gradient descent method on an error function such as the following:

$$E(W) = \sum_{c=1}^{N_c} \left(\mathbf{f}(\mathbf{x}^c; W) - \mathbf{y}^c\right)^2 \tag{1.2}$$

The result of this procedure is a single optimal set of weights W_{opt} that minimizes the error. This single set of weights is then used for future predictions from a new input. The conventionally-trained network prediction is thus:

$$\mathbf{f}_C(\mathbf{x}) = \mathbf{f}(\mathbf{x}; W_{opt}) \tag{1.3}$$

1.3 Bayesian Approach to Neural Net Learning

The Bayesian approach to neural network learning differs fundamentally from the conventional optimization approach in that, rather than obtaining a single "best" set of weights from the training process, a probability distribution over the weights is obtained instead.

1.3.1 Bayesian Inference

Generally speaking, Bayesian inference is a way by which unknown properties of a system may be inferred from observations. In the Bayesian inference framework, we model the observations z as being generated by some model with unobserved parameters ζ. Specifically, we come up with a likelihood function $p(z|\zeta)$, the probability of the observable state z given a particular setting for the parameter ζ. Next, we decide on $p(\zeta)$, the prior probability distribution over parameters ζ.

With these two functions in hand, we use Bayes' rule to infer the posterior probability that the parameters have the value ζ when we observe the state z:

$$p(\zeta|z) = \frac{p(\zeta)p(z|\zeta)}{p(z)} \tag{1.4}$$

Bayesian learning can be applied to neural networks in the following way. We model the targets as the neural network output $\mathbf{f}(\mathbf{x}; W)$ plus some noise, which defines the likelihood $p(Y|W, X)$, and we assume some form for the prior distribution of the weights $p(W)$. The posterior distribution of the weights is then:

$$p(W|X, Y) = \frac{p(W)p(Y|W, X)}{p(Y|X)} \tag{1.5}$$

where we have set $p(W|X) = P(W)$ since the prior distribution of the weights does not depend on the inputs. $p(W|X, Y)$ in Eqn. 1.5 is the probability distribution over weights that we infer in the Bayesian framework.

1.3.2 A Simple Example

As a simple example, assuming that the noise in the output of each unit is Gaussian with fixed standard deviation σ, we get for a net with N_y outputs:

$$p(Y|W, X) = \left(\frac{1}{\sqrt{2\pi}\sigma}\right)^{N_c N_y} \exp\left[-\frac{1}{2\sigma^2}\sum_{c=1}^{N_c}\left|\mathbf{f}(\mathbf{x}^c; W) - \mathbf{y}^c\right|^2\right] \tag{1.6}$$

And assuming a simple prior where all the weights $W = \{w_i\}_{i=1}^{N_w}$ have Gaussian distribution of fixed inverse variance τ:

$$p(W) = \left(\sqrt{\frac{\tau}{2\pi}}\right)^{N_w} \exp\left[-\frac{\tau}{2}\sum_{i=1}^{N_w} w_i^2\right] \tag{1.7}$$

This gives the posterior distribution for W:

$$\begin{aligned} p(W|X, Y) &\propto p(W)p(Y|W, X) \\ &\propto \exp\left[-\frac{\tau}{2}\sum_{i=1}^{N_w} w_i^2 - \frac{1}{2\sigma^2}\sum_{c=1}^{N_c}\left|\mathbf{f}(\mathbf{x}^c; W) - \mathbf{y}^c\right|^2\right] \end{aligned} \tag{1.8}$$

Here, the symbol $\propto$ denotes proportionality. We have dropped the normalizing constant $1/p(Y|X)$ as well as factors not dependent on the weights. This is because we are considering the posterior distribution over the weights only in this simple model, with σ and τ fixed.

1.3.3 Making Predictions

The prediction of the net trained using Bayesian inference is obtained as the expected output over all possible weight settings, weighted by their posterior probabilities:

$$\mathbf{f}_B(\mathbf{x}) = E[\mathbf{f}(\mathbf{x}; W)]_{W|X,Y} = \int \mathbf{f}(\mathbf{x}; W) p(W|X, Y) dW \tag{1.9}$$

Compared to Eqn. 1.3, Bayesian prediction is clearly more complicated.

1.3.4 Determining the Hyperparameters From the Data

The inverse variance of the weights τ is called a hyperparameter because it is a parameter that controls the prior distribution of the parameters w_i. In practice, it is reasonable to let the hyperparameters be determined from the data. For instance, the input-to-hidden weights for one training set might need to be larger than for another training set because its outputs vary more rapidly. Evidently, it is possible to infer the hyperparameters from the training data.

But we infer the hyperparameters not just because it is possible, but because it is desirable as well. This is because it is difficult for a human operator to guess a good setting of the hyperparameters, but it is easier to guess a prior distribution for hyperparameters, e.g., in terms of its mean and some measure of its spread. Moreover, allowing the precision τ to vary in Eqn. 1.7 couples the weights in their prior distribution and allows for a richer prior, whereas all the weights would be independent in their prior if τ were fixed. Details of the incorporation of the hyperparameters into the sampling procedure will be given in later chapters.

Once we have the posterior distribution of the weights and the hyperparameters, the network has been "trained". Predictions from the net now involve the joint posterior distribution of the weights and the hyperparameters Ψ:

$$\mathbf{f}_B(\mathbf{x}) = E[\mathbf{f}(\mathbf{x}; W, \Psi)]_{W,\Psi|X,Y} = \int \mathbf{f}(\mathbf{x}; W) p(W, \Psi|X, Y) dW d\Psi \tag{1.10}$$

The above integral usually cannot be analytically obtained for neural network models.

Note that σ, the variance of the data, is often also regarded as a hyperparameter because the role it plays in controlling the network error is similar to that played by other hyperparameters in controlling their respective weights.

1.4 Motivation

The practicality of the Bayesian framework hinges on the existence of computationally efficient ways to evaluate or approximate Eqn. 1.10. The main problem is that the posterior distribution $p(W, \Psi|X, Y)$ is often such that the integral in Eqn. 1.10 cannot be performed analytically.

In the interests of computational feasibility, Buntine and Weigend (1991) and MacKay (1991; 1999) approximate the posterior distribution of the weights and hyperparameters as a Gaussian distribution. Unfortunately, it usually cannot be seen in advance from the training data if a Gaussian distribution is a reasonable approximation to the posterior distribution. For instance, for a small network that has just enough hidden units to model some given data, we would expect that, ignoring multiple modes due to units swapping roles, the posterior distribution is peaked at a single mode because each unit has a well-constrained role to play in the mapping. In such a case, one might reasonably expect the posterior to be approximately Gaussian. However, when there are more hidden units, units are no longer so constrained, and the posterior distribution will be broader in ways that do not necessarily retain a Gaussian appearance.

Neal's (1996) and Müller and Insua's (1998) approach to the problem is to sample from the posterior distribution using Markov chain Monte Carlo (MCMC) techniques. MCMC techniques do not approximate the posterior distribution as a Gaussian, but instead sample faithfully from its true form. With n samples from the posterior distribution, we can obtain the expected output of the net as:

$$\mathbf{f}_B(\mathbf{x}) \approx \frac{1}{n}\sum_{i=1}^{n}\mathbf{f}(\mathbf{x}; W_i, \Psi_i) \tag{1.11}$$

In order for this method to be effective, each sample (W_i, Ψ_i) must be as independent of the previous sample as possible. A common problem of MCMC techniques is that samples can be highly correlated, in which case even though they are drawn from the correct distribution, they sample the distribution very slowly, and a huge number of samples might be needed for reliable estimates. In severe cases, the method becomes infeasible for practical use. The MCMC technique used by Neal faces this problem when the number of hidden units becomes large. His method alternates between using hybrid Monte Carlo to update the network parameters, and Gibbs sampling to sample the hyperparameters. Unfortunately, it is this alternation between updating the parameters and the hyperparameters that causes high correlations from one sample to the next as the number of hidden units becomes large.

The root of this inefficiency is that, during each hybrid Monte Carlo process that yields one sample of the weights, the hyperparameters are held fixed. This would not be a problem if, to obtain the next sample of the weights, the hyperparameters can be shifted to an uncorrelated value. However, because the hyperparameters are updated using Gibbs sampling given the current values of the weights, they are "pinned" and unable to move much. The larger the number of hidden units, the greater the pinning effect is.

Müller and Insua's method suffers from the same inefficiency as it also alternates Gibbs sampling of the hyperparameters with Markov chain updates of the network parameters.

This problem is the motivation for this thesis. In this thesis, we propose and investigate a modification to the hybrid Monte Carlo technique used by Neal. Specifically, rather than updating the weights by hybrid Monte Carlo and the hyperparameters by Gibbs sampling, we update both the weights and the hyperparameters using hybrid Monte Carlo. The idea is that, because both the weights and the hyperparameters are now changing at the same time, we no longer have the pinning effect, and hybrid Monte Carlo should then be able to produce samples that move much more efficiently through the posterior distribution.

Of course one might ask why we would want to use large hidden layers in the first place. There are several reasons for this. Firstly, since a neural network is a nonparametric model, it makes sense when modelling some data to use a lot of hidden units in order to maximize the network's power of representation. That is, we want the function computed by the neural network to not be constrained by there being too few units, and be determined instead by the data. Secondly, small numbers of hidden units often leads to local maxima in the posterior distribution of the weights because the few available hidden units can get trapped into representing suboptimal features in the data, leaving no spare "unused" units to seek out the important features. Using a larger hidden layer tends to connect the multiple modes into ridges and thus improves mobility. Finally, in his book, Neal (1996) has shown that the prior distribution of a neural network becomes tractable for infinite-sized hidden layers. So, using many hidden units allows for a more precise specification of a neural network's prior distribution. Incidentally, overfitting is not a problem in the first justification given above because Neal's results show how to assign appropriate priors for increasing network size.

This thesis is organized as follows. As an effort to make this a self-contained work, Chapter 2 lays down the background material on Markov chains and the hybrid Monte Carlo method that is necessary and hopefully sufficient to understand the rest of what follows. Chapter 3 describes the original method of Neal. Chapter 4 presents the new method that is the subject of this thesis. Results of the investigations of the new method are presented in Chapter 5, followed by the discussions and conclusions in the final chapter.

Chapter 2

The Hybrid Monte Carlo Method

In this chapter, we introduce the hybrid Monte Carlo method, which is the main method by which we sample from the posterior distribution of a neural network.

2.1 Background on Markov chain Monte Carlo Sampling

In this section, we present some necessary background on Markov chains at a level of technical detail sufficient to explain the rest of this work. More detailed presentations may be found elsewhere, such as Feller's (1966) book.

Definition 1 (Markov chain) *A Markov chain is a series of random variables X_0, X_1, X_2, ..., etc., such that:*

$$P(X_i|X_{i-1}, X_{i-2}, ..., X_0) = P(X_i|X_{i-1}) \tag{2.1}$$

That is, given X_{i-1}, X_i is independent of all "earlier" X's.

A Markov chain is defined by the state space S in which the X_i's live, the distribution over the initial state $P(X_0)$, and the transition probability function $P(X_i|X_{i-1})$.

For us, the utility of Markov chains lies in the fact that, under the right conditions, they converge to some probability distribution regardless of starting state. In other words, independent of starting state, in the limit of large n, X_n will become a sample from a particular distribution $Q(x)$. This allows us to obtain samples from posterior distributions that arise in probabilistic inference.

Below, we present a theorem obtained from Rosenthal (1999) that tells us the conditions under which a Markov chain converges to a distribution. In this presentation, probability density functions will be used in two ways: with a state as an argument, or with a set as an argument. Thus, $p(x)$ will refer to the probability density at x, while $p(A)$ will mean the total probability mass in the set A. First, we need the following definitions. Let S be the state space of the Markov chain.

Definition 2 (Multitransition probability) *For $x \in S$ and $A \subseteq S$, we define the multitransition probability $T^n(x, A)$ as the probability of ending up in the set A after n transitions according to the Markov transition probabilities given that we started at state x. $T^n(x, A)$ is really $P(X_n \in A | X_0 = x)$. For one transition, we also write $T(x, A)$ rather than $T^1(x, A)$.*

Definition 3 (Invariant distribution) *$\pi(x)$ is an* **invariant distribution** *of a Markov chain with transitions $T(x, A)$ if, for all sets $A \subseteq S$:*

$$\pi(A) = \int \pi(dy)T(y, A) \tag{2.2}$$

where dy is a set of infinitesimal volume at state y. We also say that the Markov chain leaves $\pi(x)$ invariant.

Note that a Markov chain may not have an invariant distribution, and if it has one, it may not be unique.

Definition 4 (Total variation distance) *The* **total variation distance** *between two probability distributions p and q on S is given by:*

$$|p - q| = \sup_{A \subseteq S} |p(A) - q(A)| \tag{2.3}$$

Suppose we start a Markov chain from state $x \in S$. Then, depending on the history of transitions it takes, it will take varying numbers of steps to enter the set $A \subseteq S$ of nonzero volume, if it does at all. Let τ_A be the history-dependent random variable that denotes the first time the Markov chain enters A, i.e., $\tau_A = \inf\{n \geq 1; X_n \in A\}$. Note that τ_A could equal infinity. Then, we have the following important definitions about the mixing properties of a Markov chain:

Definition 5 (Irreducibility) *A Markov chain is* **irreducible** *if for any set $A \subseteq S$ of nonzero volume, $P_x(\tau_A < \infty) > 0$ for all starting points $x \in S$. That is, any starting point x has some probability of going to any nonzero volume within a finite number of steps.*

Definition 6 (Aperiodicity) *A Markov chain is* **aperiodic** *if there does not exist a partition of the state space $S = S_1 \bigcup S_2 \bigcup ... \bigcup S_m$ for some $m \geq 2$ such that $T(x, S_{i+1}) = 1$ for all $x \in S_i$ with $i = 1$ to $m - 1$, and $T(x, S_1) = 1$ for all $x \in S_m$.*

If a Markov chain has an invariant distribution, and it is both irreducible and aperiodic, then it converges to that invariant distribution. This theorem, presented below without proof, is a slightly modified version of the one given by Rosenthal (1999), who also proves it.

Theorem 1 (Markov Chain Convergence) *Let $T(x, A)$ be the transition probabilities for an irreducible, aperiodic Markov chain having invariant distribution $\pi(x)$ on a state space S. Then, for all $x \in S$ such that $\pi(x) \neq 0$:*

$$\lim_{n \to \infty} |T^n(x, \cdot) - \pi(\cdot)| = 0 \tag{2.4}$$

That is, as the number of transitions goes to infinity, the total variation distance between the invariant distribution and the distribution of the Markov chain started from any state x such that $\pi(x) \neq 0$ goes to 0.

Using the above theorem, we can construct Markov chains that converge to a desired distribution by ensuring that it is aperiodic, irreducible and has the target distribution as an invariant distribution. However, while the above theorem guarantees convergence in theory, it does not say anything about the speed with which convergence is achieved. This is important as the initial portion of a Markov chain is typically not representative of the invariant distribution, and needs to be discarded in order not to bias the distribution. Moreover, a badly-constructed Markov chain can converge far too slowly to be useful in practice. Nevertheless, having one that converges to the correct distribution, and knowing that it does, is a good start.

A Markov chain that is constructed to generate samples from some target distribution is known in the literature as a Markov chain Monte Carlo (MCMC) method. An example of an MCMC method that is commonly used for multivariate distributions is Gibbs sampling. Gibbs sampling consists of update steps where each variable is updated in turn. During each update, a variable is replaced by a sample from its target distribution conditional on all the other variables having their current values. Note that the new value of the variable is chosen without reference to the old value it replaces. This leaves the desired distribution invariant because the resulting multivariate state is an outcome drawn according to the target distribution. Furthermore, because all values of a variable have non-zero probability of being generated, the method is irreducible and aperiodic so long as all the variables get updated at some point.

Although it is conceptually simple, Gibbs sampling requires that one is able to sample from the conditional distribution of each variable. For complicated distributions like the neural network posteriors in this thesis, this is usually not possible. Other schemes exist that do not have this requirement. Below, we present the Metropolis algorithm with simple proposals, which requires only that we be able to evaluate the target probability density at a given state, but whose weaknesses will motivate the more sophisticated hybrid Monte Carlo method used in this thesis.

2.2 The Metropolis Algorithm with Simple Proposals

The Metropolis algorithm (Metropolis *et al.*, 1953) is a well-known algorithm for constructing a Markov chain with a desired invariant distribution.

Let $\pi(X)$ be the desired invariant distribution. Suppose our Markov chain currently has state X_i. The Metropolis algorithm amounts to the following Markov chain transition rule. First, propose a transition to a new state X_i' from the current state X_i, where the proposal probability density $M(X_i, X_i')$ must be symmetric, that is:

$$M(X_i, X_i') = M(X_i', X_i) \tag{2.5}$$

$M(X_i, X_i')$ is the probability density of going to X_i' given that we were originally at X_i. Next, accept the proposed state as the next Markov chain state X_{i+1} with the following probability:

$$P(\text{accept}) = \min\left(1, \frac{\pi(X_i')}{\pi(X_i)}\right) \tag{2.6}$$

If we reject, the state X_{i+1} is set to be the previous state X_i.

One can show that the Metropolis algorithm guarantees that the target distribution $\pi(X)$ is an invariant distribution of the Markov chain. However, it does not guarantee that the Markov chain is irreducible and aperiodic.

Let us consider the performance of the Metropolis algorithm in sampling from some target distribution when we use a simple Gaussian proposal with a fixed covariance Σ

centred on the current point X_i:

$$P(X_i') = \frac{1}{(2\pi|\Sigma|)^{d/2}} \exp\left[-\frac{1}{2}(X_i' - X_i)^T \Sigma^{-1} (X_i' - X_i)\right] \tag{2.7}$$

where d is the dimensionality of the state space. This example will illuminate the key issues in sampling a distribution with the Metropolis algorithm.

If the variance of the Gaussian proposals are too large compared with the width of the target distribution, the Metropolis algorithm almost always rejects, as proposals usually end up in regions of low target probability. Clearly, this may lead to a slow exploration of the state space, and a proposal with a smaller variance and higher acceptance rate may be better. Indeed, with the exception of special cases like two-dimensional Gaussian target distributions, fairly high acceptance rates (≈ 0.5) are better than very low acceptance rates. But in order to keep the acceptance rate high, the standard deviation of the proposal distribution must be of a size comparable to the distribution's thinnest cross section, and so the steps taken must be very small compared to the overall distribution if the distribution is very thin in one direction, but very long in others.

Thus, the first problem is that the presence of a long, thin region in a distribution constrains such a scheme to take steps which may be very small compared to the size of the overall distribution. This by itself is not so bad if the direction of the next step is somehow correlated with that of the first. However, it is not, and that is the second problem: the next step is chosen independently of the first, and because it has the possibility of doubling back on the first step, a random walk results. This effect is illustrated in Fig. 2.1.

We expect that the posterior distribution of a neural network's weights is complicated under most circumstances, and might potentially have long, narrow regions. Thus, to sample from the posterior distribution of a neural network using the Metropolis algorithm with Gaussian proposals, we would need to use proposal distributions with small variances. This leads to inefficient random walks as described above.

A method that is more appropriate for the complicated posteriors seen in neural

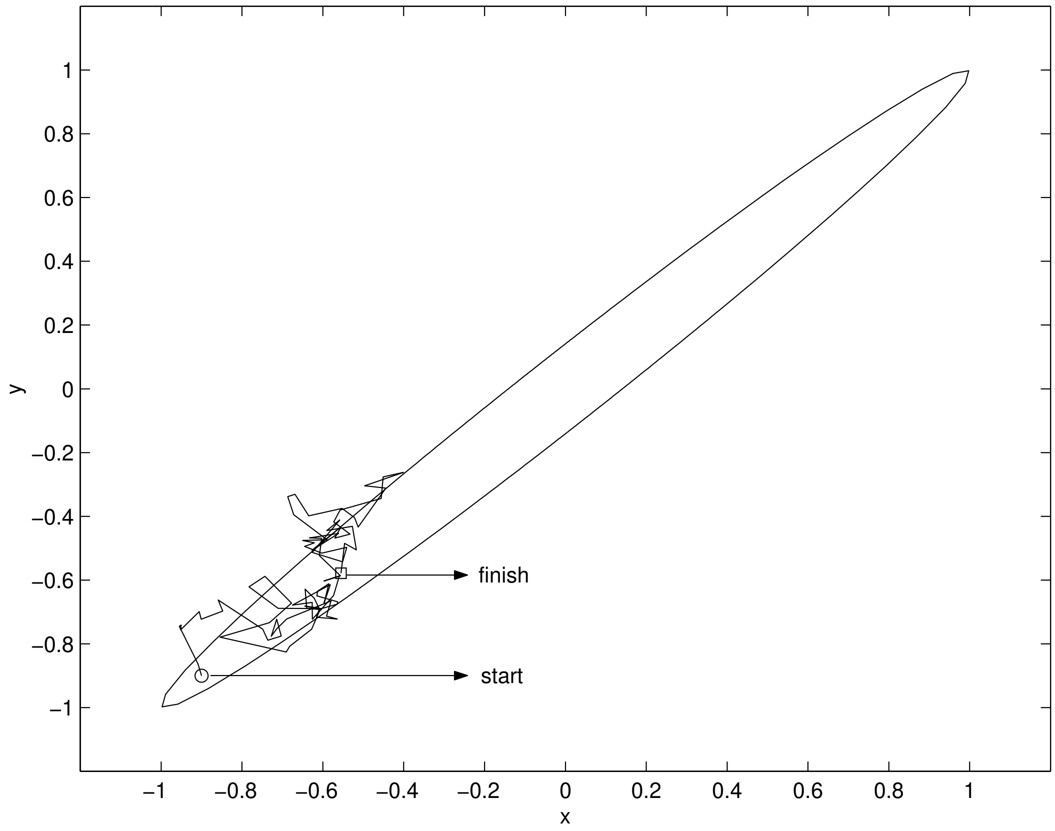


Figure 2.1: Illustration of random walk when using the Metropolis algorithm with simple Gaussian proposals to explore a two dimensional Gaussian distribution. Here, the standard deviation of the Gaussian proposals was 0.05, and of the 200 samples obtained, there were 14 rejections. (Large steps are actually more efficient for the special case of a two-dimensional Gaussian target distribution; this figure serves as an illustration of random walks only.)

network models is the hybrid Monte Carlo algorithm, which addresses the random walk problem by having auxiliary momentum variables that allow it to keep going in the same direction for many steps. This means that, in the case of hybrid Monte Carlo, the Metropolis rejection test is applied only after many steps to give it a chance at travelling a long distance.

2.3 The Hybrid Monte Carlo Method

The hybrid Monte Carlo method, first used in physics by Duane *et al.* (1987), can be thought of as a Metropolis algorithm with a sophisticated proposal. In this section, we describe how the hybrid Monte Carlo method works. Neal (1993, 1996) has written relevant expositions of this technique, but we include it here for completeness. We will use the symbols C and C' to denote normalizing constants.

In hybrid Monte Carlo, we associate a physical system with the distribution that we want to sample from. In essence, we simulate the movement of a particle moving in a potential energy well equal to the negative log of the probability density for the distribution that we want to sample from. Each iteration consists of randomizing the velocity of this particle, simulating its motion for some time, and then obtaining its position, which becomes a new sample.

Suppose that we wish to sample from the distribution $P(\mathbf{q})$, where $\mathbf{q} \in \Re^d$. $\Re^d$ is then the state space of our associated physical system, and $\mathbf{q}$ is a state of the system. We augment each state variable q_i with a momentum variable p_i, and we define the Hamiltonian:

$$H(\mathbf{q}, \mathbf{p}) = E(\mathbf{q}) + K(\mathbf{p}) \tag{2.8}$$

where the potential energy $E(\mathbf{q})$ is obtained from the desired distribution as:

$$E(\mathbf{q}) = -\log P(\mathbf{q}) - \log Z \tag{2.9}$$

for any choice of Z, and the kinetic energy $K(\mathbf{p})$ is defined with the set of masses $\{m_i\}_{i=1}^{d}$:

$$K(\mathbf{p}) = \sum_{i=1}^{d} \frac{p_i^2}{2m_i} \tag{2.10}$$

Hybrid Monte Carlo allows us to set up a Markov chain that converges to $C' \exp(-E(\mathbf{q}) - K(\mathbf{p}))$ as its unique invariant distribution. By ignoring the values of $\mathbf{p}$, we obtain samples of $\mathbf{q}$ drawn from the target distribution $P(\mathbf{q})$, since this is the marginal distribution.

In the hybrid Monte Carlo method, we simulate the time evolution of the physical system with the above Hamiltonian using Hamiltonian dynamics, which is given by:

$$\begin{aligned} \frac{d\mathbf{p}}{dt} &= -\nabla E(\mathbf{q}) \\ \frac{d\mathbf{q}}{dt} &= \frac{\mathbf{p}}{m_i} \end{aligned} \tag{2.11}$$

Let us assume for now that we have the ability to do the simulation with perfect accuracy. Since Hamiltonian dynamics leaves H invariant and keeps phase space volume constant (see Appendix A), simulating the system over any fixed length of time yields a new pair $(\mathbf{q}, \mathbf{p})$ that leaves any distribution that is a function of H invariant. In particular, it leaves $C' \exp(-H(\mathbf{q}, \mathbf{p})) = C' \exp(-E(\mathbf{q}) - K(\mathbf{p}))$ invariant.

However, a Markov chain that consists of only this update is not irreducible, as all points generated from a starting point never leave a hypershell of constant H, thus violating the irreducibility requirement for Thm 1. To rectify this situation, we update the momentum variables in such a way that the Markov chain has some chance of reaching all the other values of H after some number of iterations. Specifically, before the simulation of Hamiltonian dynamics, we replace all the momentum variables by new values drawn from the distribution $C \exp(-K(\mathbf{p}))$. Again, this update leaves the distribution $C' \exp(-E(\mathbf{q}) - K(\mathbf{p}))$ invariant since it draws $\mathbf{p}$ from the correct conditional distribution, which happens to be independent of $\mathbf{q}$.

So the joint update consisting of the momentum update followed by the Hamiltonian dynamics simulation is a Markov chain that leaves $C' \exp(-E(\mathbf{q}) - K(\mathbf{p}))$ invariant. If we can construct such a Markov chain and prove that it is irreducible and aperiodic, then we have a Markov chain that converges to the desired distribution $C' \exp(-E(\mathbf{q}) - K(\mathbf{p}))$, and we can obtain the desired samples by ignoring $\mathbf{p}$. Moreover, if, during the Hamiltonian simulation, we follow the dynamical trajectory of a state for a long time, we might obtain a state that is much less correlated with the original state than a Metropolis algorithm with simple proposals.

The method presented thus far is not the actual hybrid Monte Carlo algorithm, but it contains all the essential ideas. What is different about hybrid Monte Carlo is that, in reality, we are unable to simulate Hamiltonian dynamics perfectly. Owing to the fact that neural network models are highly complex and so lead to non-integrable Hamiltonians, we have to settle for an approximate discretized simulation of Hamiltonian dynamics, followed by a Metropolis rejection test that ensures that $C' \exp(-H)$ is kept invariant. As before, the update consisting of momentum resampling followed by the simulation keeps the desired distribution $C' \exp(-H)$ invariant. The conditions under which this Markov chain is irreducible and aperiodic depends on its details, and we delay discussing this until Section 2.3.3.

Finally, we note that the discretized simulation is now also a Metropolis proposal, with the probability of rejection increasing as the simulation error as measured by the rise in H increases. When we do the simulation well, we keep H almost invariant over long trajectories, so it is in our interests to do the simulation as well as we can in order to have a high acceptance rate, even though simulation errors are corrected by the Metropolis rejection test to give the exact desired distribution.

2.3.1 Leapfrog Proposals

Because the discretized simulation used as a Metropolis proposal is deterministic, the standard reversibility condition for Metropolis proposals (Eqn. 2.5) does not apply. Instead, the equivalent reversibility conditions for deterministic proposals are that the mapping that is the Metropolis proposal is its own inverse, and that it has Jacobian 1.

That is:

Theorem 2 (Deterministic Proposals for Metropolis Algorithm) *If* $\mathbf{Y} = \mathbf{M}(\mathbf{X})$ *is a deterministic mapping that satisfies the two conditions:*

$$\left|\frac{\partial \mathbf{Y}}{\partial \mathbf{X}}\right| = 1 \quad \textit{(volume conservation)} \tag{2.12}$$

$$\mathbf{M}(\mathbf{M}(\mathbf{X})) = \mathbf{X} \quad \textit{(reversibility)} \tag{2.13}$$

then by accepting the update $\mathbf{X} \leftarrow \mathbf{M}(\mathbf{X})$ *with probability* $\min(1, \pi(\mathbf{M}(\mathbf{X}))/\pi(\mathbf{X}))$ *and rejecting it otherwise, the distribution* $\pi(\mathbf{X})$ *is left invariant.*

We give the proof of this in Appendix B.

To satisfy the two conditions of volume conservation and reversibility, we use a deterministic proposal composed of "leapfrog updates" to simulate the Hamiltonian dynamics by performing a trajectory of l steps each lasting ϵ time. At the end of each trajectory, we negate the momentum $\mathbf{p}$. Each leapfrog update consists of:

$$\begin{aligned} p_i(t+\frac{\epsilon}{2}) &= p_i(t) - \frac{\epsilon}{2}\frac{\partial E}{\partial q_i}(\mathbf{q}(t)) \quad \text{for each } i = 1..d \\ q_i(t+\epsilon) &= q_i(t) + \epsilon\frac{p_i(t+\epsilon/2)}{m_i} \quad \text{for each } i = 1..d \\ p_i(t+\epsilon) &= p_i(t+\frac{\epsilon}{2}) - \frac{\epsilon}{2}\frac{\partial E}{\partial q_i}(\mathbf{q}(t+\epsilon)) \quad \text{for each } i = 1..d \end{aligned} \tag{2.14}$$

Note that, in the above scheme, all the components are updated before moving on to the next line. For instance, all the components of $p_i(t+\frac{\epsilon}{2})$ are calculated before the update for $\mathbf{q}$ is computed.

To see that the leapfrog update satisfies the volume conservation condition, we note that the change in each component of each state variable does not depend on itself, and so each component's update amounts to a shear, which is a volume-preserving transformation, and which therefore has Jacobian 1. This is discussed in greater detail in Appendix C.

Also, it is easy to check that the negation of the momentum at the end of a leapfrog trajectory of multiple leapfrog steps means that, if a trajectory takes us from point A to point B, then starting at B takes us back to A. Thus, leapfrog trajectories also satisfy the reversibility condition.

We summarize the algorithm for the hybrid Monte Carlo in Algorithm 1 and Algorithm 2. Note that the momentum negation is not implemented as the momentum is replaced by resampling before the next leapfrog trajectory anyways.

The algorithm discussed thus far avoids random walks by allowing long trajectory lengths. However, the actual algorithm implemented by Neal (1996) has an additional optimization of the stepsizes that estimates the local second derivative of the potential energy in order to take steps that are appropriately scaled in the various dimension. This is discussed next.

2.3.2 Stepsize Selection

In using the hybrid Monte Carlo method, the question of what values to choose for the stepsize ϵ and for the masses m_i naturally arises. As we shall see, it turns out that choosing the masses is equivalent to choosing different stepsizes in different dimensions of the state variable, and the careful choice of these stepsizes is necessary for hybrid Monte Carlo to perform well.

It is clear from Eqn. 2.14 that, since ϵ is the timestep of a discretized simulation of Hamiltonian dynamics, large values of ϵ cause an inaccurate simulation so that H can wander far from its initial value. In particular, such an inaccurate simulation will typically land the proposed state in a region of low target probability. This is analogous to using a Gaussian proposal with too large a variance in the Metropolis algorithm example of Section 2.2, and so having to reject frequently. Thus, keeping rejection rates low requires careful selection of ϵ that is low enough, and yet not so low that we explore the distribution laboriously.

Algorithm 1 HybridMonteCarlo($nsamples, l, \epsilon, \mathbf{q}_{init}$)

$\mathbf{q}^0 \leftarrow \mathbf{q}_{init}$

for each component j **do**

  $\mathbf{p}^0_j \leftarrow N(\text{mean} = 0, \text{variance} = m_j)$

end for

for $i = 1$ to $nsamples$ **do**

  for each component j **do**

    $\mathbf{p}'_j \leftarrow N(\text{mean} = 0, \text{variance} = m_j)$

  end for

  $\mathbf{p}^i \leftarrow \mathbf{p}'$

  $\mathbf{q}^i \leftarrow \mathbf{q}^{i-1}$

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

    $(\mathbf{q}^i, \mathbf{p}^i) \leftarrow LeapfrogUpdate(\mathbf{q}^i, \mathbf{p}^i, \epsilon)$

  end for

  if $U[0,1] > \min[1, \exp(-H(\mathbf{q}^i, \mathbf{p}^i) + H(\mathbf{q}^{i-1}, \mathbf{p}^{i-1}))]$ **then**

    $\mathbf{q}^i \leftarrow \mathbf{q}^{i-1}$

    $\mathbf{p}^i \leftarrow \mathbf{p}'$

  end if

end for

Return $\{\mathbf{q}^i, \mathbf{p}^i\}_{i=0}^{nsamples}$

Algorithm 2 LeapfrogUpdate($\mathbf{q}, \mathbf{p}, \epsilon$)

for each component i **do**

$p_i \leftarrow p_i - (\epsilon/2) \times \partial(E/\partial q_i)(\mathbf{q})$

end for

for each component i **do**

$q_i \leftarrow q_i + (\epsilon/m_i) \times p_i$

end for

for each component i **do**

$p_i \leftarrow p_i - (\epsilon/2) \times \partial(E/\partial q_i)(\mathbf{q})$

end for

Return $\mathbf{q}, \mathbf{p}$

As described by Neal (1996), for a toy quadratic Hamiltonian of the form:

$$H = \frac{q^2}{2\sigma^2} + \frac{p^2}{2} \tag{2.15}$$

H diverges under the leapfrog discretization if a stepsize $\epsilon > 2\sigma$ is used, whereas H stays bounded if $\epsilon < 2\sigma$. To transfer this result to a general non-quadratic $H(\mathbf{q}, \mathbf{p})$, we note that, near equilibrium, samples are usually obtained near local minima of $E(\mathbf{q})$, where it can be approximated by its Taylor expansion to second order. Thus, we expect a stepsize $\epsilon \sim (\partial^2 E/\partial q^2)^{-1/2}$ to be appropriate near equilibrium.

In the case where the state space is multi-dimensional but the Taylor expansion to second order has no correlations between its dimensions, we could set:

$$\epsilon \sim \min_i \left(\frac{\partial^2 E}{\partial q_i^2}\right)^{-\frac{1}{2}} \tag{2.16}$$

in order to prevent the leapfrog simulation from diverging. But if the Taylor expansion has correlations between its dimensions, the above might not be small enough, as the stability of the leapfrog updates is constrained by the narrowest cross-section, which might not be axis-aligned at all. In general, the stepsize ϵ has to be adjusted downwards by different amounts depending on the shape and orientation of the energy function. To take this into account, we introduce an operator-defined tuning parameter η, which we call the stepsize adjustment factor, and which controls the stepsizes as follows:

$$\epsilon = \eta \times \min_i \left(\frac{\partial^2 E}{\partial q_i^2} \right)^{-\frac{1}{2}} \tag{2.17}$$

However, choosing the same stepsize to use in all directions may cause slow, random walk-like exploration in those directions unless we use very long trajectories, which may be unnecessarily computationally intensive. The underlying problem is that of making a move that is compatible with the local length scales of the distribution. It is the same problem that we encountered earlier in considering the Metropolis algorithm with simple proposals, but under a slightly different guise.

Clearly, it is preferable to use different and appropriate stepsizes for each direction, the values of which we choose by looking at the local length scales of the distribution. Ideally, we would like to set the stepsize for direction i, ϵ_i, based on the width of the potential energy bowl in the direction q_i:

$$\epsilon_i = \eta \left(\frac{\partial^2 E}{\partial q_i^2} \right)^{-\frac{1}{2}} \tag{2.18}$$

However, one may wonder if the leapfrog update with different stepsizes for different components still simulates Hamiltonian dynamics. The answer is that the masses are the extra degrees of freedom that enable us to implement different stepsizes in different directions and still keep H approximately constant. To see this, we first note that, if we rewrite the leapfrog equations in terms of $\tilde{p}_i \equiv p_i/\sqrt{m_i}$, they become:

$$\begin{aligned}
\tilde{p}_i(t + \frac{\epsilon}{2}) &= \tilde{p}_i(t) - \frac{1}{2}\frac{\epsilon}{\sqrt{m_i}}\frac{\partial E}{\partial q_i}(\mathbf{q}(t)) \\
q_i(t+\epsilon) &= q_i(t) + \frac{\epsilon}{\sqrt{m_i}}\tilde{p}_i(t+\epsilon/2) \\
\tilde{p}_i(t+\epsilon) &= \tilde{p}_i(t + \frac{\epsilon}{2}) - \frac{1}{2}\frac{\epsilon}{\sqrt{m_i}}\frac{\partial E}{\partial q_i}(\mathbf{q}(t+\epsilon))
\end{aligned} \tag{2.19}$$

We set the stepsizes $\epsilon_i \equiv \epsilon/\sqrt{m_i}$, and rewrite the leapfrog updates as:

$$\begin{aligned}
\tilde{p}_i(t+\frac{\epsilon}{2}) &= \tilde{p}_i(t) - \frac{\epsilon_i}{2}\frac{\partial E}{\partial q_i}(\mathbf{q}(t)) \\
q_i(t+\epsilon) &= q_i(t) + \epsilon_i \tilde{p}_i(t+\epsilon/2) \\
\tilde{p}_i(t+\epsilon) &= \tilde{p}_i(t+\frac{\epsilon}{2}) - \frac{\epsilon_i}{2}\frac{\partial E}{\partial q_i}(\mathbf{q}(t+\epsilon))
\end{aligned} \tag{2.20}$$

These new updates are exactly equivalent to the original ones in Eqns. 2.14, except we work in terms of rescaled momenta. Should we choose to, we can always recover the old momenta after an update. But rather than using the original leapfrog updates, we can work in terms of $\tilde{p}_i$ using the new mass-absorbed updates. We note the following important facts about one mass-absorbed update of $\mathbf{q}$ and $\mathbf{p}$ using Eqns. 2.20:

Fact 1 *Each mass-absorbed leapfrog update keeps $H(\mathbf{q}, \tilde{\mathbf{p}}) = E(\mathbf{q}) + \sum_{i=1}^{d} \tilde{p}_i^2/2$ approximately invariant. This is because it keeps $H(\mathbf{q}, \mathbf{p}) = E(\mathbf{q}) + \sum_{i=1}^{d} p_i^2/2m_i$ approximately invariant, and the two H's are equal.*

Fact 2 *Each mass-absorbed leapfrog update conserves phase space volume in the state space $(\mathbf{q}, \tilde{\mathbf{p}})$ since $\tilde{p}_i$ is related to p_i merely by the scale factor $\sqrt{m_i}$. This holds if the ϵ_i's are set independently of the current value of $\mathbf{q}$ or $\mathbf{p}$.*

Fact 3 *Each mass-absorbed leapfrog update is reversible so long as the ϵ_i's are set without using the current values of $\mathbf{q}$ and $\mathbf{p}$, since these are different at the beginning and at the end of each step.*

In view of these three facts, we have the following revised algorithm that stores $\tilde{\mathbf{p}}$ instead of $\mathbf{p}$. Before the leapfrog updates, we estimate the stepsize ϵ_i:

$$\epsilon_i \approx \eta \left(\frac{\partial^2 E}{\partial q_i^2} \right)^{-\frac{1}{2}} \tag{2.21}$$

independently of the current state using some problem-dependent heuristic. The leapfrog trajectory now consists of mass-absorbed updates, at the end of which we apply the Metropolis rejection test using the Hamiltonian $H(\mathbf{q}, \tilde{\mathbf{p}}) = E(\mathbf{q}) + \sum_{i=1}^{d} \tilde{p}_i^2/2$. Since the proposals in the $(\mathbf{q}, \tilde{\mathbf{p}})$ state space are reversible and conserve phase space volume, the Metropolis rejection test can be used to produce an update that keeps $\exp(-H(\mathbf{q}, \tilde{\mathbf{p}}))$ invariant. Thus, we have a Markov chain that leaves $\exp(-E(\mathbf{q}) - \sum_{i=1}^{d} \tilde{p}_i^2/2)$ invariant.

We summarize hybrid Monte Carlo with stepsize selection in Algorithm 3, where the function Stepsizes() computes the appropriate stepsize to use for each component. We call the function LeapfrogUpdate() with a vector for its stepsize parameter unlike the scalar in Algorithm 2, but what we mean here should be clear. The setting of the stepsizes depends on the exact problem at hand. For a neural network model, Neal (1996) sets them based on the current values of the hyperparameters. These do not change over the course of a leapfrog trajectory in the scheme presented in his book, so that leapfrog trajectories are reversible.

2.3.3 Convergence of Hybrid Monte Carlo

In this section, we discuss the conditions under which this Markov chain algorithm converges to a unique invariant distribution.

Thm. 1 tells us that, in order for hybrid Monte Carlo to converge to a unique distribution, it must be both irreducible and aperiodic. Whether or not this is true depends on the details of the Hamiltonian, the leapfrog trajectory length, and the stepsize adjustment factor. Although we have no formal proof, we have strong reasons to believe that both conditions are satisfied in most neural network applications, whose Hamiltonian dynamics are highly nonlinear and whose Hamiltonians have values that are finite for finite values of the state parameters.

Let us first discuss peridocity. For most problems involving complex nonlinear Hamiltonians such as the ones we encounter in neural network applications, we expect that periodicity is unlikely and, if it should appear, is pathological rather than typical. An example of such an unlikely periodicity is a case where the Hamiltonian dynamics takes us exactly halfway or completely around a hypershell of constant H, and this periodicity

Algorithm 3 HybridMonteCarloWithStepsizes($nsamples, l, \eta, \mathbf{q}_{init}$)

$\mathbf{q}^0 \leftarrow \mathbf{q}_{init}$

$\mathbf{p}^0 \leftarrow N(\text{mean} = 0, \text{ variance} = I)$

for $i = 1$ to $nsamples$ **do**

  $\mathbf{p}' \leftarrow N(\text{mean} = 0, \text{ variance} = I)$

  $\boldsymbol{\epsilon} \leftarrow \eta \times Stepsizes()$

  $\mathbf{p}^i \leftarrow \mathbf{p}'$

  $\mathbf{q}^i \leftarrow \mathbf{q}^{i-1}$

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

    $(\mathbf{q}^i, \mathbf{p}^i) \leftarrow LeapfrogUpdate(\mathbf{q}^i, \mathbf{p}^i, \boldsymbol{\epsilon})$

  end for

  if $U[0,1] > \min[1, \exp(-H(\mathbf{q}^i, \mathbf{p}^i) + H(\mathbf{q}^{i-1}, \mathbf{p}^{i-1}))]$ **then**

    $\mathbf{q}^i \leftarrow \mathbf{q}^{i-1}$

    $\mathbf{p}^i \leftarrow \mathbf{p}'$

  end if

end for

Return $\{\mathbf{q}^i, \mathbf{p}^i\}_{i=0}^{nsamples}$

persists in hypershells of all values of H so that momentum resampling does not avail us of an escape from periodicity. Such a situation appears very unlikely for the highly nonlinear neural network Hamiltonians that we use, so we expect that we will probably always have aperiodicity in practice. Still, if one wishes to be on the safe side, one can vary the stepsize adjustment factors randomly over a small range, and that should remove any periodicities (Mackenzie, 1989). This modification was not implemented in our version of the algorithm.

Irreducibility depends on the exact shape of the potential energy surface. Let us assume for now that our hybrid Monte Carlo algorithm is able to simulate Hamiltonian dynamics perfectly. Then it seems intuitively clear that, so long as the potential energy does not become infinity for finite values of $\mathbf{q}$, hybrid Monte Carlo should be irreducible. In particular, while moving around in a local minimum, it always has some probability of gaining a sufficiently large kinetic energy from momentum replacement to leave it and visit other parts of state space. This can only be prevented if that local minimum is bounded by walls of infinite potential energy. And since our neural network Hamiltonian is always finite for finite values of $\mathbf{q}$, we expect that we will always have irreducibility. However, hybrid Monte Carlo really only simulates Hamiltonian dynamics approximately, so it is conceivable that a finite potential well could be a trap like an infinite one. Nevertheless, the fact that hybrid Monte Carlo does simulate Hamiltonian dynamics is reason to believe that the above argument for irreducibility should usually apply.

Chapter 3

Hyperparameter Updates Using Gibbs Sampling

3.1 Neural Network Architecture

In this thesis, we concern ourselves with a neural network with one hidden layer only. The techniques described here can readily be extended to networks with more hidden layers.

The neural network model used here will have N_x input units, N_h hidden units and N_y output units. The parameters of the neural network are referred to collectively as θ. As shown in Fig. 3.1, they are:

Input-to-hidden weights	$U = \{u_i\}_{i=1}^{N_u}$
Hidden-to-output weights	$V = \{v_i\}_{i=1}^{N_v}$
Hidden biases	$A = \{a_i\}_{i=1}^{N_a}$
Output biases	$B = \{b_i\}_{i=1}^{N_b}$

where $N_u = N_x N_h$, $N_a = N_h$, $N_v = N_h N_y$ and $N_b = N_y$.

We will sometimes use the alternative notation:

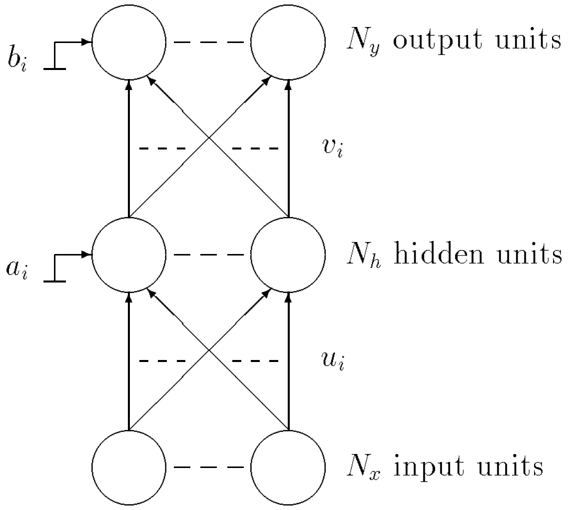


Figure 3.1: Neural network architecture used in this work

Input-to-hidden weight from unit i to unit j	U_{ij}
Hidden-to-output weight from unit j to unit k	V_{jk}
Bias on hidden unit j	A_j
Bias on output unit k	B_k

In this neural network, we use the $\tanh(\cdot)$ nonlinearity, and it occurs only at the single hidden layer. Thus, for input vector $\{x_i\}_{i=1}^{N_x}$ the j'th hidden unit output is:

$$h_j = \tanh\left(\sum_{i=1}^{N_x} U_{ij} x_i + A_j\right) \tag{3.1}$$

while the output at unit k has no nonlinearity:

$$f_k = \sum_{j=1}^{N_h} V_{jk} h_j + B_k \tag{3.2}$$

3.2 Neural Network Model of the Data

The neural network model of the data is as follows. Let $\boldsymbol{\delta}$ be the error in the output of the neural network for training input $\mathbf{x}$ and target $\mathbf{y}$:

$$\boldsymbol{\delta} = \mathbf{f}(\mathbf{x}; \theta) - \mathbf{y} \tag{3.3}$$

We assume that each component of $\boldsymbol{\delta}$ has precision (or inverse variance) τ_δ. Then, given an input case $\mathbf{x}$, the parameters of the network θ, and τ_δ, the probability of observing the target $\mathbf{y}$ is:

$$P(\mathbf{y}|\theta,\tau_\delta,\mathbf{x}) = \left(\frac{\tau_\delta}{2\pi}\right)^{N_y/2} \exp\left[-\frac{\tau_\delta}{2}|\mathbf{f}(\mathbf{x};\theta)-\mathbf{y}|^2\right] \tag{3.4}$$

The prior distributions of the four groups of parameters are normal with means 0 and precisions τ_*. The asterisk indicates the corresponding group of network parameters, and may be u, v, a or b. For instance, for the input-to-hidden weights U:

$$P(U|\tau_u) = \left(\frac{\tau_u}{2\pi}\right)^{N_u/2} \exp\left(-\frac{\tau_u}{2}\sum_{i=1}^{N_u} u_i^2\right) \tag{3.5}$$

The prior distribution of the parameters θ is simply the joint distribution:

$$P(\theta|\gamma) = P(U|\tau_u)P(V|\tau_v)P(A|\tau_a)P(B|\tau_b) \tag{3.6}$$

where γ denotes $\{\tau_\delta,\tau_u,\tau_v,\tau_a,\tau_b\}$. γ is the set of hyperparameters which control the prior distribution of each group of parameters. These prior distributions keep the network parameters small, and amount to a principled formulation of the weight decay terms found in the neural network training literature (see Bishop, 1995).

Rather than fixing the hyperparameters, we allow them to vary also, and we let them each have gamma distributions. For instance, for τ_u:

$$P(\tau_u) = \frac{(\alpha_u/2\omega_u)^{\alpha_u/2}}{\Gamma(\alpha_u/2)}\tau_u^{\alpha_u/2-1}\exp(-\tau_u\alpha_u/2\omega_u) \tag{3.7}$$

which is a gamma distribution with mean ω_u and shape parameter α_u for each τ_u. The prior distribution of the hyperparameters $P(\gamma)$ is simply the joint distribution:

$$P(\gamma) = P(\tau_\delta)P(\tau_u)P(\tau_v)P(\tau_a)P(\tau_b) \tag{3.8}$$

In this work, α_* and ω_* are fixed by hand.

3.3 Posterior Distributions of the Parameters and the Hyperparameters

We can infer the posterior distributions for the parameters $\theta = \{U, V, A, B\}$ and the hyperparameters $\gamma = \{\tau_\delta, \tau_u, \tau_v, \tau_a, \tau_b\}$ when the training data is observed. In the Monte Carlo approach, we do this by obtaining samples from the distribution $P(\theta, \gamma|x, y)$, where $x = \{\mathbf{x}^c\}_{c=1}^{N_c}$ and $y = \{\mathbf{y}^c\}_{c=1}^{N_c}$ are the training data.

Neal (1996) samples from $P(\theta, \gamma|x, y)$ by obtaining a series of Markov chain samples $\{\theta_i, \gamma_i\}_{i=1}^{N_s}$ where $\gamma_i \sim P(\gamma|\theta = \theta_{i-1}, x, y)$ is obtained by Gibbs sampling, and θ_i is obtained from θ_{i-1} as a hybrid Monte Carlo update that leaves the distribution $P(\theta|\gamma = \gamma_i, x, y)$ invariant. During the first iteration, γ_1 is set to some moderate values. We discuss the convergence properties of this Markov chain in Section 3.4.1.

In the remainder of this section, we give the distributions $P(\theta|\gamma, x, y)$ and $P(\gamma|\theta, x, y)$, which are required for the above sampling scheme.

Using Bayes' Rule, we obtain the posterior distribution for θ as:

$$\begin{aligned} P(\theta|\gamma, x, y) &= \frac{P(y|\theta, \gamma, x)P(\theta|\gamma, x)}{P(y|\gamma)} \\ &\propto P(y|\theta, \tau_\delta, x)P(\theta|\gamma) \\ &\propto \left[\prod_{c=1}^{N_c} P(\mathbf{y}^c|\theta, \tau_\delta, \mathbf{x}^c)\right] P(U|\tau_u)P(V|\tau_v)P(A|\tau_a)P(B|\tau_b) \end{aligned} \tag{3.9}$$

The reader is referred to Eqns. 3.4 and 3.5 for the full expansion of the above expression.

The posterior distribution for γ, $P(\gamma|\theta, x, y)$, is the probability of the hyperparameters conditioned on θ, x and y:

$$P(\gamma|\theta, x, y) = P(\tau_\delta|\theta, x, y)P(\tau_u|\theta, x, y)P(\tau_v|\theta, x, y)P(\tau_a|\theta, x, y)P(\tau_b|\theta, x, y) \tag{3.10}$$

Consider the hyperparameter τ_*. Since each τ_* is the precision of its group of parameters, it can be inferred solely from those parameters independently of x and y. For instance, for τ_u:

$$\begin{aligned} P(\tau_u|\theta, x, y) &= P(\tau_u|\{u_i\}_{i=1}^{N_u}) \\ &\propto P(\{u_i\}_{i=1}^{N_u}|\tau_u)P(\tau_u) \\ &\propto \tau_u^{(\alpha_u+N_u)/2-1} \exp\left[-\frac{\tau_u}{2}\left(\frac{\alpha_u}{\omega_u} + \sum_{i=1}^{N_u} u_i^2\right)\right] \end{aligned} \tag{3.11}$$

τ_δ, on the other hand, is the noise in each component of $\mathbf{y}^c$. τ_δ is inferred from the errors $\boldsymbol{\delta}^c = \mathbf{f}(\mathbf{x}^c; \theta) - \mathbf{y}^c$:

$$\begin{aligned} P(\tau_\delta|\theta, x, y) &\propto P(\{\boldsymbol{\delta}^c\}_{c=1}^{N_c}|\tau_\delta, \theta, x)P(\tau_\delta) \\ &\propto \tau_\delta^{(\alpha_\delta+N_cN_y)/2-1} \exp\left[-\frac{\tau_\delta}{2}\left(\frac{\alpha_\delta}{\omega_\delta} + \sum_{c=1}^{N_c} |\mathbf{f}(\mathbf{x}^c; \theta) - \mathbf{y}^c|^2\right)\right] \end{aligned} \tag{3.12}$$

Note that Eqns. 3.11 and 3.12 are gamma distributions.

3.4 Sampling From the Posterior Distributions of the Parameters and the Hyperparameters

Since Eqns. 3.11 and 3.12 are gamma distributions, independent samples for the hyperparameters can be drawn using well-known techniques (Devroye, 1986).

Drawing samples from the posterior distribution of the parameters is more difficult. For instance, standard Gibbs sampling cannot be used because the conditional distribution of each network parameter can be a very complicated function due to the training error terms. A simple Metropolis method suffers from random walks as previously described. So instead, Neal (1996) obtains samples from the network parameters using the hybrid Monte Carlo technique with stepsize selection as described in Algorithm 3 of Section 2.3. To sample from the posterior distribution for θ, the potential energy is set as follows:

$$E(\theta) = -\log P(\theta|\gamma, x, y)$$
$$= \frac{\tau_\delta}{2}\sum_{c=1}^{N_c}|\mathbf{f}(\mathbf{x}^c;\theta) - \mathbf{y}^c|^2 + \frac{\tau_u}{2}\sum_{i=1}^{N_u}u_i^2 + \frac{\tau_v}{2}\sum_{i=1}^{N_v}v_i^2 + \frac{\tau_a}{2}\sum_{i=1}^{N_a}a_i^2 + \frac{\tau_b}{2}\sum_{i=1}^{N_b}b_i^2 + const \tag{3.13}$$

where the constant is immaterial because it does not affect the dynamics of the system that gives rise to the desired probability distribution. This constant corresponds to prefactors in the distribution $P(\theta|\gamma, x, y)$ that do not depend on θ. The parameters θ correspond to $\mathbf{q}$ in Algorithm 3.

3.4.1 Convergence of the Algorithm

In this section, we discuss the conditions under which this Markov chain algorithm converges to a unique invariant distribution.

Recall that our algorithm alternately updates the hyperparameters by Gibbs sampling and the parameters using hybrid Monte Carlo. In Section 2.3.3, we have already discussed the reasons why hybrid Monte Carlo by itself should converge to a unique distribution. The question is, combined with the hyperparameter Gibbs sampling update step, does the resulting Markov chain converge?

A Markov chain update is periodic so long as it is periodic in one of the parameters of its state space, so the fact that Gibbs sampling of the hyperparameters has non-zero probability of producing any value does not immediately imply aperiodicity. However, when the hyperparameters are updated from one iteration to the next, hybrid Monte Carlo sees a random modification of the potential energy surface that, if anything, would prevent systematic behaviour like periodicity. Thus, the Gibbs sampling step renders periodicity even more unlikely than ever.

When the hyperparameters change from one iteration to the next, hybrid Monte Carlo sees a modification of the potential energy surface that does not introduce any infinite barriers, so the argument from Section 2.3.3 still applies, and we expect that, with sufficient iterations, all regions of the network parameters' state space can be visited regardless of the values the hyperparameters have. Thus, we expect that all regions of the joint state space containing both parameter and hyperparameter can be visited after a sufficient number of iterations, and our algorithm should be irreducible.

Although we have no formal proof, based on the above arguments, we expect that our Markov chain does converge to a unique distribution.

3.5 Inefficiency Due to Gibbs Sampling of Hyperparameters

Despite the fact that we avoid random walks in network parameter space given the hyperparameters, Gibbs sampling of the hyperparameters can lead to a slow random walk in the joint state space of the hyperparameters and the network parameters. This is because the distribution of the parameters conditional on the hyperparameters is restricted by the conditioning on the hyperparameters; this restricts the possible values the parameters are likely to visit, and so the distribution of the hyperparameters conditional on the parameters are unlikely to change much from the previous iteration in order to be consistent with the parameters.

This is especially apparent when there are many hidden units. Consider sampling the input-to-hidden weights u_i given τ_u. When there are many of them, they represent their distribution well, with a variance close to $1/\tau_u$. Thus, when τ_u is Gibbs-sampled, we are likely to obtain a value close to τ_u again, and so the Markov chain becomes highly correlated from sample to sample.

This problem can be alleviated when there are many training cases. In Eqn. 3.13, the prior terms will be of order $(N_u + N_v + N_a + N_b)/2$. When $N_c N_y \gg N_u + N_v + N_a + N_b$, the likelihood term has a stronger effect in determining the shape of the potential energy bowl sampled from, and so the weights move mostly to fit the data rather than to satisfy the prior constraints imposed by their precisions, and so in this case, the hyperparameters become slaved to the weights, which in turn are well-determined by the data.

When we do not have the option of obtaining more training cases, and we use large numbers of hidden units, the random walk described above becomes an issue, and may slow the method down so much as to render it impractical to use. The next chapter introduces the solution that is considered in this thesis: that of updating the hyperparameters using hybrid Monte Carlo as opposed to Gibbs sampling.

Chapter 4

Hyperparameter Updates Using Hamiltonian Dynamics

4.1 The New Scheme

To overcome the slow movement of the hyperparameters when using Gibbs sampling, we propose to update the hyperparameters using Hamiltonian dynamics in the same way as the parameters.

4.1.1 The Idea

The problem with the old scheme is that the alternating updates of the hyperparameter and its associated parameters cause them to pin each other down, resulting in Markov chain moves that are small compared to the overall distribution, and which can double back since Markov chains have only state memory and no momentum memory. This doubling back is similar to the random walk behaviour of the Metropolis algorithm with simple proposals. Since hybrid Monte Carlo is our way of overcoming that, we hope that hybrid Monte Carlo, by producing trajectories that can keep going in the same general direction for long distances, may also allow hyperparameters to travel long distances in a single leapfrog trajectory without doubling back. This should lead to gains in how rapidly the parameters and the hyperparameters sample the posterior distribution.

4.1.2 The New Scheme in Detail

Rather than updating the parameters conditional on the hyperparameters, and vice versa, our aim is now to update the parameters θ and the hyperparameters γ jointly according to the joint posterior distribution:

$$\begin{aligned} P(\theta,\gamma|x,y) &= P(\gamma)P(\theta|\gamma)P(y|\theta,\gamma,x)/P(y|x) \\ &\propto P(\gamma)P(\theta|\gamma)P(y|\theta,\tau_\delta,x) \end{aligned} \tag{4.1}$$

where we have dropped the normalizing constant $P(y|x)$ and used the fact that, given θ, y's dependence on the hyperparameters γ is restricted to just the noise hyperparameter τ_δ.

From Eqns. 3.4, 3.6 and 3.8, we get:

$$\begin{aligned} E(\theta,\gamma) &= -\log(P(\theta,\gamma|x,y)) \\ &= \left(\sum_{*=u,v,a,b} E_*\right) - \left(\frac{\alpha_\delta+N_\delta}{2}-1\right)\log(\tau_\delta) + \frac{\tau_\delta}{2}\left[\frac{\alpha_\delta}{\omega_\delta} + \sum_{c=1}^{N_c} |\mathbf{f}(\mathbf{x}^c;\theta)-\mathbf{y}^c|^2\right] \end{aligned} \tag{4.2}$$

where $N_\delta = N_c N_y$ (the total number of target variables in the training data), and:

$$E_u = -\left(\frac{\alpha_u+N_u}{2}-1\right)\log(\tau_u) + \frac{\tau_u}{2}\left[\frac{\alpha_u}{\omega_u} + \sum_{i=1}^{N_u} u_i^2\right] \tag{4.3}$$

and E_v, E_a and E_b are similarly defined.

The above is the new potential energy that hybrid Monte Carlo must use in its simulation. Accordingly, we now expand the state space to include the hyperparameters. We denote the position and momentum variables corresponding to the parameters and hyperparameters with the subscripts θ and γ respectively:

$$\begin{aligned} \mathbf{q} &= (\mathbf{q}_\theta, \mathbf{q}_\gamma) \\ \mathbf{p} &= (\mathbf{p}_\theta, \mathbf{p}_\gamma) \end{aligned} \tag{4.4}$$

However, some complications now arise due to the necessity for the leapfrog proposals to be symmetric. The main problem is that the hyperparameters at the beginning and at the end of a leapfrog trajectory are now different, so setting the parameter stepsizes based on the hyperparameters at the beginning does not lead to reversible dynamics, i.e., by reversing the momentum and following the leapfrog dynamics backwards from the finishing point, one does not arrive back at the starting point.

The solution is to first update just the parameters by one step using stepsizes based on the current value of the hyperparameters. This is the same dynamical update with stepsize selection as before, and due to Fact 3 of Section 2.3.2, it is reversible, as the hyperparameters have not changed. Then, we update the hyperparameters by one step using stepsizes based only on the newly-computed value of the parameters. This is also reversible as the parameters do not change over the step. These two updates comprise one step in the new leapfrog trajectory updating both parameters and hyperparameters. By repeating this l times, we obtain a leapfrog trajectory of length l that, by being reversible in each step, is fully reversible end to end.

Due to Fact 1, each step leaves $H(\mathbf{q},\mathbf{p}) = E(\mathbf{q}) + \sum p_i^2/2$ approximately constant, and so H is left approximately constant over the entire trajectory for small enough η. Also, phase space volume is conserved by each step due to Fact 2, and so it is conserved over the entire trajectory. Thus, we see that we have a trajectory that keeps H roughly constant, and is a valid Metropolis proposal due to reversibility and phase space volume conservation.

There is actually a slight complication: if we update first the parameters, then the hyperparameters, the reversed trajectory is the one that updates first the hyperparameters and then the parameters, which is not actually the one we are using. To overcome this problem, at the beginning of a trajectory, we choose with equal probability to update either the parameters or the hyperparameters first. Thus, a trajectory that goes from point A to point B is proposed with 50% probability, while one that goes in reverse from

B to A is proposed also with 50% probability, and so we have symmetric proposals. We summarize this algorithm in Algorithm 4. There is a different stepsize for each component of $\mathbf{q}$, including the expanded γ portion of the state, and we denote the set of stepsizes for the parameters as υ_θ and the stepsizes for the hyperparameters as υ_γ.

Algorithm 4 Leapfrog trajectory that updates both parameters and hyperparameters using Hamiltonian dynamics

---

$r \leftarrow U[0,1]$
if $r < 0.5$ **then**
  for $i = 1$ to l **do**
    $\upsilon_\theta \leftarrow ParamStepsize(\mathbf{q}_\gamma)$
    $\{\mathbf{q}_\theta, \mathbf{p}_\theta\} \leftarrow LeapfrogUpdate(\mathbf{q}_\theta, \mathbf{p}_\theta; \upsilon_\theta, \mathbf{q}_\gamma)$
    $\upsilon_\gamma \leftarrow HyperparamStepsize(\mathbf{q}_\theta)$
    $\{\mathbf{q}_\gamma, \mathbf{p}_\gamma\} \leftarrow LeapfrogUpdate(\mathbf{q}_\gamma, \mathbf{p}_\gamma; \upsilon_\gamma, \mathbf{q}_\theta)$
  end for
else
  for $i = 1$ to l **do**
    $\upsilon_\gamma \leftarrow HyperparamStepsize(\mathbf{q}_\theta)$
    $\{\mathbf{q}_\gamma, \mathbf{p}_\gamma\} \leftarrow LeapfrogUpdate(\mathbf{q}_\gamma, \mathbf{p}_\gamma; \upsilon_\gamma, \mathbf{q}_\theta)$
    $\upsilon_\theta \leftarrow ParamStepsize(\mathbf{q}_\gamma)$
    $\{\mathbf{q}_\theta, \mathbf{p}_\theta\} \leftarrow LeapfrogUpdate(\mathbf{q}_\theta, \mathbf{p}_\theta; \upsilon_\theta, \mathbf{q}_\gamma)$
  end for
end if

An alternative way of achieving reversibility is to always start and end a leapfrog trajectory with either a parameter update or with a hyperparameter update. The exact method used should not significantly affect the performance of the algorithm.

4.2 Reparameterization of the Hyperparameters

Numerically, it is inconvenient to work with the hyperparameters as precisions, as negative values of precisions are invalid. Thus, we reparameterize the hyperparameters by working with log precisions instead:

$$\begin{aligned} \lambda_\delta &= \log(\tau_\delta) \\ \lambda_u &= \log(\tau_u) \\ \lambda_v &= \log(\tau_v) \\ \lambda_a &= \log(\tau_a) \\ \lambda_b &= \log(\tau_b) \end{aligned} \tag{4.5}$$

We let the set β denote the reparameterized hyperparameters:

$$\beta = \{\lambda_\delta, \lambda_u, \lambda_v, \lambda_a, \lambda_b\} \tag{4.6}$$

The posterior probability density is changed by this reparameterization. The new density is obtained by multiplying by the appropriate Jacobian of each variable transformation in turn:

$$\begin{aligned} P(\theta, \beta|x, y) &= P(\theta, \gamma|x, y) \times \left|\frac{\partial \tau_\delta}{\partial \lambda_\delta}\right| \prod_{*=u,v,a,b} \left|\frac{\partial \tau_*}{\partial \lambda_*}\right| \\ &= P(\theta, \gamma|x, y) \times \exp\left(\lambda_\delta + \sum_{*=u,v,a,b} \lambda_*\right) \end{aligned} \tag{4.7}$$

And so the new potential energy is:

$$\begin{aligned} E(\theta, \beta) &= -\log P(\theta, \beta|x, y) \\ &= E(\theta, \gamma) - \lambda_\delta - \sum_{*=u,v,a,b} \lambda_* \\ &= \left(\sum_{*=u,v,a,b} E_*^\lambda\right) - \left(\frac{\alpha_\delta + N_\delta}{2}\right)\lambda_\delta + \frac{e^{\lambda_\delta}}{2}\left[\frac{\alpha_\delta}{\omega_\delta} + \sum_{c=1}^{N_c} |\mathbf{f}(\mathbf{x}^c; \theta) - \mathbf{y}^c|^2\right] \end{aligned} \tag{4.8}$$

where:

$$E_u^\lambda = -\left(\frac{\alpha_u + N_u}{2}\right)\lambda_u + \frac{e^{\lambda_u}}{2}\left[\frac{\alpha_u}{\omega_u} + \sum_{i=1}^{N_u} u_i^2\right] \tag{4.9}$$

and E_v^λ, E_a^λ and E_b^λ are similarly defined.

4.3 Reparameterization of the Weights

The current parameterization scheme has a weakness that can be seen by considering the potential energy as a function of the weights U and their hyperparameter λ_u. In the absence of data, the potential energy depends on U and λ_u simply through E_u^λ, and we see that it is shallow and broad for low values of λ_u and narrow and deep for higher values (but not too high). This is because, for fixed λ_u, E_u^λ is quadratic in u_i with width proportional to $1/\sqrt{e^{\lambda_u}}$. This makes sense as $1/\sqrt{e^{\lambda_u}} = 1/\sqrt{\tau_u}$ is the prior standard deviation of u_i. When there is data, the landscape will be changed somewhat, but the tendencies imposed by the priors will still be there.

The effect of this shape of the potential energy function is to make it unlikely for a sample that starts in the broad, shallow region to end up in the narrow, deep region. The reason is because, since H is (approximately) conserved during a leapfrog trajectory, a particle that enters the narrow, deep region from the shallow region has enough energy to escape out to the shallow region again, and will indeed likely do so before we catch it in the deep region, since the latter has a comparatively small volume. Similarly, a particle that starts off in a narrow, deep region will likely not have enough total energy to escape unless it acquired an unusually large amount of energy during momentum resampling.

This situation is suboptimal as it increases autocorrelations. To move around more easily in state space, we introduce the following reparameterization of the weights:

$$\begin{aligned} \tilde{u}_i &= u_i\sqrt{\tau_u} = u_i e^{\lambda_u/2} \\ \tilde{v}_i &= v_i\sqrt{\tau_v} = v_i e^{\lambda_v/2} \\ \tilde{a}_i &= a_i\sqrt{\tau_a} = a_i e^{\lambda_a/2} \\ \tilde{b}_i &= b_i\sqrt{\tau_b} = b_i e^{\lambda_b/2} \end{aligned} \tag{4.10}$$

and we continue to use:

$$\begin{aligned} &\lambda_\delta = \log(\tau_\delta) \\ &\lambda_* = \log(\tau_*) \quad \text{where } * = u, v, a, b \end{aligned} \tag{4.11}$$

The notation of Chapter 3 will continue to apply, except we will use a tilde to indicate a reparameterized parameter. For instance, $\tilde{U}$ will represent the group of reparameterized input-to-hidden weights $\{u_i\}_{i=1}^{N_u}$, and $\tilde{U}_{ij}$ will represent a reparameterized weight from input unit i to hidden unit j. We will also use the following naming for all the reparameterized weights:

$$\phi = \{\tilde{U}, \tilde{V}, \tilde{A}, \tilde{B}\} \tag{4.12}$$

Due to the reparameterization, the posterior probability of the parameters and the hyperparameters must change accordingly. The complete reparameterization is obtained from Eqn. 4.7 as:

$$\begin{aligned} P(\phi, \beta|x, y) &= P(\theta, \beta|x, y) \times \left| \prod_{i=1}^{N_u} \frac{\partial u_i}{\partial \tilde{u}_i} \prod_{i=1}^{N_v} \frac{\partial v_i}{\partial \tilde{v}_i} \prod_{i=1}^{N_a} \frac{\partial a_i}{\partial \tilde{a}_i} \prod_{i=1}^{N_b} \frac{\partial b_i}{\partial \tilde{b}_i} \right| \\ &= P(\theta, \beta|x, y) \times \exp\left(-\frac{1}{2} \sum_{*=u,v,a,b} N_* \lambda_* \right) \end{aligned} \tag{4.13}$$

And so under the reparameterization, the potential energy becomes:

$$\begin{aligned} E(\phi, \beta) &= -\log P(\phi, \beta|x, y) \\ &= E(\theta, \gamma) - \lambda_\delta - \sum_{*=u,v,a,b} \lambda_* + \frac{1}{2}\left(\sum_{*=u,v,a,b} N_* \lambda_* \right) \\ &= \left(\sum_{*=u,v,a,b} E^{\lambda}_{\tilde{*}} \right) + \frac{1}{2}\left[-(N_\delta + \alpha_\delta)\lambda_\delta + \frac{\alpha_\delta}{\omega_\delta} e^{\lambda_\delta} + e^{\lambda_\delta} \sum_{c=1}^{N_c} |\mathbf{f}(\mathbf{x}^c; \phi, \beta) - \mathbf{y}^c|^2 \right] \end{aligned} \tag{4.14}$$

where:

$$E^{\lambda}_{\tilde{u}} = \frac{1}{2}\left(-\alpha_u \lambda_u + \frac{\alpha_u}{\omega_u} e^{\lambda_u} + \sum_{i=1}^{N_u} \tilde{u}_i^2 \right) \tag{4.15}$$

and similarly for $E^{\lambda}_{\tilde{v}}$, $E^{\lambda}_{\tilde{a}}$ and $E^{\lambda}_{\tilde{b}}$.

It can be seen that the the quadratic term $\tilde{u}_i^2$ in $E^{\lambda}_{\tilde{u}}$ now has a constant coefficient, so this reparameterization is effective at removing the variation with its hyperparameter of the width of u_i's potential bowl.

To distinguish between the new methods with and without the reparameterization of the weights, we will call the new method before the weight reparameterization the Dynamical A method, and the new method with the weight reparameterization the Dynamical B method.

4.4 First Derivatives of the Potential Energy

Each leapfrog step update requires the first derivatives of the Hamiltonian with respect to the parameters and the hyperparameters. To take steps that are appropriately scaled in the various directions for stability and efficiency, we need the second derivatives as well, which we give in Section 4.5. Here, we give the expressions for the first derivatives.

We first define:

$$L^c(\phi, \beta) = e^{\lambda_\delta} |\mathbf{f}(\mathbf{x}^c; \phi, \beta) - \mathbf{y}^c|^2/2 \tag{4.16}$$

which is the negative log likelihood of one training case, less the normalizing term. We then obtain from Eqn. 4.14:

$$\frac{\partial E(\phi, \beta)}{\partial \lambda_\delta} = \frac{1}{2}\left[-(N_\delta + \alpha_\delta) + \frac{\alpha_\delta}{\omega_\delta} e^{\lambda_\delta}\right] + \sum_{c=1}^{N_c} L^c(\phi, \beta) \tag{4.17}$$

$$\frac{\partial E(\phi, \beta)}{\partial \lambda_*} = \frac{1}{2}\left(-\alpha_* + \frac{\alpha_*}{\omega_*} e^{\lambda_*}\right) + \sum_{c=1}^{N_c} \frac{\partial L^c(\phi, \beta)}{\partial \lambda_*} \tag{4.18}$$

where, as usual, $* = u, v, a$ or b, and:

$$\frac{\partial E(\phi, \beta)}{\partial \tilde{U}_{ij}} = \tilde{U}_{ij} + \sum_{c=1}^{N_c} \frac{\partial L^c(\phi, \beta)}{\partial \tilde{U}_{ij}} \tag{4.19}$$

Derivatives for the other weight types are obtained from Eqn. 4.19 by replacing $\tilde{U}_{ij}$ by the corresponding parameter.

To compute the derivative $\partial E/\partial \lambda_\delta$ using Eqn. 4.17, we need to compute the network outputs for every training case. This involves performing a forward pass through the net for each training case, each pass requiring compute time of order the number of connections in the network. The fact that we do a forward pass means that using backpropagation to compute other first derivatives will be efficient. We now explain how these other derivatives are computed.

4.4.1 First Derivatives with Respect to Parameters

To compute the derivatives with respect to the parameters (Eqn. 4.19), we need to find the corresponding derivatives of the output L^c. They are most efficiently computed using backpropagation provided certain results are stored during a preceding forward pass such as the one required by the above computation of $\partial E/\partial\lambda_\delta$.

The backpropagation works as follows. Consider Figure 4.1, which represents two arbitrary adjacent layers in the network.

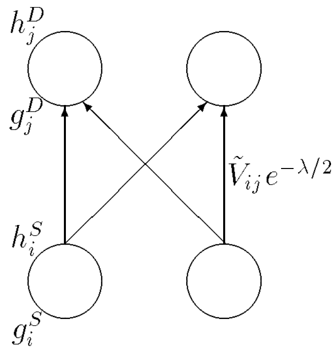


Figure 4.1: Two adjacent layers.

Here, $\tilde{V}_{ij}e^{-\lambda/2}$ is the weight connecting a source unit i to a destination unit j. We use g_i to denote the total activation of unit i before the $\tanh(\cdot)$ nonlinearity, and h_i to denote the output of unit i after the nonlinearity. Source unit values are denoted by superscript S, while destination unit values have superscript D. The total input into destination unit j is:

$$g_j^D = \sum_i h_i^S \tilde{V}_{ij} e^{-\lambda/2} + \tilde{b}_j e^{-\lambda'/2} \tag{4.20}$$

where λ' is the hyperparameter controlling the biases $\tilde{b}_j$. Thus, the first derivative of the potential energy with respect to $\tilde{V}_{ij}$ can be computed as follows:

$$\begin{aligned}\frac{\partial L^c(\phi,\beta)}{\partial \tilde{V}_{ij}} &= \frac{\partial L^c(\phi,\beta)}{\partial g_j^D}\frac{\partial g_j^D}{\partial \tilde{V}_{ij}} \\ &= \frac{\partial L^c(\phi,\beta)}{\partial g_j^D} h_i^S e^{-\lambda/2}\end{aligned} \tag{4.21}$$

A similar expression holds for derivatives with respect to biases. In the above, if j is actually an output unit, then $g_j^D = f_j$, so:

$$\frac{\partial L^c(\phi,\beta)}{\partial g_j^D} = e^{\lambda_\delta}[f_j(\mathbf{x}^c;\phi,\beta) - y_j^c] \tag{4.22}$$

The outputs $f_j(\mathbf{x}^c;\phi,\beta)$ can once again be considered to have already been obtained "free" from the forward pass. As in standard backpropagation, we start with the above error derivatives at the output layer and propagate them backwards using:

$$\begin{aligned}\frac{\partial L^c(\phi,\beta)}{\partial g_i^S} &= \sum_j \frac{\partial L^c(\phi,\beta)}{\partial g_j^D}\frac{\partial g_j^D}{\partial h_i^S}\frac{\partial h_i^S}{\partial g_i^S} \\ &= \operatorname{sech}^2(g_i^S)\sum_j \frac{\partial L^c(\phi,\beta)}{\partial g_j^D}\tilde{V}_{ij} e^{-\lambda/2}\end{aligned} \tag{4.23}$$

If g_i was stored for every unit during the forward pass, the above derivatives can be computed rapidly in a backward pass taking time of order the number of connections in the network for each training case. Actually, we will see below that, if we can save only one quantity, the most useful one is:

$$\mu_j = e^{-\lambda/2}\sum_i h_i^S \tilde{V}_{ij} \tag{4.24}$$

which is the total input going into unit j from all the units feeding into it. From μ_j, g_j can easily be obtained in constant time. Furthermore, it will be useful in other calculations that will be presented in the subsequent discussion.

Note that, thus far, we have seen how the computation of $\partial E/\partial \lambda_\delta$ and the first derivatives with respect to all the parameters is dominated by a forward pass and a backward pass through the network.

4.4.2 First Derivatives with Respect to the Hyperparameters

The computation of $\partial E/\partial\lambda_\delta$ has already been described. To compute $\partial E/\partial\lambda_*$, we need to compute the corresponding derivatives of $L^c(\phi,\beta)$. In a similar fashion as the weights' computation, we can write for the hyperparameter λ in Eqn. 4.20 that controls weights:

$$\begin{aligned}\frac{\partial L^c(\phi,\beta)}{\partial\lambda} &= \sum_j \frac{\partial L^c(\phi,\beta)}{\partial g_j^D}\frac{\partial g_j^D}{\partial\lambda} \\ &= -\frac{1}{2}\sum_j \frac{\partial L^c(\phi,\beta)}{\partial g_j^D}\left(\sum_i h_i^S\tilde{V}_{ij}\right)e^{-\lambda/2} \\ &= -\frac{1}{2}\sum_j \frac{\partial L^c(\phi,\beta)}{\partial g_j^D}\mu_j\end{aligned} \tag{4.25}$$

while for the hyperparameter λ':

$$\begin{aligned}\frac{\partial L^c(\phi,\beta)}{\partial\lambda'} &= \sum_j \frac{\partial L^c(\phi,\beta)}{\partial g_j^D}\frac{\partial g_j^D}{\partial\lambda} \\ &= -\frac{1}{2}\sum_j \frac{\partial L^c(\phi,\beta)}{\partial g_j^D}\tilde{b}_j e^{-\lambda'/2}\end{aligned} \tag{4.26}$$

We already computed the derivatives $\partial L^c/\partial g_j$ during the backward pass for the derivatives of the parameters. By ensuring that we save μ_j during the forward pass, the first derivatives of L^c with respect to the u, v, a and b hyperparameters can be efficiently computed in time of order the number of units in the network. Summed over all cases, the computational cost is $O(N_c(N_h+N_y))$;

Because the number of units is considerably smaller than the number of parameters, calculating these first derivatives with respect to the u, v, a and b hyperparameters is considered to add negligible cost to the forward and backward passes we have already done. Therefore, the computation of all the first derivatives is dominated by the forward and backward passes through the net, each of which takes $O(N_c|\phi|)$

4.5 Approximations to the Second Derivatives of the Potential Energy

The second derivative of the potential energy with respect to the weights and the hyperparameters are needed to compute stepsizes for each leapfrog step update. Unlike the first derivatives, we cannot compute the second derivatives exactly because, in order to preserve phase space conservation and reversibility of leapfrog steps, the stepsizes used for, say, hyperparameters, cannot depend on the current values of the hyperparameters. We will also use additional simplifications to make evaluation easier and faster.

From Eqn. 4.14, we see that the problem is to obtain for the parameters:

$$\frac{\partial^2 E(\phi,\beta)}{\partial \tilde{u}_i^2} = 1 + \sum_{c=1}^{N_c} \frac{\partial^2 L^c(\phi,\beta)}{\partial \tilde{u}_i^2} \tag{4.27}$$

and similarly for $\tilde{v}_i, \tilde{a}_i$ and $\tilde{b}_i$; and for the hyperparameters:

$$\frac{\partial^2 E(\phi,\beta)}{\partial \lambda_\delta^2} = \frac{\alpha_\delta}{2\omega_\delta} e^{\lambda_\delta} + \sum_{c=1}^{N_c} L^c(\phi,\beta) \tag{4.28}$$

and for $*$ taking the values u, v, a and b:

$$\frac{\partial^2 E(\phi,\beta)}{\partial \lambda_*^2} = \frac{\alpha_*}{2\omega_*} e^{\lambda_*} + \sum_{c=1}^{N_c} \frac{\partial^2 L^c(\phi,\beta)}{\partial \lambda_*^2} \tag{4.29}$$

4.5.1 Second Derivatives with Respect to the Parameters

To obtain the derivative $\partial^2 L^c / \partial \tilde{u}_i^2$, we follow the heuristic given by Neal (1996, Appendix A), which we include here for completeness. The heuristic operates by approximately backpropagating the 2nd derivative of L^c with respect to the output units back through the net. Its details are as follows.

Referring to Fig. 4.1, Neal uses the following approximation:

$$\begin{aligned}\frac{\partial^2 L^c(\phi,\beta)}{\partial \tilde{V}_{ij}^2} &\approx \frac{\partial^2 L^c(\phi,\beta)}{\partial (g_j^D)^2}\left(\frac{\partial g_j^D}{\partial \tilde{V}_{ij}}\right)^2 \\ &\approx \frac{\partial^2 L^c(\phi,\beta)}{\partial (g_j^D)^2} e^{-\lambda} \times \begin{cases} (x_i^c)^2 & \text{for } i \text{ an input unit,} \\ 1 & \text{otherwise.}\end{cases}\end{aligned} \tag{4.30}$$

Correspondingly, for biases:

$$\frac{\partial^2 L^c(\phi,\beta)}{\partial \tilde{b}_j^2} \approx \frac{\partial^2 L^c(\phi,\beta)}{\partial (g_j^D)^2} e^{-\lambda'} \tag{4.31}$$

We see that it is necessary to compute the derivatives $\partial^2 L^c/\partial(g_j^D)^2$. We will do so in a way analogous to the backpropagation of the first derivative $\partial L^c/\partial(g_j^D)$ as described in Section 4.4.1. In the case that unit j is an output unit, the derivative is fixed, namely:

$$\frac{\partial^2 L^c(\phi,\beta)}{\partial (g_j^D)^2} = \frac{\partial^2 L^c(\phi,\beta)}{\partial (f_j^c)^2} = e^{\lambda_\delta} \tag{4.32}$$

This is propagated backwards to obtain the second derivatives of L^c with respect to all the inputs g_i's except for the input units', whose derivatives are not needed. Neal propagates the derivatives using:

$$\frac{\partial^2 L^c(\phi,\beta)}{\partial (g_i^S)^2} \approx \sum_j \frac{\partial^2 L^c(\phi,\beta)}{\partial \tilde{(g_j^D)}^2} \tilde{V}_{ij}^2 e^{-\lambda} \tag{4.33}$$

Because we are not allowed to use the current value of the parameters, we replace $\tilde{V}_{ij}^2$ in the above by the estimate 1, since $\tilde{V}_{ij}^2$ has variance 1 at equilibrium. Thus, we actually use:

$$\frac{\partial^2 L^c(\phi,\beta)}{\partial (g_i^S)^2} \approx \sum_j \frac{\partial^2 L^c(\phi,\beta)}{\partial \tilde{(g_j^D)}^2} e^{-\lambda} \tag{4.34}$$

From Eqns. 4.32 and 4.34, we see that the second derivatives are the same for all training cases. Thus, the backpropagation pass is done only once regardless of the number of training cases, after which the second derivatives with respect to the parameters can be estimated in time of order equal to the number of parameters.

4.5.2 Second Derivatives with Respect to the Hyperparameters

To obtain $\sum_{c=1}^{N_c} L^c$ in Eqn. 4.28, we need to calculate the network output for each training case. Like the computation for $\partial E/\partial\lambda_\delta$, this can be done with a forward pass for each case. Unlike that computation, we cannot use the current values of the hyperparameters. In this thesis, we replace β by $\hat{\beta}$, the prior hyperparameter means:

$$\begin{aligned}
\hat{\lambda}_\delta &= \log \omega_\delta \\
\hat{\lambda}_u &= \log \omega_u \\
\hat{\lambda}_v &= \log \omega_v \\
\hat{\lambda}_a &= \log \omega_a \\
\hat{\lambda}_b &= \log \omega_b
\end{aligned} \tag{4.35}$$

Thus, we really compute:

$$\frac{\partial^2 E(\phi,\beta)}{\partial\lambda_\delta^2} \approx \frac{\alpha_\delta}{2\omega_\delta} e^{\hat{\lambda}_\delta} + \sum_{c=1}^{N_c} L^c(\phi,\hat{\beta}) \tag{4.36}$$

For the second derivatives with respect to the other hyperparameters, we similarly replace all occurrences of β by $\hat{\beta}$. From Eqn. 4.29, we see that we need to estimate $\partial^2 L^c(\phi,\hat{\beta})/\partial\lambda_*^2$. For this, we will use the same kind of backpropagation as when estimating the second derivatives with respect to the parameters.

Once again referring to Fig.4.1, for λ being either a hyperparameter for the weights or the biases that contribute to the calculation of the g_j^D's:

$$\begin{aligned}
\frac{\partial^2 L^c}{\partial\lambda^2} &= \frac{\partial}{\partial\lambda}\left(\frac{\partial L^c}{\partial\lambda}\right) \\
&= \frac{\partial}{\partial\lambda}\sum_j \frac{\partial L^c}{\partial g_j^D}\frac{\partial g_j^D}{\partial\lambda} \\
&= \sum_j \left[\frac{\partial L^c}{\partial g_j^D}\frac{\partial^2 g_j^D}{\partial\lambda^2} + \frac{\partial^2 L^c}{\partial\lambda\partial g_j^D}\frac{\partial g_j^D}{\partial\lambda}\right] \\
&= \sum_j \frac{\partial L^c}{\partial g_j^D}\left(-\frac{1}{2}\frac{\partial g_j^D}{\partial\lambda}\right) + \sum_j\sum_{j'} \frac{\partial^2 L^c}{\partial g_{j'}^D\partial g_j^D}\frac{\partial g_{j'}^D}{\partial\lambda}\frac{\partial g_j^D}{\partial\lambda}
\end{aligned} \tag{4.37}$$

where we have made the replacement $\partial^2 g_j^D/\partial\lambda^2 = -(1/2)\partial g_j^D/\partial\lambda$, which can easily be checked. Thus:

$$\frac{\partial^2 L^c}{\partial\lambda^2} = -\frac{1}{2}\frac{\partial L^c}{\partial\lambda} + \sum_j \sum_{j'} \frac{\partial^2 L^c}{\partial g_{j'}^D \partial g_j^D} \frac{\partial g_{j'}^D}{\partial\lambda} \frac{\partial g_j^D}{\partial\lambda} \tag{4.38}$$

For the second term, following Neal (1996, Appendix A), we ignore multiple connections from the same unit, which amounts to dropping all cross terms. We end up with:

$$\frac{\partial^2 L^c}{\partial\lambda^2} \approx -\frac{1}{2}\frac{\partial L^c}{\partial\lambda} + \sum_j \frac{\partial^2 L^c}{\partial(g_j^D)^2}\left(\frac{\partial g_j^D}{\partial\lambda}\right)^2 \tag{4.39}$$

We have already seen how we can estimate $\partial^2 L^c/\partial(g_j^D)^2$ using the approximate backpropagation (Eqn. 4.34) of Section 4.5.1. The difference here is that we can use the actual values of the parameters, but not the hyperparameters. Thus, we replace λ by its estimate $\hat{\lambda} \in \hat{\beta}$ in the backpropagation equation:

$$\frac{\partial^2 L^c(\phi,\beta)}{\partial(g_i^S)^2} \approx \sum_j \frac{\partial^2 L^c(\phi,\beta)}{\partial(g_j^D)^2}\tilde{V}_{ij}^2 e^{-\hat{\lambda}} \tag{4.40}$$

As before, we compute the above quantity in a backward pass only down to the first hidden layer, and these derivatives are all independent of the training case.

For the second factor in a summation term in Eqn. 4.39, evaluation is straightforward. From Eqn. 4.20, we have:

$$\left(\frac{\partial g_j^D}{\partial\lambda'}\right)^2 = \frac{e^{-\hat{\lambda}'}}{4} b_j^2 \tag{4.41}$$

for a hyperparameter controlling biases, and:

$$\begin{aligned}\left(\frac{\partial g_j^D}{\partial\lambda}\right)^2 &= \frac{e^{-\hat{\lambda}}}{4}\left(\sum_{i=1}^n \tilde{V}_{ij} h_i^S\right)^2 \\ &= \frac{1}{4}\mu_j^2\end{aligned} \tag{4.42}$$

for a hyperparameter controlling weights. Once again, if we save μ_j during the forward pass to obtain the network outputs for estimating $\partial^2 E/\partial\lambda_\delta^2$, the above factors can be obtained almost for free. We emphasize here that the this μ_j differs from the μ_j used in first derivative calculations in that the forward pass during which μ_j is saved uses the estimates $\hat{\beta}$ instead of β.

Thus far, the dominating computation in estimating the second derivative of E with respect to the hyperparameters is the estimation of $\partial^2 E/\partial\lambda_\delta^2$, which requires a forward pass for every training case.

Let us now turn our attention to the first derivative $\partial L^c/\partial\lambda$ in Eqn. 4.39. Its computation was already discussed in Section 4.4.2, except that it uses the estimates $\hat{\beta}$ instead of β, and the error propagated backwards comes from the forward pass used in estimating $\partial^2 E/\partial\lambda_\delta^2$.

The estimation of this derivative requires one backward pass, which must be done for each training case. Thus, combined with the forward passes of $\partial^2 E/\partial\lambda_\delta^2$, two passes through the net are necessary to estimate the second derivatives of E with respect to the hyperparameters.

4.6 Summary of Compute Times

In this section, we summarize the compute time required by the Dynamical B method per leapfrog update of both the parameters and hyperparameters. Recall that the Dynamical B method includes the reparameterization of the weights.

First, we summarize the compute times required to calculate each group of derivatives in Table 4.1. The compute times are dominated by passes through the network for each training case, which takes time $O(N_c|\phi|)$, and we consider other operations as essentially free.

Examining a leapfrog update in detail, we see that it looks like Table 4.2.

When the parameters are changed (step 2), the first derivatives of both parameters and hyperparameters need to be recalculated in order to update the momenta in steps

Group of Derivatives	Compute time
1st with respect to parameters and hyperparameters	$2 \times O(N_c\|\phi\|)$
2nd with respect to parameters	"free"
2nd with respect to hyperparameters	$2 \times O(N_c\|\phi\|)$

Table 4.1: Cost of computing various groups of derivatives for the Dynamical B method.

Step	Description
1	Update momentum of parameters
2	Update parameters
3	Update momentum of parameters
4	Update momentum of hyperparameters
5	Update hyperparameters
6	Update momentum of hyperparameters
	(Step 1 is then repeated for the next leapfrog update)

Table 4.2: The steps in one complete leapfrog update in the Dynamical B method. The Dynamical A method has the same sequence of steps comprising one leapfrog update.

3 and 4. The cost is one forward-backward pass pair. Also, at the end of step 2, the second derivatives with respect to the hyperparameters need to be recalculated for use in steps 4 through 6, taking a second forward-backward pass pair. After the update of the hyperparameters at step 5, the first derivatives need to be recalculated again for the momentum updates at step 6 and step 1 of the next complete leapfrog update. This requires a third forward-backward pass pair. Finally, the second derivatives with respect to the parameters also need to be recalculated for use in step 1 of the next iteration, but this is essentially free.

Thus, each leapfrog update costs 3 forward-backward pass pairs. This result will be used later in determining how long to let the parameterized new method run when comparing its performance to the old method.

4.7 Computation of Stepsizes

We have described our heuristic for approximating the second derivative of the potential energy with respect to the hyperparameters. The stepsize is then computed as the inverse square root of that second derivative, as in Eqn. 2.21. But because this heuristic uses Eqn. 4.39, second derivatives have the possibility of being negative, and square roots would then be imaginary. We note that when the second derivative becomes negative, it merely indicates that the potential energy surface is now concave downwards, but its magnitude should still be indicative of the length scale of the surface variations in the region. Thus, the negative sign is really no problem, and we take absolute values to obtain the stepsize as:

$$\epsilon = \eta \left| \frac{\partial^2 E}{\partial \lambda^2} \right|^{-\frac{1}{2}} \tag{4.43}$$

Similarly, the stepsize for the parameters are obtained as:

$$\epsilon = \eta \left(\frac{\partial^2 E}{\partial \tilde{u}_i^2} \right)^{-\frac{1}{2}} \tag{4.44}$$

and similarly for the other parameter types.

4.8 Compute Times for the Dynamical A Method

We also give the compute time for one leapfrog update for the Dynamical A method as this has to be taken into account later in performance comparison. Recall that the Dynamical A method is the new method before the weight reparameterization.

The computation of the first and second derivatives with respect to the weights in this scheme are not significantly different from Dynamical B's. The algorithmically demanding portions of these computations are the derivatives of L^c with respect to a weight, and the backpropagation algorithms given above work the same way except that the factors of $e^{-\lambda/2}$ that always go with the reparameterized weights are missing. Thus, first derivatives with respect to the weights also require one forward and one backward pass for each training case, while second derivatives are essentially free.

The computation of the first derivatives and second derivatives with respect to the hyperparameters differ substantially, however, because the hyperparameters do not appear in the computation of the network output $\mathbf{f}(\cdot)$ in this case. The first derivatives are:

$$\frac{\partial E(\theta,\beta)}{\partial \lambda_\delta} = -\left(\frac{\alpha_\delta + N_\delta}{2}\right) + \frac{e^{\lambda_\delta}}{2}\left[\frac{\alpha_\delta}{\omega_\delta} + \sum_{c=1}^{N_c}\left(\mathbf{f}(\mathbf{x}^c;\theta) - \mathbf{y}^c\right)^2\right] \tag{4.45}$$

and:

$$\frac{\partial E(\theta,\beta)}{\partial \lambda_u} = -\left(\frac{\alpha_u + N_u}{2}\right) + \frac{e^{\lambda_u}}{2}\left[\frac{\alpha_u}{\omega_u} + \sum_{i=1}^{N_u} u_i^2\right] \tag{4.46}$$

and similarly for hyperparameters of v, a and b.

The compute times for hyperparameters of first derivatives of λ_*, where $* = u, v, a$ and b take time of order the number of parameters and is independent of the number of training cases. This makes it of lower order time complexity than that the forward and backward passes for computing the first derivatives of the parameters. The computation of $\partial E/\partial \lambda_\delta$ requires all the network outputs for each training case, but which were already computed during the forward passes for the first derivatives with respect to the parameters, so we essentially get this derivative for free as well.

The second derivatives with respect to the hyperparameters are:

$$\frac{\partial^2 E(\theta,\beta)}{\partial \lambda_\delta^2} = \frac{e^{\lambda_\delta}}{2}\left[\frac{\alpha_\delta}{\omega_\delta} + \sum_{c=1}^{N_c}\left(\mathbf{f}(\mathbf{x}^c;\theta) - \mathbf{y}^c\right)^2\right] \tag{4.47}$$

and:

$$\frac{\partial^2 E(\theta,\beta)}{\partial \lambda_u^2} = \frac{e^{\lambda_u}}{2}\left[\frac{\alpha_u}{\omega_u} + \sum_{i=1}^{N_u} u_i^2\right] \tag{4.48}$$

and similarly for hyperparameters of v, a and b.

The compute times for each second derivative is essentially the same as that for the first derivative with respect to the same hyperparameter. Once again, we can use the network outputs already computed for the first derivatives with respect to the parameters. Note that this differs from the case of the reparameterized weights because, there, the hyperparameters are involved in computing the network outputs, and so the the outputs computed during the forward passes for the first derivatives cannot be used, as the second derivatives with respect to the hyperparameters cannot use the current values of the hyperparameters. That forced us to redo the passes through the network with estimates for the hyperparameters, but we do not have to do that here, thereby saving computation.

We summarize the various compute costs in the Table 4.8

Group of Derivatives	Compute time
1st with respect to parameters and hyperparameters	$2 \times O(N_c|\phi|)$
2nd with respect to parameters	"free"
2nd with respect to hyperparameters	"free"

Table 4.3: Cost of computing various groups of derivatives for the Dynamical A method.

Like before, one leapfrog update requires the computation of all the first derivatives twice. Therefore, one leapfrog update requires two forward-backward pass pairs in this case.

Chapter 5

Results

5.1 Training Data

To verify the new methods and compare their performance with the old method, the synthetic data set of Table 5.1 was used. We use a small data set to reduce the compute time required to obtain the results for this thesis. Even then, months passed before all the necessary runs were completed.

Input x	Input y	Output z
5.8964216e-01	7.5148380e-01	1.9992522e+00
9.1368690e-01	4.7461198e-01	2.4043475e+00
4.5180698e-02	-7.2696252e-01	9.3298996e-01
7.6028441e-01	-9.7648663e-01	5.4781275e-01
-6.5408772e-01	7.8779593e-01	1.9858707e-01
9.5949379e-01	-6.0172387e-01	4.6687881e-01
-4.5710548e-01	-4.0255398e-01	6.4682705e-03
-4.9534131e-01	3.2288515e-01	-1.7153228e-01

Table 5.1: Training data has two inputs and 1 output. These data are plotted in Fig. 5.1.

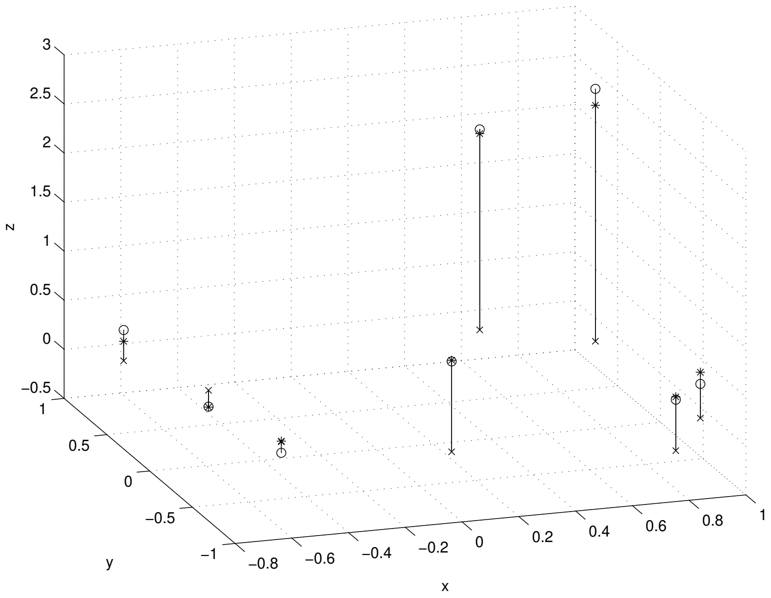


Figure 5.1: 8 points comprising the synthetic data set used. There are 2 inputs (x and y) and 1 target (z). The training data are shown using asterisks. The circles represent the training data before the addition of Gaussian noise, while the crosses are the input data drawn on the plane $z = 0$.

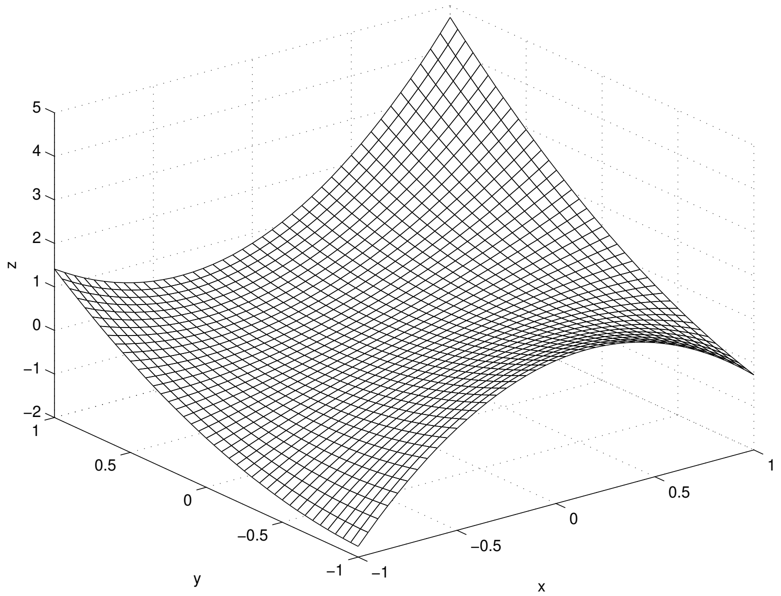


Figure 5.2: The surface from which the synthetic data was taken.

The training data was synthesized as follows. The inputs (x_i, y_i) were uniformly drawn from $[-1,1] \times [-1,1]$. The mapping used on each input pair was calculated using the function below:

$$f(x,y) = 0.3 + 1.2x + 0.7(y-0.2)^2 + 2.3y(x+0.1)^2 \tag{5.1}$$

This mapping is illustrated in Fig. 5.2. Gaussian noise with standard deviation 0.1 was added to each function output to obtain the target.

5.2 Verification of the New Methods

To verify the correctness of the new methods, the posterior distribution was obtained using all the methods for a network of 8 hidden units on the above training data. The old Gibbs sampling program as implemented by Neal was treated as the standard against which the new programs were compared.

The priors for the hyperparameters are specified as gamma distributions as in Eqn. 3.7 using α and ω parameters. For the demonstration of the correctness of the new methods, they were set as in Table 5.2.

Parameter	Setting	Parameter	Setting
$\omega_\delta^{-\frac{1}{2}}$	0.10	α_δ	0.50
$\omega_u^{-\frac{1}{2}}$	1.00	α_u	0.50
$\omega_v^{-\frac{1}{2}}$	0.45	α_v	0.50
$\omega_a^{-\frac{1}{2}}$	1.00	α_a	0.50
$\omega_b^{-\frac{1}{2}}$	0.80	α_b	0.50

Table 5.2: Settings for parameters specifying the priors of the hyperparameters

Fairly long runs were done with all three methods at the settings in Table 5.3.

5.2.1 Results From Old Method

Method	η	l	Saved every
Gibbs	0.35	400	100
Dynamical A	0.48	100	10
Dynamical B	0.08	300	10

Table 5.3: Settings for the various methods in order to verify correctness of new programs. For the new methods, we set the parameter and hyperparameter stepsize adjustment factors equal to each other, and indicate their value by η in this table.

The output surface as predicted from 5000 samples obtained using the Gibbs update method is shown in Fig. 5.3. We see that the data points are being fitted reasonably. The fact that the points are being fitted implies that the posterior distribution has changed from the prior. Indeed, from Fig. 5.4, we see that the marginal posterior distributions of the hyperparameters differ from the marginal prior distributions.

We show in Fig. 5.5 the correlation between the input-to-hidden and the hidden bias hyperparameters. As expected, when larger input-to-hidden weights are allowed, larger biases (with the opposite sign) are required to compensate. This is because the output function cannot be composed of hidden units that all saturate, so the input into at least some of the hidden units must be kept small.

Also, we show in Fig. 5.6 how, as the number of hidden units increases, the hidden bias hyperparameter becomes more pinned to the actual standard deviation of the hidden biases, and vice versa. This is manifested as an increased correlation between them. This is a direct demonstration of the problem we set out to solve.

5.2.2 Results of New Methods Compared with the Old

To verify the correctness of the programs implementing the new methods, we compare the posterior distributions obtained using the new programs against that from the old.

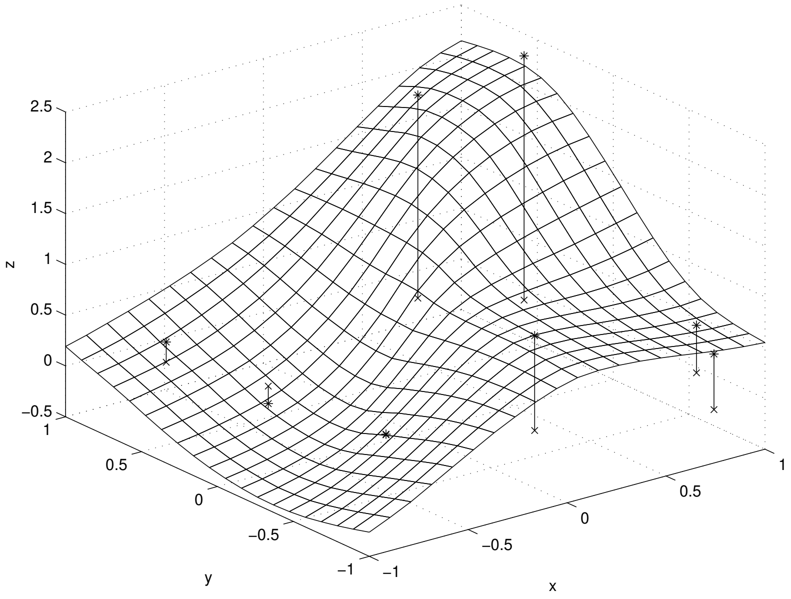


Figure 5.3: The surface predicted by the samples obtained for the master run for 8 hidden units. The training data are shown using asterisks, while the crosses are the input data drawn on the plane $z = 0$.

To obtain the posterior distribution, any samples near the beginning found by visual inspection not to be in equilibrium were first dropped. 950 samples from each method were then used to plot the histogram of each hyperparameter. Here, we look at the histogram of $\log \sigma_*$, which is the log of each hyperparameter specified as a standard deviation. Each such histogram approximates the marginal distribution of a hyperparameter.

We need to be able to compare the joint distribution over hyperparameters for two methods. Because we are unable to plot a distribution over the joint 5-dimensional space of all the hyperparameters, we compare marginal distributions instead. This comparison is valid because, if the marginal distributions of the hyperparameters match for two methods, then they almost certainly have the same joint distribution over hyperparameters. While it is true that, in principle, equal marginal distributions does not imply equal joint distributions, the fact that the marginals match is too amazing a coincidence to be explained any other way than by concluding that the joint distributions match.

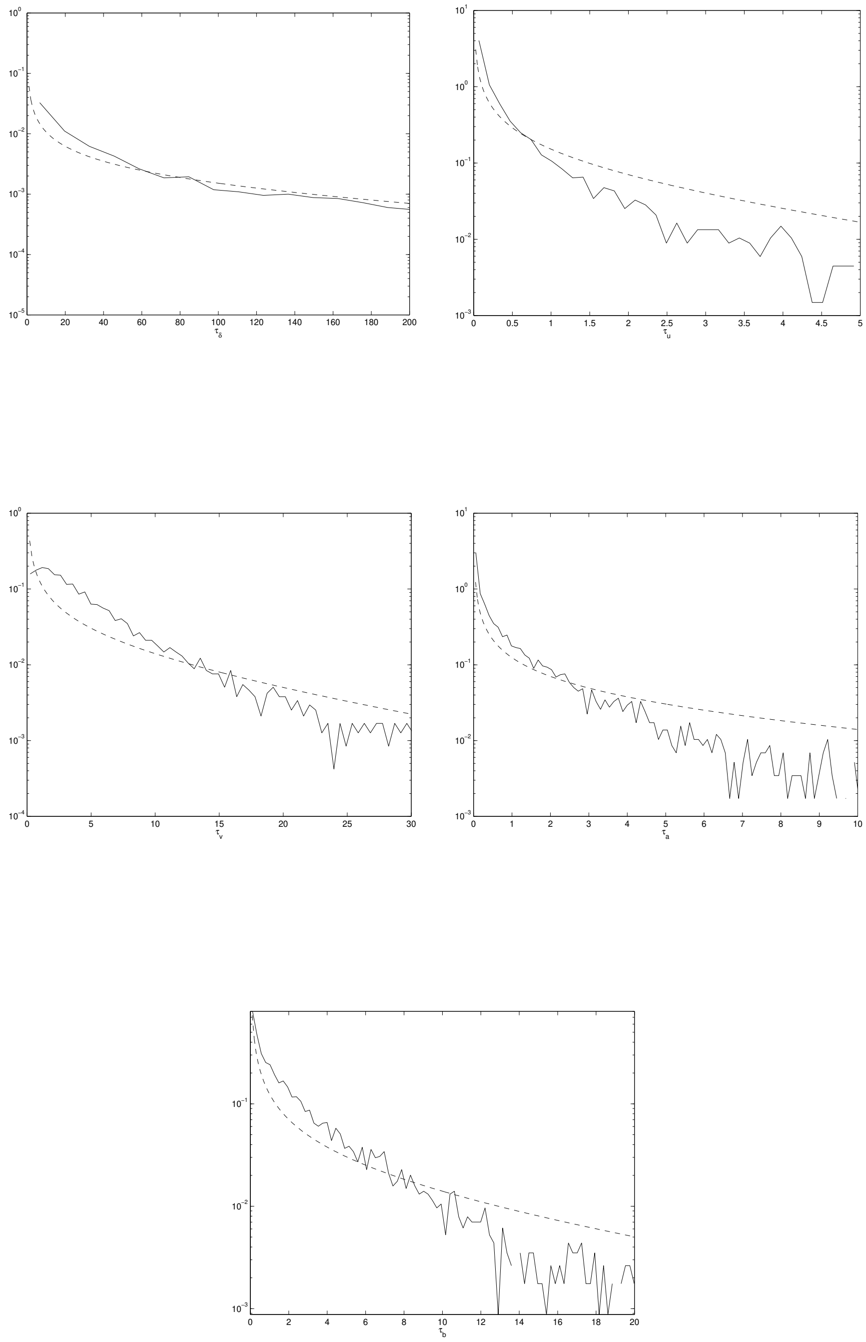


Figure 5.4: Dotted lines represent prior densities (obtained analytically) while solid lines represent posterior marginal distributions obtained from the Gibbs sampling method. These plots, obtained using 8 hidden units, show that the posterior distributions of the hyperparameters have changed from the priors.

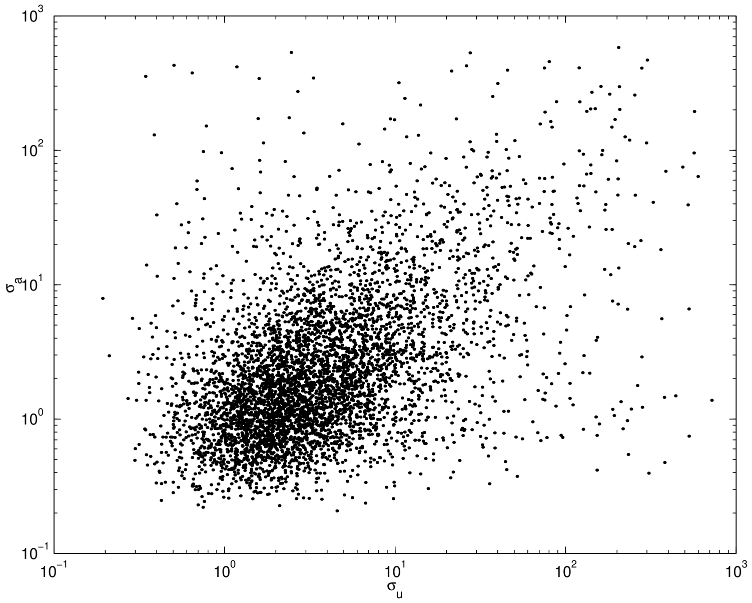
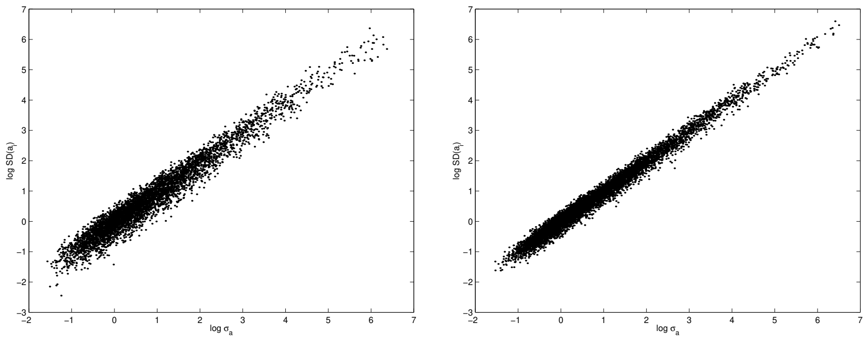


Figure 5.5: Correlation between input-to-hidden and hidden bias hyperparameters obtained using the old method.



(a) 8 hidden units

(b) 20 hidden units

Figure 5.6: Correlation between the standard deviation of the hidden biases and the their hyperparameter becomes stronger as the hidden layer size increases.

As can be seen in Figs. 5.8 and 5.7, the marginals for all the hyperparameters obtained by the programs running both the new methods match those of the original as implemented by Neal.

5.3 Methodology for Evaluating Performance

Having shown that the new methods have been correctly implemented, we are now ready to assess their performance.

A Markov chain Monte Carlo method typically goes through a "burn in" phase before settling down to equilibrium. Before reaching equilibrium, its samples are not representative of its invariant distribution. As these non-representative samples should be discarded in order not to skew later estimates, the speed with which a Markov chain equilibrates is a matter of interest. However, due to time constraints, we will not be considering this question. Instead, we will only assess the relative performances of the old and the new methods in moving about in the posterior distributions of the hyperparameters once equilibrium has been reached. We do not consider the speed with which the posterior of the parameters is explored as there can be large numbers of parameters, and it is difficult to know which parameters to compare as they can sometimes taken on different roles.

In addition to examining the performances of the original method and the Dynamical B method, we also look at the performance of the Dynamical A method to ascertain if the reparameterization of the weights is indeed beneficial.

The performance of the methods depends on the number of leapfrog steps allowed in one trajectory and the stepsize adjustment factor. These can be viewed as tuning parameters that affect the efficiency of each method. Since performance can vary dramatically depending on the setting of these tuning parameters, it is only fair to compare how well the methods work when optimally tuned.

For the old method, the tuning parameters are l, the number of leapfrog steps allowed

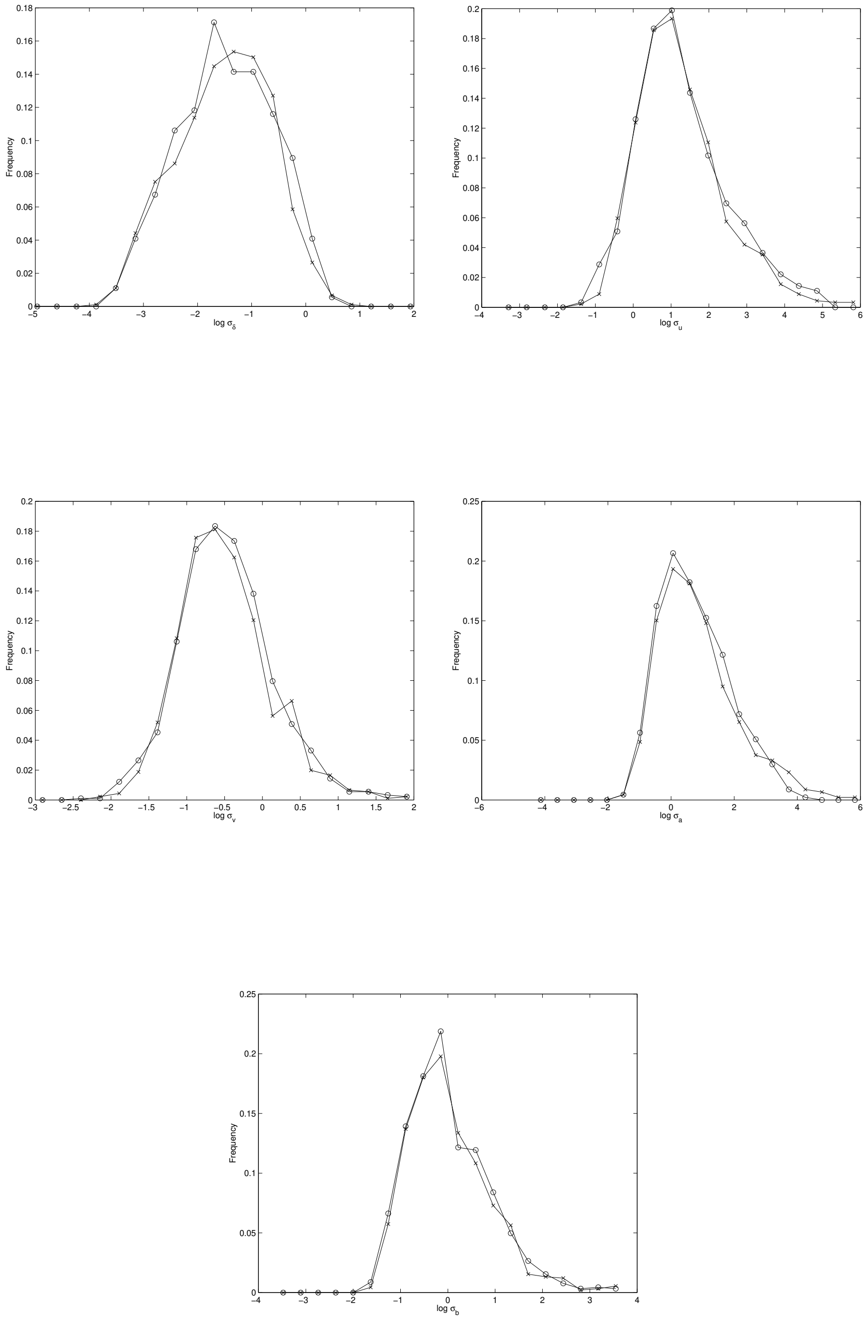


Figure 5.7: Crosses represent the distribution obtained from the original Gibbs update method, while circles represent that of the Dynamical A update method. Each distribution has been normalized to have area 1.

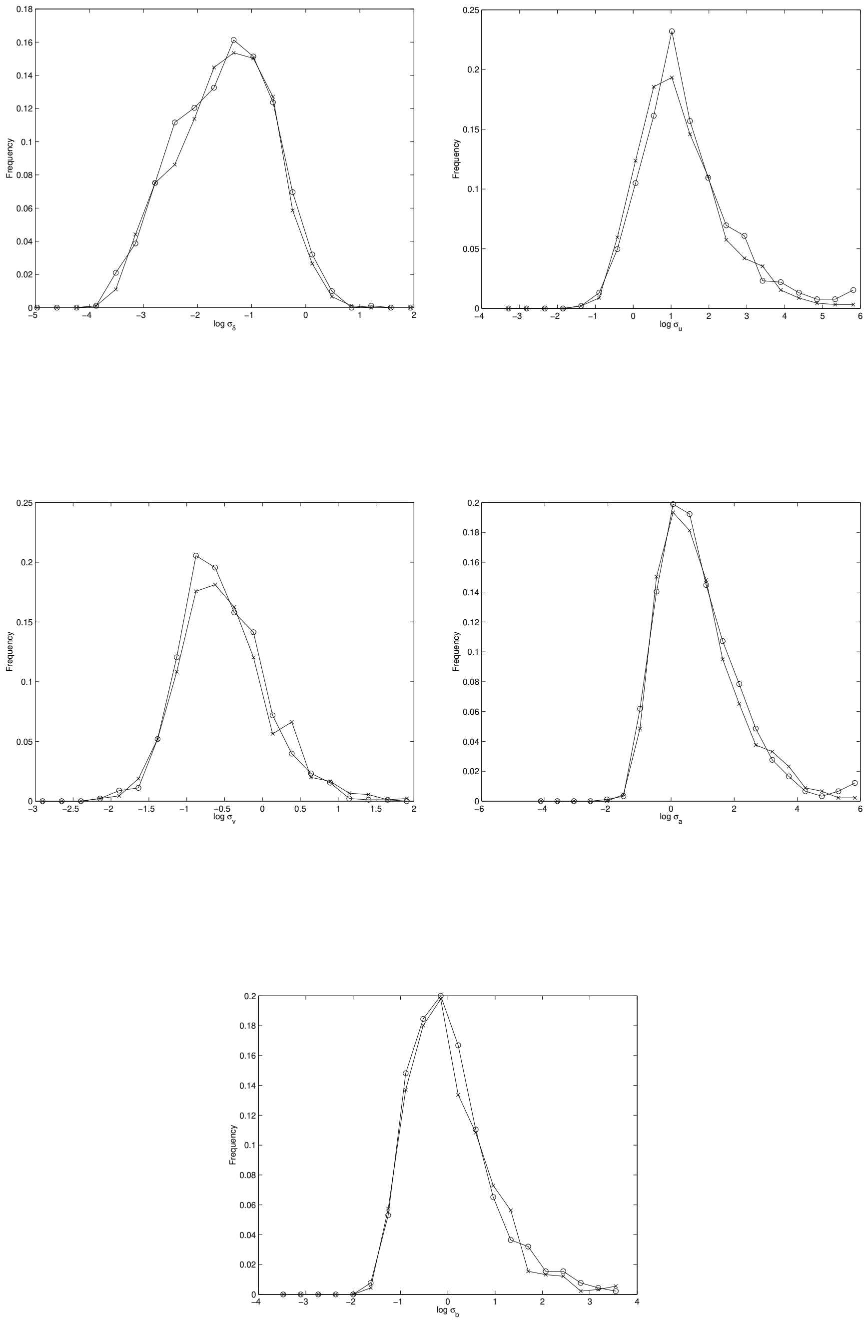


Figure 5.8: Crosses represent the distribution obtained from the original Gibbs update method, while circles represent that of the Dynamical B update method. Each distribution has been normalized to have area 1. Note that the distribution for the original method looks slightly different from that of the plots for the previous comparison with the Dynamical A method because the binning for the histograms is slightly different.

in one trajectory, and η, the stepsize adjustment factor. For the new methods, l is also a tuning parameter, but we now have two stepsize adjustment factors: η_p for the parameters, and η_h for the hyperparameters. We have two stepsize adjustment factors because we might wish to control how fast the parameters move compared to the hyperparameters in order to obtain the best performance. Moreover, the different heuristics with which the stepsizes are computed for the parameters and the hyperparameters means that their relative magnitudes might be quite different, which also suggests separate stepsize adjustment factors. However, due to time constraints, we will not explore the problem of how to set the two adjustment factors separately, and will instead set them equal to each other ($\eta_p = \eta_h$) and call it η, with the understanding that the performance of the new methods could be increased if the two η's are not set equal to each other.

The performace of a method on a hyperparameter is assessed using the variance of means measure, whose presentation we delay till the next section. For a given setting of the tuning parameters, this measure can be computed for each hyperparameter. The smaller the measure is, the more efficiently the method explores the marginal posterior distribution of that hyperparameter. Since we wish to compare the methods when they are operating optimally, we will compare their variance of means measures at optimal settings of l and η.

To find the optimal setting of l and η for a given method, we run it over a grid of settings in tuning parameter space, measuring its variance of means performance at each setting. The geometric mean of the variance of means of the hyperparameters is computed to obtain a single measure that combines the performances over the different hyperparameters. In computing the geometric mean, we leave out the variance of means measure for the output bias hyperparameter as that hyperparameter controls only one parameter for this network and is therefore not that meaningful. The optimal setting of the tuning parameters is then picked as the setting that minimizes the geometric mean.

It is not safe to directly use the variance of means at this optimal setting to compare the performances of the methods, however, as they are biased downwards as a result of this selection process. Instead, the programs are then re-run with different random seeds to re-obtain the variance of means measures so as to avoid the bias. In the next section, we describe the variance of means measure in greater detail.

5.3.1 The Variance of Means Measurement of Performance

The variance of means measure characterizes how well a method explores the posterior distribution of a hyperparameter at a fixed setting of the tuning parameters. To obtain this measure, we run several Markov chains, each started from an independent point drawn from the posterior distribution of the parameters and hyperparameters. Each chain is run for a fixed number of N_f leapfrog steps regardless of what l is.

For a given setting of the tuning parameters, we obtain the means of the hyperparameters sampled by each chain. The entire chain of N_f leapfrog steps can be thought of as being divided up into a fixed number of N_s super-transitions each comprised of N_f/N_s leapfrog steps. Although each leapfrog trajectory yields a sample from the correct distribution, we use only one sample per super-transition to compute the means of the hyperparameters for each chain. Thus, regardless of the value of l, the same number of samples N_s is used to compute the means as shown in Table 5.4. For the i'th chain, the hyperparameter means obtained are:

$$(\overline{\log \sigma_\delta^i}, \overline{\log \sigma_u^i}, \overline{\log \sigma_v^i}, \overline{\log \sigma_a^i}, \overline{\log \sigma_b^i}) \tag{5.2}$$

where, for instance, each mean $\overline{\log \sigma_u^i}$ is computed as follows:

$$\overline{\log \sigma_u^i} = \sum_{j=1}^{N_s} \log \sigma_u^{i,j} / N_s \tag{5.3}$$

σ refers to a hyperparameter expressed as a standard deviation. We take the log before computing the mean as experience shows that the standard deviation can vary over several orders of magnitude, and yet variations on a small scale are as interesting as variation on a scale a few orders of magnitude larger.

Chain	Samples						
	$\sigma_\delta^{1,1}$	$\sigma_\delta^{1,2}$	$\sigma_\delta^{1,3}$	$\ldots\ \ldots$	σ_δ^{1,N_s}	$\rightarrow$	$\overline{\log \sigma_\delta^1}$
	$\sigma_u^{1,1}$	$\sigma_u^{1,2}$	$\sigma_u^{1,3}$	$\ldots\ \ldots$	σ_u^{1,N_s}	$\rightarrow$	$\overline{\log \sigma_u^1}$
1	$\sigma_v^{1,1}$	$\sigma_v^{1,2}$	$\sigma_v^{1,3}$	$\ldots\ \ldots$	σ_v^{1,N_s}	$\rightarrow$	$\overline{\log \sigma_v^1}$
	$\sigma_a^{1,1}$	$\sigma_a^{1,2}$	$\sigma_a^{1,3}$	$\ldots\ \ldots$	σ_a^{1,N_s}	$\rightarrow$	$\overline{\log \sigma_a^1}$
	$\sigma_b^{1,1}$	$\sigma_b^{1,2}$	$\sigma_b^{1,3}$	$\ldots\ \ldots$	σ_b^{1,N_s}	$\rightarrow$	$\overline{\log \sigma_b^1}$
	$\sigma_\delta^{2,1}$	$\sigma_\delta^{2,2}$	$\sigma_\delta^{2,3}$	$\ldots\ \ldots$	σ_δ^{2,N_s}	$\rightarrow$	$\overline{\log \sigma_\delta^2}$
	$\sigma_u^{2,1}$	$\sigma_u^{2,2}$	$\sigma_u^{2,3}$	$\ldots\ \ldots$	σ_u^{2,N_s}	$\rightarrow$	$\overline{\log \sigma_u^2}$
2	$\sigma_v^{2,1}$	$\sigma_v^{2,2}$	$\sigma_v^{2,3}$	$\ldots\ \ldots$	σ_v^{2,N_s}	$\rightarrow$	$\overline{\log \sigma_v^2}$
	$\sigma_a^{2,1}$	$\sigma_a^{2,2}$	$\sigma_a^{2,3}$	$\ldots\ \ldots$	σ_a^{2,N_s}	$\rightarrow$	$\overline{\log \sigma_a^2}$
	$\sigma_b^{2,1}$	$\sigma_b^{2,2}$	$\sigma_b^{2,3}$	$\ldots\ \ldots$	σ_b^{2,N_s}	$\rightarrow$	$\overline{\log \sigma_b^2}$
$\vdots$				$\vdots$			
$\vdots$				$\vdots$			
	$\sigma_\delta^{N_m,1}$	$\sigma_\delta^{N_m,2}$	$\sigma_\delta^{N_m,3}$	$\ldots\ \ldots$	$\sigma_\delta^{N_m,N_s}$	$\rightarrow$	$\overline{\log \sigma_\delta^{N_m}}$
	$\sigma_u^{N_m,1}$	$\sigma_u^{N_m,2}$	$\sigma_u^{N_m,3}$	$\ldots\ \ldots$	$\sigma_u^{N_m,N_s}$	$\rightarrow$	$\overline{\log \sigma_u^{N_m}}$
N_m	$\sigma_v^{N_m,1}$	$\sigma_v^{N_m,2}$	$\sigma_v^{N_m,3}$	$\ldots\ \ldots$	$\sigma_v^{N_m,N_s}$	$\rightarrow$	$\overline{\log \sigma_v^{N_m}}$
	$\sigma_a^{N_m,1}$	$\sigma_a^{N_m,2}$	$\sigma_a^{N_m,3}$	$\ldots\ \ldots$	$\sigma_a^{N_m,N_s}$	$\rightarrow$	$\overline{\log \sigma_a^{N_m}}$
	$\sigma_b^{N_m,1}$	$\sigma_b^{N_m,2}$	$\sigma_b^{N_m,3}$	$\ldots\ \ldots$	$\sigma_b^{N_m,N_s}$	$\rightarrow$	$\overline{\log \sigma_b^{N_m}}$

Table 5.4: Each chain is run for N_s super-transitions at some setting of l and η. The samples obtained from the super-transitions are used to compute the means of each hyperparameter.

If we run N_m Markov chains, we will have N_m values of $\overline{\log \sigma_*}$. The variance of means measure that we have been talking about is then just the variance of these values of $\overline{\log \sigma_*}$. We define ρ_* to be these variance of means measures:

$$\begin{aligned}\rho_\delta &= \mathrm{Var}[\overline{\log \sigma_\delta}] \\ \rho_u &= \mathrm{Var}[\overline{\log \sigma_u}] \\ \rho_v &= \mathrm{Var}[\overline{\log \sigma_v}] \\ \rho_a &= \mathrm{Var}[\overline{\log \sigma_a}] \\ \rho_b &= \mathrm{Var}[\overline{\log \sigma_b}]\end{aligned} \tag{5.4}$$

Each variance is calculated using the mean estimated from a very long run of the old method, which we assume to be very close to the true mean. For example, if the mean for $\log \sigma_u$ obtained from a very long run of the old method is $< \log \sigma_u >$, then we compute the variance from N_m Markov chains as:

$$\rho_u = \frac{\sum_{i=1}^{N_m}(\overline{\log \sigma_u^i} - < \log \sigma_u >)^2}{N_m} \tag{5.5}$$

This is the variance of means measure of performance, with lower variance indicating better performance.

Let T_u be the inefficiency factor in units of super-transitions for N_s samples of $\log \sigma_u$. T_u measures the worth in computing $\overline{\log \sigma_u}$ of each super-transition of $\log \sigma_u$ relative to one independent sample of $\log \sigma_u$. For example, if T_u is 2, then it takes twice as many super-transitions to obtain a given variance of $\overline{\log \sigma_u}$ as would be needed using independent samples. Mathematically, the variance of means measures are related to inefficiency factors as follows:

$$\rho_u = \frac{T_u}{N_s}\mathrm{Var}[\log \sigma_u] \tag{5.6}$$

Since $\mathrm{Var}[\log \sigma_u]$ is a constant property of the posterior, and we are using the same N_s for all tuning parameter settings, our estimate of $\mathrm{Var}[\log \sigma_u]$ is proportional to T_u. Although this T_u is for this particular number of samples N_s only, we would expect that, if a method has a lower T_u than another for this N_s, then it really is more efficient at exploring the posterior distribution, and so it should remain better for a different N_s. Thus, this measure is indicative of performance in general.

5.3.2 Error Estimation for Variance of Means

Error bars on the variance of the means are obtained as follows. Since the variance of ρ_u is calculated as the mean of the square deviations $(\overline{\log \sigma_u^i} - < \log \sigma_u >)^2$, its variance is given by:

$$\mathrm{Var}[\rho_u] = \frac{\mathrm{Var}[(\overline{\log \sigma_u^i} - < \log \sigma_u >)^2]}{N_m} \tag{5.7}$$

which is valid so long as the chains are independent. We ensure this by picking their starting points sufficiently far apart from a long master run and by using a different random number seed for each chain.

5.3.3 Geometric Mean of Variance of Means

Rather than characterizing performance by the variance of the hyperparameter means for all the hyperparameters, we define the following single geometric mean scalar measure of performance:

$$g = \left(\rho_\delta \rho_u \rho_v \rho_a\right)^{\frac{1}{4}} \tag{5.8}$$

We obtain error bars for g by bootstrapping (see Efron and Tibshirani, 1993) as follows. Let Z be the original set of N_m Markov chains. Thus, Z determines a single value of g. A bootstrap realization Z^* is a set of N_m Markov chains sampled uniformly with replacement from the original chains. During bootstrapping, many bootstrap realizations Z^* are generated from Z. The idea is that the empirical distribution represented by Z contains within it the natural variations that g has over the true distribution from which Z is drawn. So, a value of g can be calculated from each Z^*, and the histogram of these resulting g's gives an estimate for the actual distribution of g. We quote error bars as the 90% confidence interval of the histogram of g, i.e., we quote the error bar $[g_{lo}, g_{hi}]$, where g_{lo} and g_{hi} are such that 5% of the values of g obtained from Z^* are below g_{lo}, and 5% are above g_{hi}. We use 1000 bootstrap samples to obtain these error bars.

5.3.4 Iterations Allowed for Each Method

As shown in Eqn. 5.6, ρ_* is proportional to the inefficiency factor of the method in units of super-transitions. In order for comparisons of variances between two methods to make sense, the amount of compute time used per super-transition should be the same in both cases. However, the compute time of a program is tricky to calculate from things such as number of multiplications, as programming style can affect it. Thus, in this thesis, the amount of "work" that goes into a super-transition is estimated in algorithmic terms in which we assume that the number of training cases N_c is large, and the number of hidden units N_h is also large so that the number of weights $|\phi|$ is large. This results in terms of order $N_c|\phi|$ dominating the time complexity, which comes entirely from complete sweeps through the neural network for each training case.

In the old method, each trajectory is composed of 3 steps: sampling the hyperparameters, computing the stepsizes, and performing the leapfrog steps. First, we note that each leapfrog step requires the evalutaion of the derivative of the potential energy with respect to a parameter:

$$\frac{\partial E}{\partial u_i} = \tau_\delta \sum_{c=1}^{N_c} [\mathbf{f}(\mathbf{x}^c;\theta) - \mathbf{y}^c] \cdot \frac{\partial \mathbf{f}(\mathbf{x}^c;\theta)}{\partial u_i} + \tau_u u_i \tag{5.9}$$

which is most efficiently computed for each training case using a forward sweep through the network for $\mathbf{f}(\mathbf{x}^c;\theta)-\mathbf{y}^c$ and a backward sweep for $\partial\mathbf{f}(\mathbf{x}^c;\theta)/\partial u_i$. Each of the 2 sweeps takes $O(N_c|\phi|)$ compute time for all the training cases. By comparison, the sampling of the hyperparameters is dominated by the computation of the sum of the square of the weights, as can be seen in Eqn. 3.11, and by the computation of the total squared error of the network output (Eqn. 3.12). The former takes $O(|\phi|)$, while the latter costs only $O(N_c)$ since the network error has already been computed at the end of the last leapfrog step for the Metropolis rejection test. So, both can be neglected when compared to the compute time required for a leapfrog step. The computation of the stepsizes is also free by comparison because its summation over training cases (Neal, 1996, Appendix A) can be factored out and computed at the very beginning of the program. Thus, each super-transition in the old method is approximately dominated by the leapfrog steps, each of which takes 1 pair of forward-backward sweeps, each of which takes $O(N_c|\phi|)$ compute time. We summarize this along with the compute times for the new methods in Table 5.5.

Hyperparameter updates by	Network passes per leapfrog step
Gibbs	2
Dynamical A	4
Dynamical B	6

Table 5.5: For a detailed explanation of the number of passes required for the two dynamical methods, please refer to Section 4.8.

Because of the results in Table 5.5, the Dynamical B method is only allowed a third as many leapfrog steps per super-transition as the old method, while the Dynamical A method is allowed half. These measures ensure that each super-transition uses approximately the same amount of compute time regardless of method.

5.4 Markov chain start states

5.4.1 Master Runs

As mentioned earlier, each chain is started from a state chosen from equilibrium. To obtain the starting points for a particular network, a very long run was done using the old method, typically resulting in several hundred thousand to a million or more samples. Any initial portion of the run not in equilibrium was discarded, and starting points were then obtained from the remaining samples. These starting states were spaced many inefficiency factors apart so that they are likely to be nearly independent points that represent the posterior distribution well.

The long run was done with a relatively small stepsize adjustment factor so that the rejection rate is low (around 5%). This is because chains using large stepsize adjustment factors may sometimes be unable to enter certain regions of state space where its rejection rate is high. The reason for this is because, once it enters, it is likely to stay there for a long time due to its high rejection rate. Because it remains stuck there for a long time, it follows that it is unlikely to enter that region in the first place. Thus, using a low rejection rate reduces the risk of overlooking such regions.

Long runs were obtained for networks with 8, 12, 16 and 20 hidden units. Table 5.6 shows the prior settings used for these networks, while table 5.7 tabulates the various run settings used, the number of samples obtained, and the resulting rejection rate.

Hidden units	$\omega_\delta^{-\frac{1}{2}}$	$\omega_u^{-\frac{1}{2}}$	$\omega_v^{-\frac{1}{2}}$	$\omega_a^{-\frac{1}{2}}$	$\omega_b^{-\frac{1}{2}}$	α_δ	α_u	α_v	α_a	α_b
8	0.10	1.00	0.45	1.00	0.80	0.50	0.50	0.50	0.50	0.50
12	0.10	1.00	0.37	1.00	0.80	0.50	0.50	0.50	0.50	0.50
16	0.10	1.00	0.32	1.00	0.80	0.50	0.50	0.50	0.50	0.50
20	0.10	1.00	0.285	1.00	0.80	0.50	0.50	0.50	0.50	0.50

Table 5.6: Prior settings for the neural net architectures tested.

In order to ensure that our start states are from the equilibrium distribution, the hyperparameters from the master runs were visually checked for the absence of long term trends.

Hidden units	l	η	No. of samples	Rejection rate
8	400	0.35	500000	6.16%
12	400	0.32	800000	7.22%
16	400	0.26	3500000	5.71%
20	400	0.24	700000	5.95%

Table 5.7: Tuning parameter settings, number of samples collected and rejection rates. Rejection rates were chosen to be relatively low to reduce the risk of not being able to enter regions where the rejection rates are high. For all the runs, only one sample was saved for every 100 samples collected.

5.4.2 Starting States Used

Variance of means measures were obtained for networks with 8, 12, 16 and 20 hidden units. This requires several Markov chain starting states for each number of hidden units. This section shows in detail how these states were obtained.

Table 5.8 shows the inefficiency factors of $\log \sigma_u$, as that was found to be the slowest moving hyperparameter, i.e., it had the highest inefficiency factor. We use σ_* to denote $\tau_*^{-\frac{1}{2}}$. It is the hyperparameter expressed as a standard deviation and is often easier to understand. We compute the inefficiency factor of $\log \sigma_*$ rather than simply σ_* as the inefficiency factor is then the same regardless of what power σ_* is raised to, thus removing any questions as to whether it is more appropriate to measure the inefficiency factor of the variance or the standard deviation.

As each chain used in the variance of means measure should be started from an independent point, we use samples from the master run spaced many inefficiency factors apart.

All the variance of means measures were computed using 50 chains. The start states of the first 40 chains are evenly spaced according to Table 5.8. It was decided to add 10 more chains because it was empirically observed that the different methods seem to do very differently on chains starting at large $\log \sigma_u$ and $\log \sigma_a$, large meaning greater than 4 in this case. Specifically, it appeared from the first 40 chains that the old method and the Dynamical A method do badly on chains starting at large values of these hyperparameters, whereas the Dynamical B method does well. Because such large hyperparameter values appear rarely in the posterior distribution, there are few of them among the 40 chains, so assuming that they do have an important effect on the results, it is necessary to oversample the region with large values of $\log \sigma_u$ and $\log \sigma_a$ and then downweight those points accordingly in the computation of the variance of the means. Otherwise, one might by chance compute the variance of means without any chain starting at large values of those hyperparameters, and erroneously conclude that all the methods perform similarly.

Hidden units	T_u	Start state separation (samples)	Start state separation (T_u's)	Initial samples dropped
8	320	10000	31	0
12	470	10000	21	0
16	2500	50000	20	0
20	770	10000	13	0

Table 5.8: First 40 starting states are drawn so that they are many inefficiency factors T_u apart. T_u is the inefficiency factor of $\log \sigma_u$.

To ensure that we have a decent number of points from the region G satisfying $\log \sigma_u > 4$ or $\log \sigma_a > 4$ or both, it was decided to stratify the starting points so that 40 of them are outside G and 10 are inside. Of the first 40 points chosen according to the separations in Table 5.8, some are already in G. So, the choice of points 41 through 50 was done as follows: for each number of hidden units, we chose enough additional points in G so as to make them total 10 in number, and we also chose enough points outside of G so as to obtain 40 of them.

To obtain these 10 new points, a long sequence of many points was obtained from the master file at fixed separations (always at least 3 inefficiency factors). For 12 and 16 hidden units, the 10 new points were obtained past the end of the portion of the master run used to obtain the first 40 starting states; for 8 and 20 hidden units, the new points used the portion already used for the first 40 states, and beyond if available, taking care to always maintain at least 3 inefficiency factors from the first 40 points. This sequence was then uniformly sampled to obtain the desired number of points in G and points outside of G.

5.4.3 Modified Performance Measures Due to Stratification

The stratification of the starting states does change the calculation of the variance of means measure and its error somewhat. For N_G equal to the number of points taken from region G and $N_{\overline{G}}$ the number of points taken from outside, the variance of means within each stratum for hyperparameter σ_u is:

$$\begin{aligned} \rho_{u,\overline{G}} &= \frac{1}{N_{\overline{G}}} \sum_{i=1}^{N_{\overline{G}}} (\overline{\log \sigma_u^i} - < \log \sigma_u >)^2 \\ \rho_{u,G} &= \frac{1}{N_G} \sum_{i=N_{\overline{G}}+1}^{N_m} (\overline{\log \sigma_u^i} - < \log \sigma_u >)^2 \end{aligned} \tag{5.10}$$

so that the overall variance of means that takes stratification into account is:

$$\rho_u = (1-p)\rho_{u,\overline{G}} + p\rho_{u,G} \tag{5.11}$$

where p is the fraction of the posterior distribution in region G.

For error estimates of the variance of the means, we compute the variance of the above as before, which is similarly computed as the weighted sum of the error variances computed separately for each stratification:

$$\mathrm{Var}[\rho_u] = (1-p)^2 \mathrm{Var}[\rho_{u,\overline{G}}] + p^2 \mathrm{Var}[\rho_{u,G}] \tag{5.12}$$

Hidden units	p
8	4.98%
12	5.29%
16	7.05%
20	4.91%

Table 5.9: Estimated fraction of the posterior distribution in region G for each number of hidden units. Region G is defined as the region for which $\log \sigma_u > 4$ or $\log \sigma_a > 4$ or both.

The above expressions correct for the oversampling of G using p. Table 5.9 gives estimates of these quantities from each master run.

It should be noted that we have assumed that p is known in the calculation of the error bars of the variance of means measures; but really, all we have are estimates. This simplification introduces some inaccuracies into the error calculations, but it is not likely to change our conclusions much, as we will see later.

One might question why the various values of p in Table 5.9 are quite different. The author has some empirical experience showing that G is a region of somewhat higher rejection rate than normal (around 12% for 20 hidden units). It is possible that some of the master runs have η's that are high enough that they enter G rarely enough to make a difference in the estimates of p. This aspect of the experiment is difficult to control, as it is usually not possible to tell in advance what the regions with high rejection rates are going to be, and if they will make a difference to the final variance of means estimates in the end. Furthermore, if such regions are identified after obtaining master runs, it can be very costly computationwise to redo the master runs at a smaller setting of η.

Finally, the bootstrap procedure takes the stratification into account as follows: the first 40 chains in Z^* are sampled uniformly with replacement from the first 40 chains in Z, while the last 10 chains in Z^* are sampled uniformly with replacement from the last

10 chains in Z. This ensures that each realization Z^* is obtained from the same empirical distribution represented by Z.

5.5 Number of Leapfrog Steps Allowed

In accordance with the deemed ratios of computation involved in each super-transition for the various methods (Table 5.5), different numbers of leapfrog iterations were used for each Markov chain. These are listed in Table 5.10.

Hyperparameter updates by	Leapfrog steps per chain
Gibbs	614400
Dynamical A	307200
Dynamical B	204804

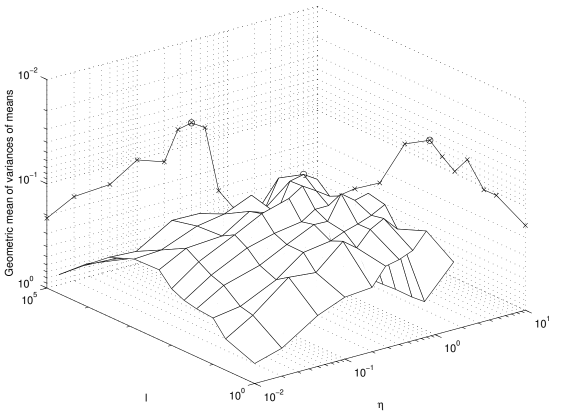
Table 5.10: The varying numbers of leapfrog steps that were allowed per Markov chain in accordance with the ratios of computation involved in each super-transition for the various methods as listed in Table 5.5

.

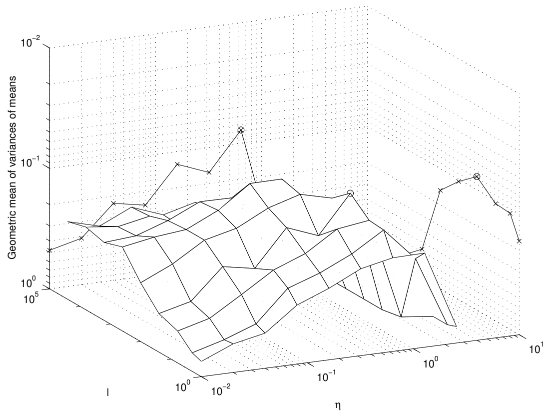
5.6 Results of Performance Evaluation

The optimal tuning parameter settings for each method and for each number of hidden units were assessed from Figs. 5.9 to 5.12. The optimal tuning parameters thus obtained are given in Table 5.11. The corresponding g's and variance of means and rejection rates obtained at these optimal settings but in new runs with new random seeds are given in Table 5.12.

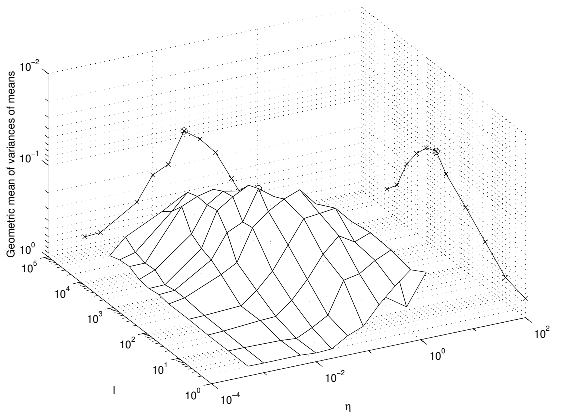
As can be seen from Table 5.11, the optimal setting of η for the Gibbs and Dynamical A methods tends to drecrease with increasing hidden layer size. Dynamical B, interestingly, increases with increasing hidden layer size.



(a) Gibbs update

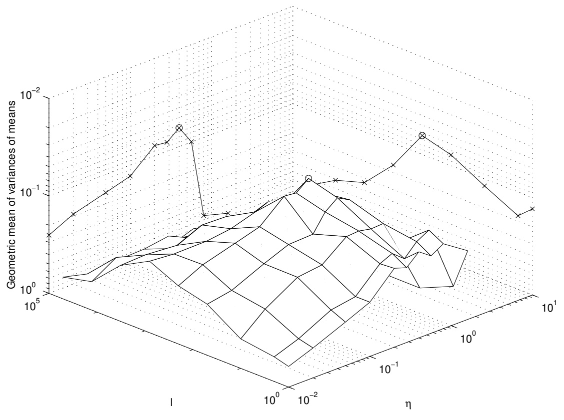


(b) Dynamical A update

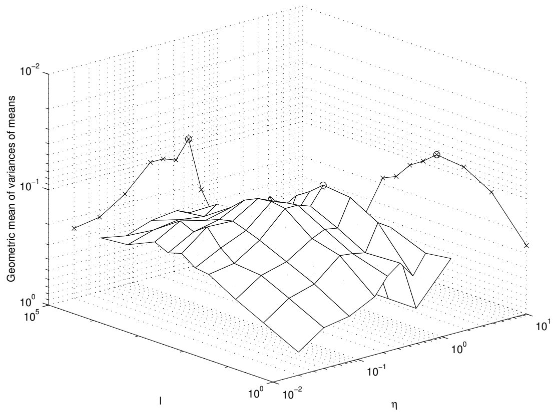


(c) Dynamical B update

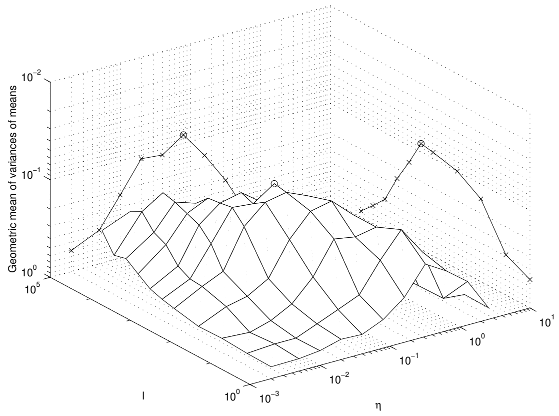
Figure 5.9: 8 hidden units: a circle is drawn at the optimum point, and the surface as a function of l and η are backprojected on to the walls at optimal η and l respectively. Note that the vertical scale is inverted.



(a) Gibbs update

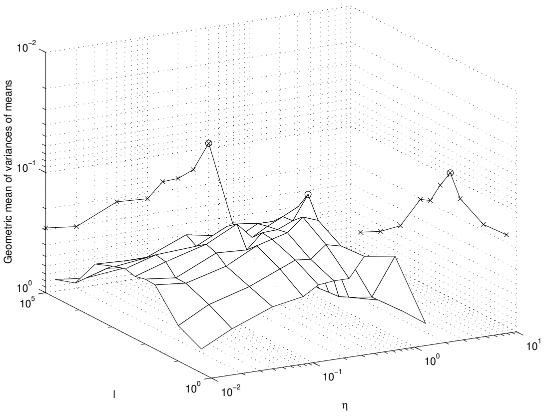


(b) Dynamical A update

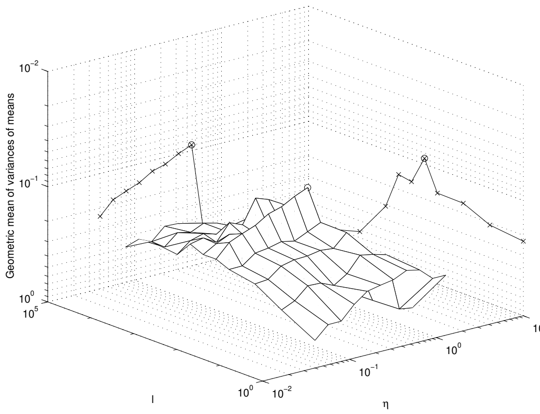


(c) Dynamical B update

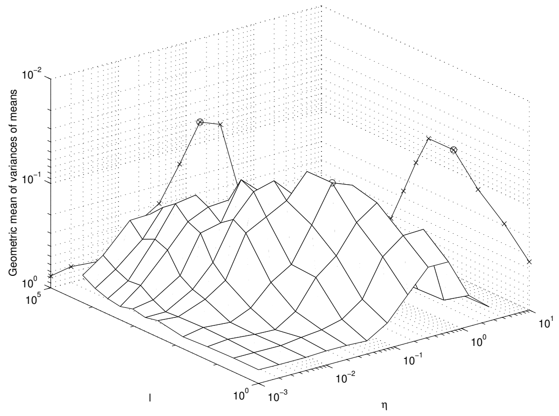
Figure 5.10: 12 hidden units: a circle is drawn at the optimum point, and the surface as a function of l and η are backprojected on to the walls at optimal η and l respectively. Note that the vertical scale is inverted.



(a) Gibbs update

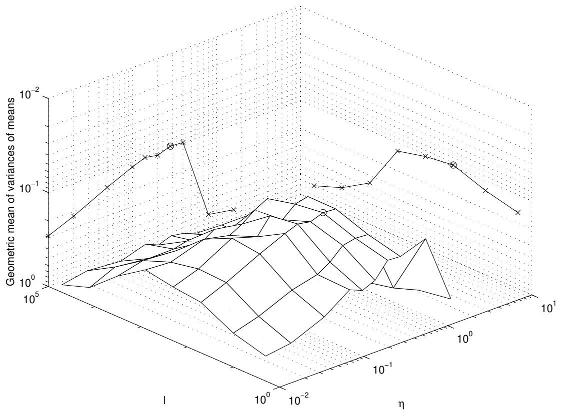


(b) Dynamical A update

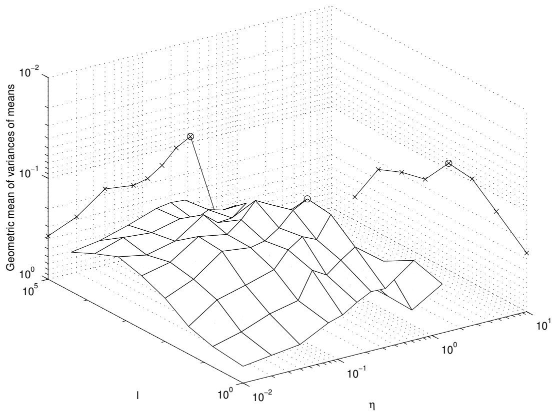


(c) Dynamical B update

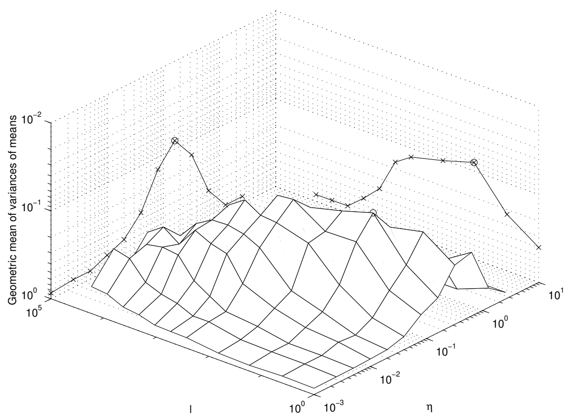
Figure 5.11: 16 hidden units: a circle is drawn at the optimum point, and the surface as a function of l and η are backprojected on to the walls at optimal η and l respectively. Note that the vertical scale is inverted.



(a) Gibbs update



(b) Dynamical A update



(c) Dynamical B update

Figure 5.12: 20 hidden units: a circle is drawn at the optimum point, and the surface as a function of l and η are backprojected on to the walls at optimal η and l respectively. Note that the vertical scale is inverted.

Hyperparameter updates by	Hidden units	l	η	Rejection rate
Gibbs	8	200	0.400	8.2%
Dynamical A	8	25	0.640	42.9%
Dynamical B	8	533	0.040	21.7%
Gibbs	12	200	0.400	12.6%
Dynamical A	12	100	0.453	26.7%
Dynamical B	12	533	0.080	51.6%
Gibbs	16	100	0.400	15.8%
Dynamical A	16	200	0.453	44.0%
Dynamical B	16	66	0.160	30.8%
Gibbs	20	50	0.283	7.9%
Dynamical A	20	100	0.320	15.6%
Dynamical B	20	17	0.160	14.8%

Table 5.11: The optimal tuning parameters for each method and each number of hidden units.

The values of g are plotted for each number of hidden units in Fig. 5.13. The error bars are big, and it is possible that there is essentially no difference in all the three methods. However, there is some evidence that when the number of hidden units N_h is increased to 16, the Dynamical B method begins to work better than the old method. Note that the error bars in the figure are 90% confidence intervals of the performance measures, and do not take into account inaccuracies in assessing the optimal settings of the tuning parameters. Thus, the real error bars are actually bigger by some unknown amount.

From the graph, the Dynamical A method does not seem to perform that differently from the other methods, except at 16 hidden units. The cause of this seeming anomaly at 16 hidden units is discussed in the next chapter.

The graph also shows that the Dynamical B method does not show any improvement over the old method that is measurable given the size of the error bars. This is unfortunate. However, the old method does seem to show a noticeable upward trend, while the two new methods do not. This suggests that the old method is becoming more and more inefficient as the number of hidden units N_h increases even though the compute time given to it is also increasing linearly with N_h. Thus, the graph suggests that the compute time required to maintain the same level of performance as measured by g grows superlinearly with N_h for the old method. On the other hand, the two new methods appear more likely to be either linear or sublinear, although it is difficult to tell for certain with the limited number of data points and the noise.

The size of the error bars impedes our analysis of the results. In the next section, we show how we can perform bootstrapping on pairwise comparisons to obtain clearer indications of how one method does compared to another.

5.7 Pairwise Bootstrap Comparison

To more sensitively compare how two methods perform, we can compute the ratio of g's for two different methods, and use bootstrapping to obtain a confidence interval for that ratio. Here, a bootstrap realization is a choice of 50 Markov chain start states rather than Markov chains, as the chains themselves differ for the two methods. This couples the g's for the 2 methods together, causing them to be evaluated at the same Markov chain start state for each bootstrap realization. In this way, we might be able to better distinguish between good and bad methods. For example, we might see that one method always has a higher g than another when started from the same point even though their individual g's wander over a large range for different bootstrap realizations so that the error bars in the 2 g's overlap significantly.

In Fig. 5.15, we show these ratios with 90% confidence intervals obtained using 500

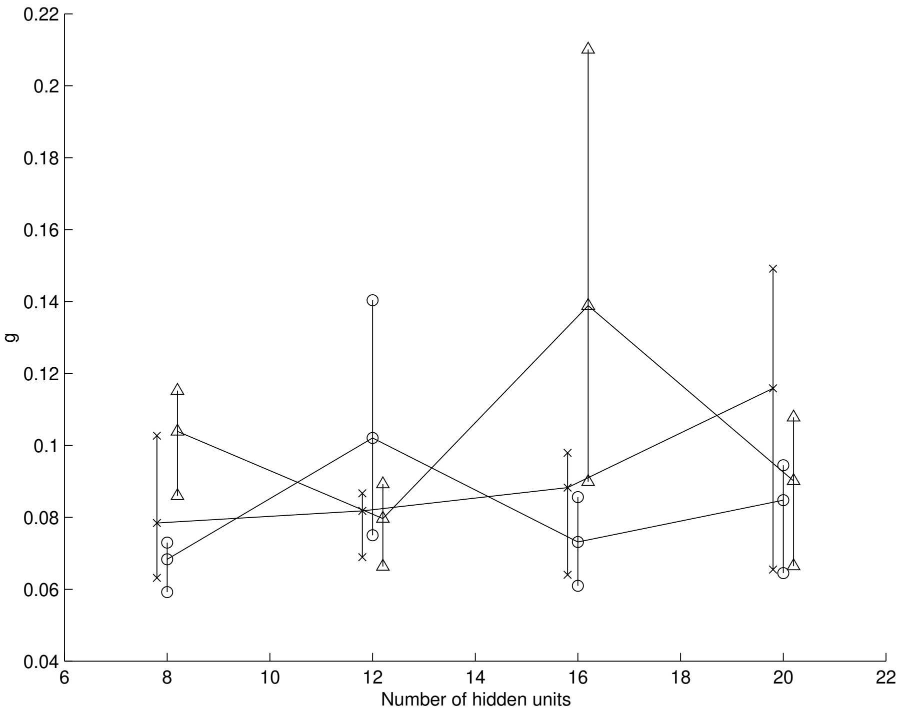


Figure 5.13: Comparison of g between the three methods at the optimal setting of the tuning parameters as the number of hidden units increases. Crosses are the old method, triangles are the Dynamical A method, and circles are the Dynamical B method. Error bars terminate with the same symbol that represents each point. These error bars represent 90% confidence intervals, and do not take into account the error in the estimation of the optimal tuning parameters. Thus, the true error bar is greater than those shown.

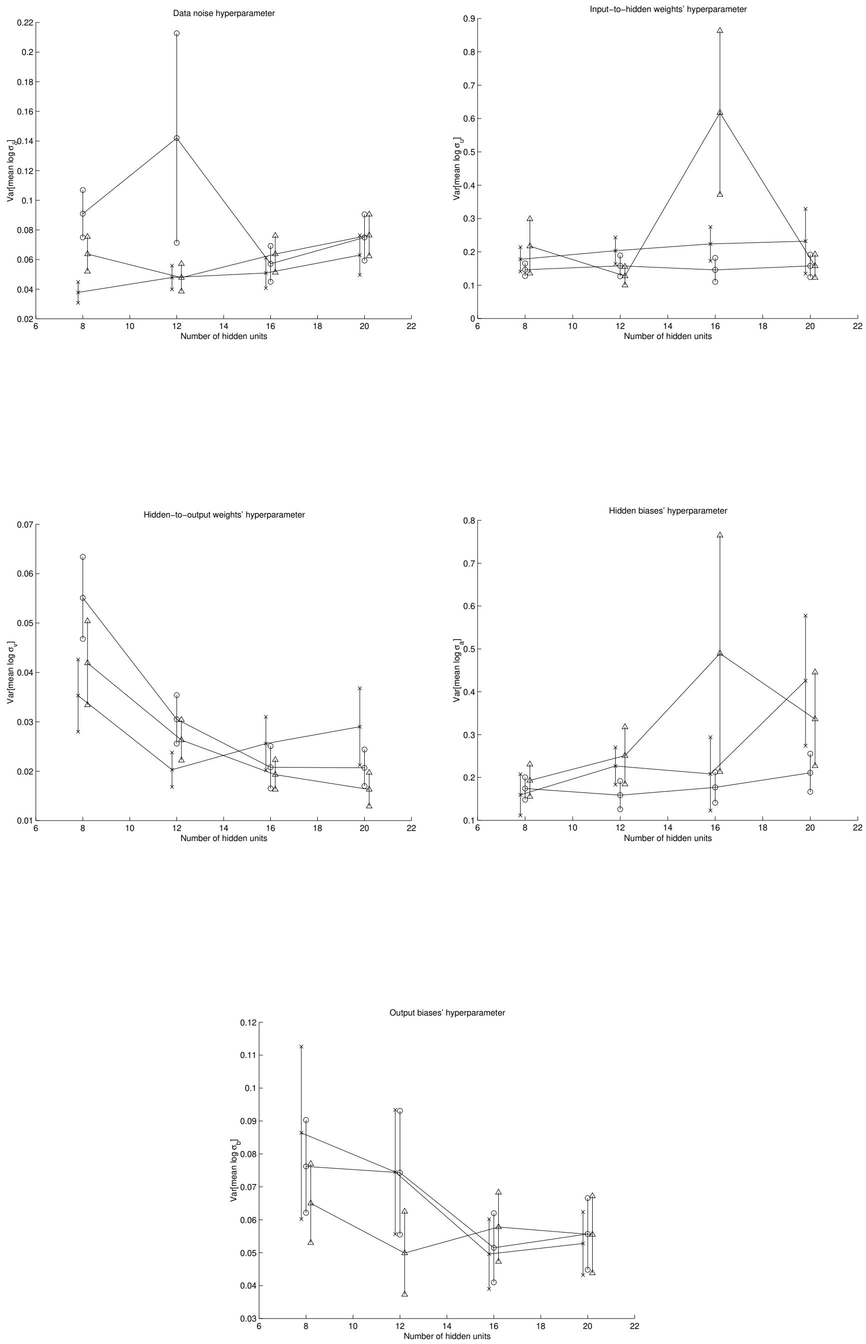
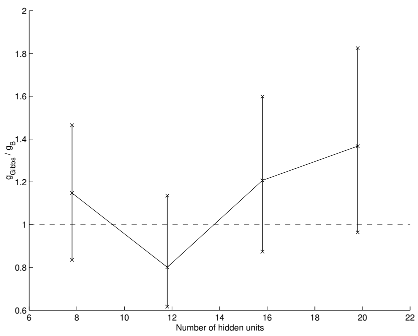


Figure 5.14: Comparison of optimal performance of the three methods as the number of hidden units increases. Error bars terminate with the same symbol that represents each point. These error bars are the standard deviation in the each variance of means measure, and do not take into account the error in the estimation of the optimal tuning parameters. Thus, the true error bar is greater than those shown.

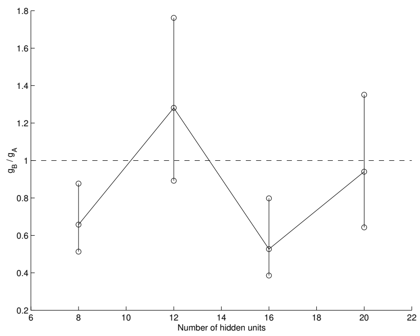
boostrap samples. From Fig. 5.15a, it seems fairly convincing that Dynamical B works better than the old method for 20 hidden units. Fig. 5.15b suggests that Dynamical B tends also to work better than Dynamical A, while Fig. 5.15c is inconclusive on the relative performances of Dynamical A and the old Gibbs method. However, we should note that that there is some uncertainty in the error bars due to the fact that we might not really have found the true optimal settings of the tuning parameters. Also, the 50 Markov chains we used might not have been enough to capture all the important regions.

Thus, even though the pairwise comparisons might suggest that Dynamical B works better than the Gibbs method for 20 hidden units, it is better to be cautious and conclude that Dynamical B **may** work better than Gibbs, and if it does, it is not by much.

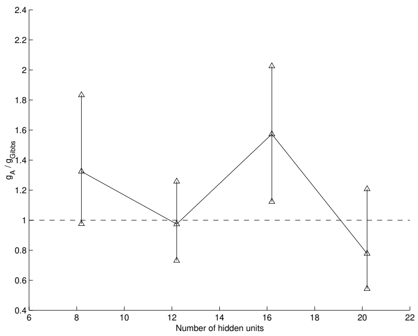
Now that it is clear that Dynamical B is not as good as one might hope, the question is, why is that, and can it be made to go faster? We address these questions, as well as the question of Dynamical A's large error bars in g at 16 hidden units, in the next chapter.



(a) Gibbs vs. Dynamical B



(b) Dynamical B vs. Dynamical A



(c) Dynamical A vs. Gibbs

Figure 5.15: Pairwise comparisons of the various methods obtained by taking the ratio of g. Error bars represent 90% confidence intervals obtained using 500 bootstrap realizations. Dashed lines have been drawn at the level of 1, which signifies equal performance.

Method	Hidden units	ρ_δ	ρ_u	ρ_v	ρ_a	ρ_b	g
Gibbs	8	0.0378 $\pm$ 0.0070	0.1777 $\pm$ 0.0367	0.0353 $\pm$ 0.0073	0.1595 $\pm$ 0.0480	0.0864 $\pm$ 0.0262	0.0784
Dynamical A	8	0.0637 $\pm$ 0.0117	0.2171 $\pm$ 0.0815	0.0419 $\pm$ 0.0085	0.1931 $\pm$ 0.0376	0.0650 $\pm$ 0.0120	0.1039
Dynamical B	8	0.0909 $\pm$ 0.0160	0.1463 $\pm$ 0.0190	0.0551 $\pm$ 0.0083	0.1743 $\pm$ 0.0261	0.0762 $\pm$ 0.0141	0.0683
Gibbs	12	0.0479 $\pm$ 0.0080	0.2034 $\pm$ 0.0399	0.0203 $\pm$ 0.0035	0.2267 $\pm$ 0.0433	0.0745 $\pm$ 0.0189	0.0818
Dynamical A	12	0.0478 $\pm$ 0.0093	0.1277 $\pm$ 0.0279	0.0263 $\pm$ 0.0041	0.2510 $\pm$ 0.0667	0.0499 $\pm$ 0.0126	0.0797
Dynamical B	12	0.1420 $\pm$ 0.0707	0.1581 $\pm$ 0.0313	0.0305 $\pm$ 0.0049	0.1586 $\pm$ 0.0330	0.0743 $\pm$ 0.0188	0.1021
Gibbs	16	0.0509 $\pm$ 0.0102	0.2238 $\pm$ 0.0509	0.0256 $\pm$ 0.0054	0.2083 $\pm$ 0.0856	0.0496 $\pm$ 0.0106	0.0883
Dynamical A	16	0.0637 $\pm$ 0.0124	0.6173 $\pm$ 0.2459	0.0193 $\pm$ 0.0030	0.4893 $\pm$ 0.2758	0.0578 $\pm$ 0.0105	0.1389
Dynamical B	16	0.0571 $\pm$ 0.0121	0.1460 $\pm$ 0.0359	0.0208 $\pm$ 0.0043	0.1768 $\pm$ 0.0357	0.0515 $\pm$ 0.0105	0.0731
Gibbs	20	0.0630 $\pm$ 0.0134	0.2319 $\pm$ 0.0973	0.0290 $\pm$ 0.0078	0.4259 $\pm$ 0.1517	0.0528 $\pm$ 0.0096	0.1159
Dynamical A	20	0.0764 $\pm$ 0.0141	0.1576 $\pm$ 0.0348	0.0163 $\pm$ 0.0034	0.3363 $\pm$ 0.1092	0.0555 $\pm$ 0.0117	0.0901
Dynamical B	20	0.0749 $\pm$ 0.0156	0.1579 $\pm$ 0.0337	0.0207 $\pm$ 0.0037	0.2109 $\pm$ 0.0445	0.0557 $\pm$ 0.0109	0.0848

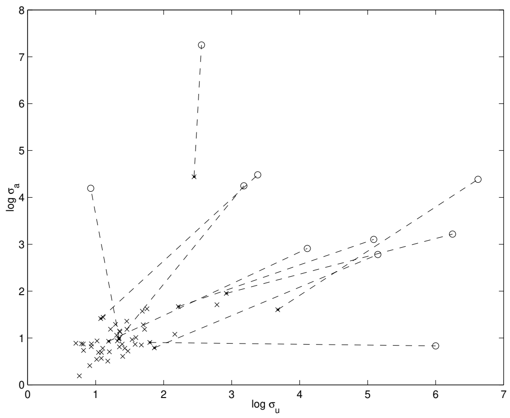
Table 5.12: Variance of means performances obtained by re-running the various methods with new random seeds at the optimal tuning parameter settings. Each error bar is the standard deviation of its variance of means measure.

Chapter 6

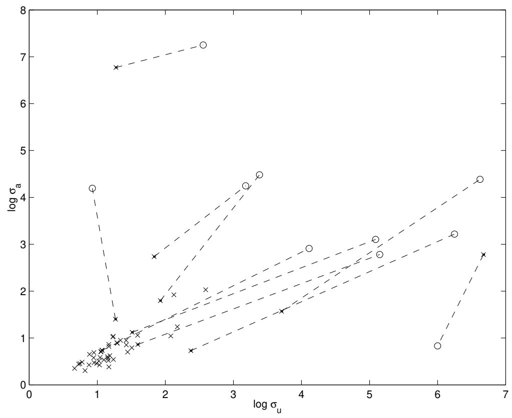
Discussion

6.1 Has the Reparameterization of the Network Weights Been Useful?

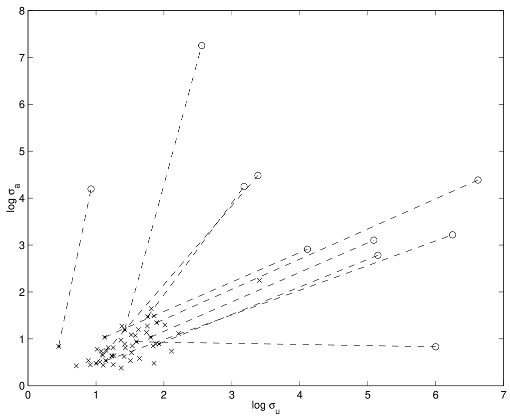
As we saw in the last chapter, the Dynamical A method seems to perform comparably to Dynamical B except for 16 hidden units. Upon closer examination, we see that the large value of g for the Dynamical A method at 16 hidden units is due to it not performing well for some Markov chain starting states with large hyperparameter values for $\log \sigma_u$ and $\log \sigma_a$, i.e., the second stratum. This can be seen in Fig. 6.1, and is a manifestation of Dynamical A's inability to move efficiently between large and small values of hyperparameters. This effect is less pronounced for 20 hidden units probably because the starting states in the second stratification are less extreme in value. As mentioned before, this is an aspect of the experiment that is difficult to control. Nevertheless, our present results indicate that the Dynamical A method should be avoided because it may move extremely slowly from some starting states.



(a) Gibbs update



(b) Dynamical A update



(c) Dynamical B update

Figure 6.1: These plots compare how the various method fare on the 10 Markov chain starting states with large hyperparameters for 16 hidden units. The circles show the 10 starting states with large hyperparameters, the crosses show the hyperparameter means of all 50 chains, while the dots show the means for the 10 Markov chains that started at the large hyperparameter values. Dashed lines connect each starting state with its resulting mean.

6.2 Making the Dynamical B Method Go Faster

As we saw in the last chapter, the Dynamical B method offers only a small improvement over the old method, if any at all. The question then is, why.

The key may lie in the rejection rates. As can be seen in Fig. 6.2, the rejection rates of the Dynamical B method differ from those of the old method and the Dynamical A method in that it shows a significant increase with leapfrog trajectory length l.

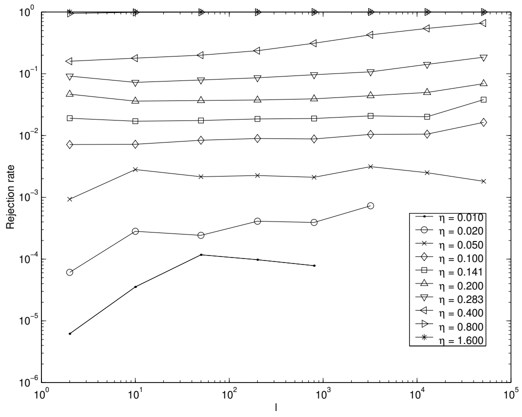
The primary motivation for the new methods is to allow the hyperparameters to be updated with the parameters during the course of a leapfrog trajectory. For this to explore the posterior distribution efficiently, long trajectory lengths are necessary. As we can see from Fig. 6.2, the Dynamical B method ended up having rejection rate characteristics that penalizes long trajectory lengths with high rejection rates. Thus, the optimal setting of l is not as long as we might like.

With this observation in mind, if we can find out why the Dynamical B method has this behaviour, we might be able to fix it. The next section formulates a simple model that accounts for this increase in rejection rate with l.

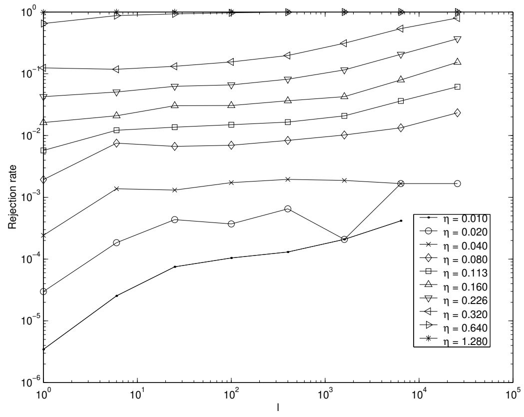
6.2.1 Explanation for the Rising Rejection Rates

If the stepsizes were infinitesimally small, the leapfrog trajectories would simulate Hamiltonian dynamics perfectly, and the rejection rate would be zero. However, they are not, and are furthermore calculated by heuristics that may yield inappropriate values sometimes. This leads to rejection behaviour that affects the rate at which the Markov chain explores the state space.

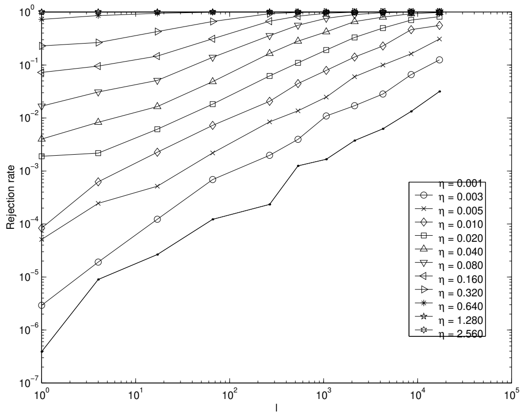
There are two qualitatively different ways by which a leapfrog proposal under the Dynamical B method may be rejected. In the first way, the leapfrog simulation of the Hamiltonian dynamics is stable and H varies over a small range due to the discrete nature of the simulation. At the end of the trajectory, the distribution of H over this



(a) Gibbs update



(b) Dynamical A update



(c) Dynamical B update

Figure 6.2: Rejection rates versus trajectory length l for 20 hidden units.

small range determines the rejection rate. In the second way, the leapfrog simulation becomes unstable at some point during the trajectory due to the heuristics yielding stepsizes inappropriate for that region of state space. The effect of this is catastrophic because the simulation is almost never able to recover: the value of H either diverges or moves to a much higher value, resulting in near certain rejection. H has a much greater tendency to move to a higher rather than a lower value because stepsizes that are inappropriately large may still be stable in a wider orbit.

To illustrate this instability, H is plotted in Fig. 6.3 for 10 trajectories started from the same state but with different randomly-selected initial momenta. We see that once an instability can occur at any time, and once it occurs, recovery is virtually impossible, and will lead to near-certain rejection at the end of the trajectory. The cumulative effect of risking a grossly wrong stepsize with each step is that, for very long trajectories, the probability that we manage to get to the end without once having experienced a catastrophic instability is tiny. Therefore, the rejection rate should increase with trajectory length. This explains the observed increase of the rejection rate with l for Dynamical B.

Another view of the instability of H is shown in Fig. 6.4, which shows how the distribution of H broadens with increasing number of leapfrog steps.

The old method, which updates the hyperparameters by Gibbs sampling, is not prone to this cumulative rejection effect as its stepsize is not being constantly recalculated during a trajectory. Even if its stepsize heuristic gives inapporpriate stepsizes with too high a probability, the stepsizes are computed only once at the beginning of the trajectory, and a good stepsize will tend to lead to a stable trajectory no matter how long it is. Such long trajectories then become unstable only by entering a region for which its stepsizes are inappropriate. This effect leads to rejection rates that increases with l as, the longer the trajectory, the more likely such regions are encountered. However, the fact that rejection rates for the old method show very little dependency on l (see Fig. 6.2) indicates that entry into such regions happens very rarely. Thus, for the old method, most rejections

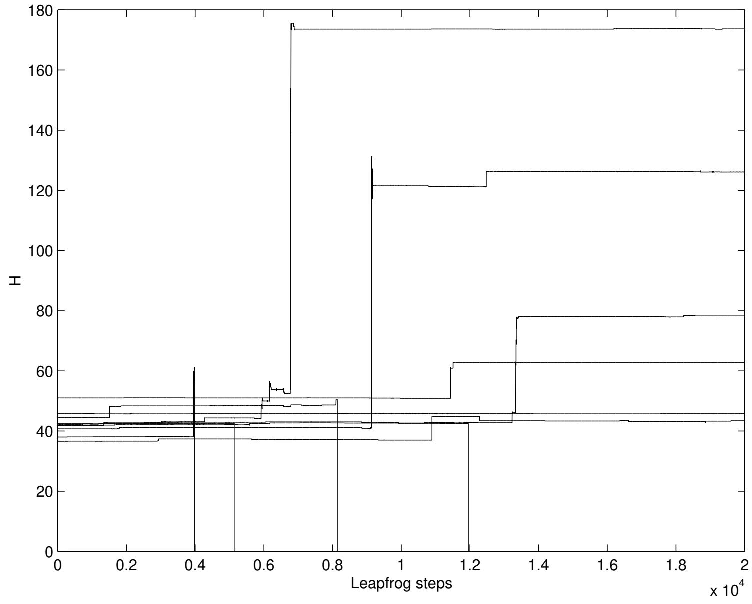
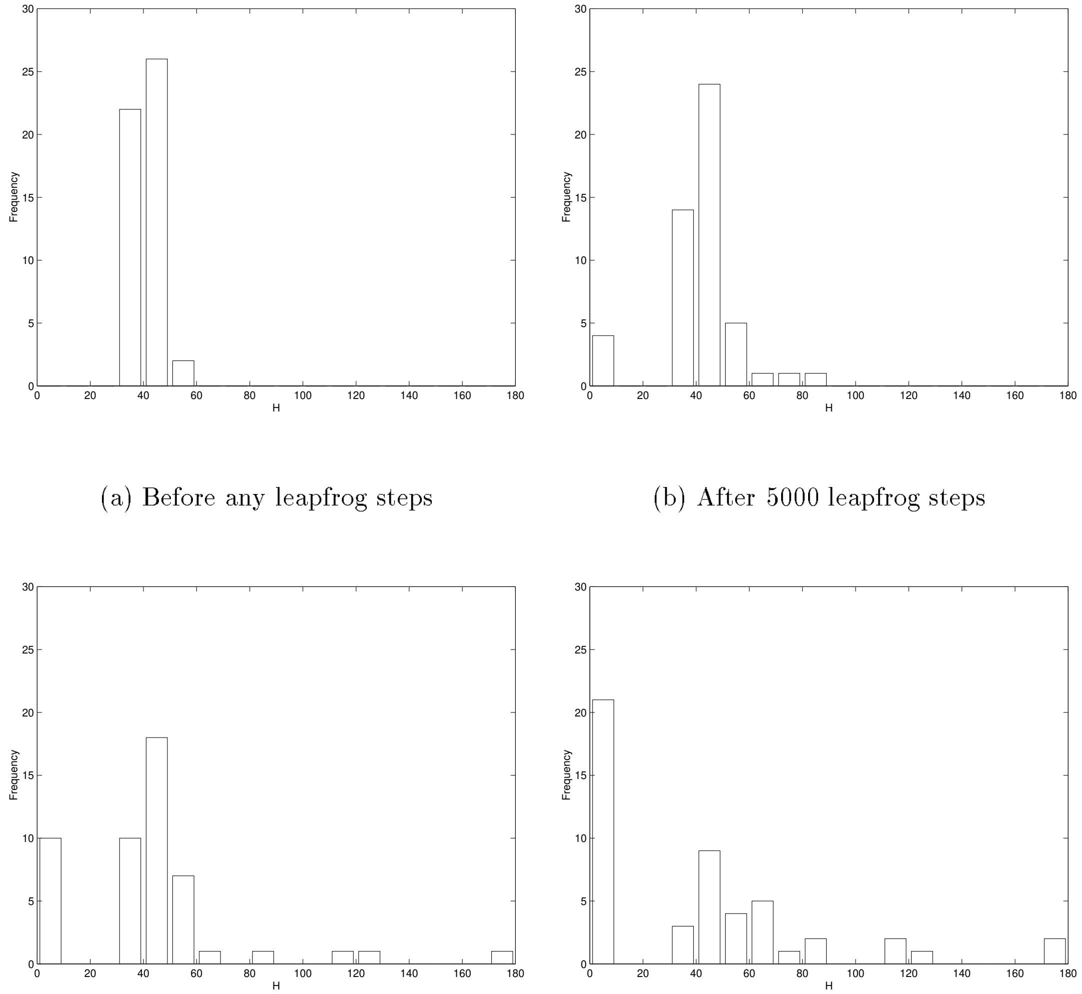


Figure 6.3: H plotted over 10 trajectories started from the same state but with different initial momenta. Each trajectory has 20000 leapfrog steps. As each trajectory progresses, it may become unstable. If it does, H usually rises catastrophically. A transition to infinite H is shown here as a transition to 0. η was set at 0.040, and a network of 8 hidden units was used.

are due to the normal deviation of H away from its initial value.

That the repeated stepsize calculations handicaps the Dynamical B method with its cumulative rejection effect might be cause for pessimism. However, the fact that the Dynamical A method achieves fairly flat rejection rates (Fig. 6.2) shows that the rise in rejection rates is not an inescapable cost of calculating the stepsizes before each step; rather, with appropriate stepsize heuristics, it might be possible to achieve flat rejection rates even in the Dynamical B method.

The stepsize heuristics used in the Dynamical B method for the parameters are the same as that used in the old method, so they are unlikely to be the cause of the catastrophically wrong stepsizes. We expect that it is the stepsize heuristics for the hyperparameters that is at fault. The next section seeks to confirm this.



(a) Before any leapfrog steps

(b) After 5000 leapfrog steps

(c) After 10000 leapfrog steps

(d) After 20000 leapfrog steps

Figure 6.4: The distribution of H broadens as a trajectory progresses. Infinity is binned at 0 in these histograms. 50 trajectories, all started at the same state but with different initial momenta, were used to generate these histograms. η was set at 0.040. A network of 8 hidden units was used.

6.2.2 The Appropriateness of Stepsize Heuristics

If a stepsize is inappropriately large and leads to instability, then perhaps a smaller stepsize half as large might not. If, for example, the parameter stepsizes are sometimes inappropriate, then under a leapfrog discretization where the hyperparameter update remains the same but where each parameter update is split into two consecutive updates, each with half the stepsize adjustment factor, the acceptance probability p should increase, since the parameter update is now closer to the true Hamiltonian dynamics. On the other hand, if the parameter stepsizes are usually appropriate and it is the hyperparameter stepsizes that are at fault, then the rejection rate should not change much.

As shown in Fig. 6.5, when the parameter updates were split, the rejection rates did not change, while they dropped when the hyperparameter updates were split. This indicates that the hyperparameter stepsizes calculated according to the current heuristics are often inappropriate. Fig. 6.5 was obtained by averaging the rejection rates over a small number of Markov chains run at various settings of l with $\eta = 0.010$. A network with 8 hidden units was used, along with the training data from the last chapter. The Markov chain starting states were chosen from the ones used in the tests in the last chapter, with momenta randomly initialized from the unit normal distribution.

To remedy the inappopriateness of the hyperparameter stepsizes, it is possible that the stepsize heuristics for the hyperparameters needs to be changed. On the other hand, it is also possible that some setting of $\eta_h < \eta_p$ rather than $\eta_h = \eta_p$ is all that is necessary. In the next two sections, we explore these two possibilities.

6.2.3 Different Settngs for η_h/η_p

In this section, we report the results of some experiments to test the possibility that some setting of $\eta_h < \eta_p$ can flatten the rejection rate versus l curves without us having to change the stepsize heuristics.

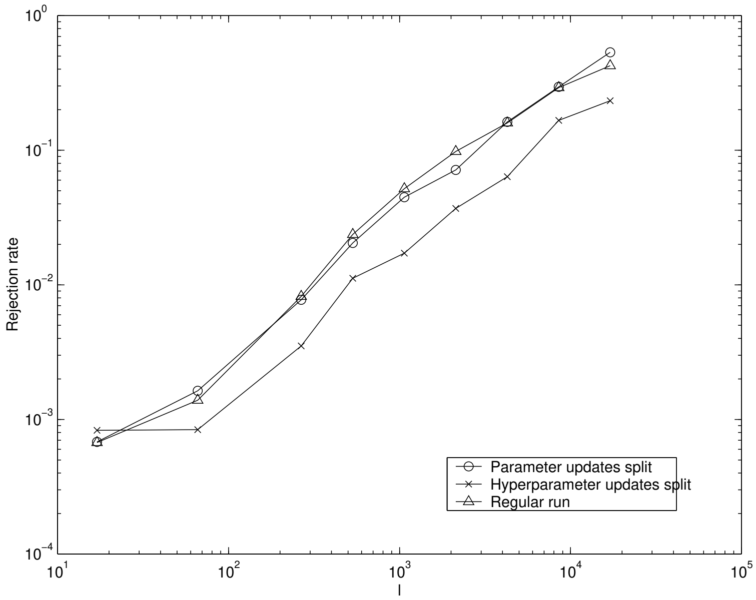
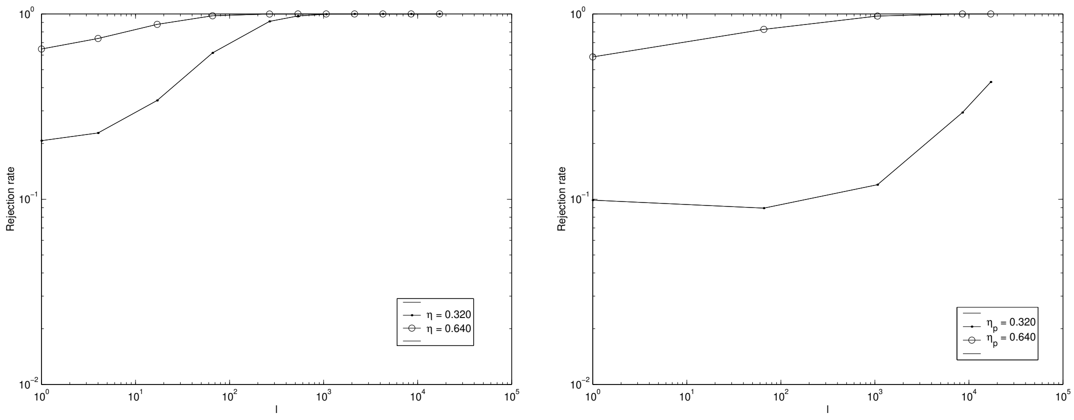


Figure 6.5: Effect on rejection rates of splitting either the parameter updates or the hyperparameter updates into two while using the Dynamical B method. Here, $\eta_h = \eta_p = 0.010$. This plot was obtained using a network with 8 hidden units.

We used a network with 16 hidden units and the training data from the last chapter. 20 Markov chains were run at various trajectory lengths l with $\eta_p = 0.32$ and 0.64, and $\eta_h = \eta_p/100$. The Markov chain starting states were taken from the states used for the tests in the last chapter, and the momenta were randomly initialized from a unit normal distribution. The resulting rejection rates averaged over the 20 chains are plotted in Fig. 6.6. Compared to the case when $\eta_h = \eta_p$, the rejection rates do not rise as fast as the trajectory length increases.

This suggests that, by decreasing the ratio η_h/η_p, it may be possible to gain enough of the advantage of having long trajectory lengths to offset the smaller distances travelled in each step due to the smaller η_h.

The geometric mean performance measure g was also calculated at each setting of l. It was found that the best value of g is 0.14, which occurs at $l = 17067, \eta_p = 0.32$. This value of g is considerably worse than the optimal one (0.0731) found for the Dynamical B



(a) $\eta_h = \eta_p = 0.32$ and $\eta_h = \eta_p = 0.64$ (b) $\eta_p = 0.32$ and $\eta_p = 0.64$. $\eta_h = \eta_p/100$

Figure 6.6: These pictures show rejection rates for 16 hidden units when different ratios of η_h/η_p are used. The rejection rates do not rise as fast with increasing trajectory length when η_h is set to be smaller than η_p. The first set of rejection rates were averaged over 50 Markov chains while the second were obtained using 20.

method at this number of hidden units. So we see that, even though we are able to take much longer trajectories with smaller η_h/η_p, the smallness of η_h may erase our advantage. It is possible that some other setting of η_p, or some other ratio of η_h/η_p does better than setting $g = 0.0731$, but this question will not be explored further in this thesis due to time constraints. The key conclusion of this section is that smaller ratios of η_h/η_p can flatten the rejection curve and may be more advantageous than simply setting $\eta_h = \eta_p$.

6.2.4 Fine Splitting of Hyperparameter Updates

Apart from setting a low ratio for η_h/η_p, we conjecture that, with sufficiently good heuristics, we should also be able to get the rejection rate to stay flat as l increases. To test this, we split the hyperparameter updates into 100 fine updates (many more than the two before). The reason for doing this is that the more stable trajectory obtained by the splitting may roughly model what a good heuristic gives.

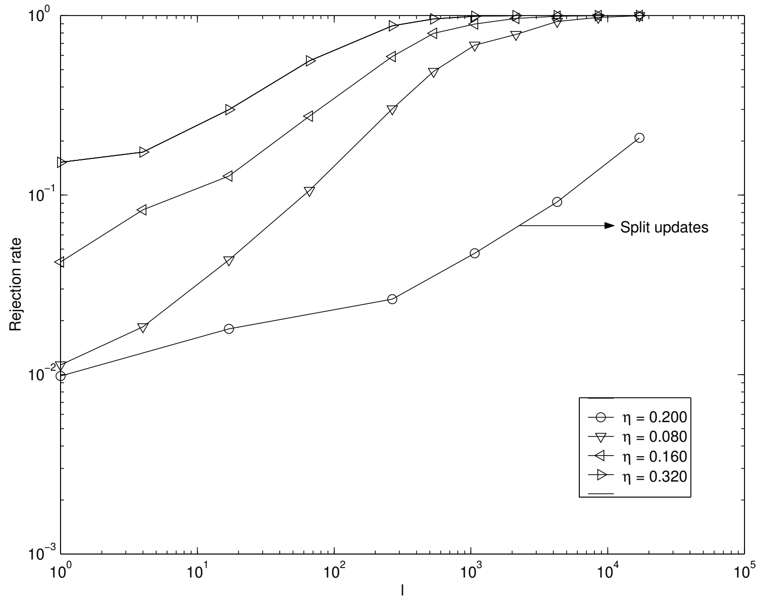


Figure 6.7: Effect on rejection rates of splitting the hyperparameter updates into 100 updates while using the Dynamical B method with $\eta_h = \eta_p = 0.200$. Compared to the normal unsplit updates, rejection rates are now much lower. The network used here has 8 hidden units.

The split hyperparameter updates were tried on a network with 8 hidden units, using the same training data as in the last chapter, and using 10 Markov chain starting states taken also from the tests conducted in the previous chapter. Momenta were randomly initialized from a unit normal distribution. The 10 Markov chains were run at $\eta = 0.20$ with various values of l. Fig. 6.7 shows that the resulting rejection rates for runs with split hyperparameter updates still increases with l, but its rate of increase is much gentler now, increasing about one order of magnitude from about 1%. This is much better than the rejection rate obtained from the normal unsplit updates, which we contrast in the same figure. The unsplit updates do worse even for η less than half the size.

This suggests that, with sufficiently good heuristics, trajectories might be able to go far enough to truly reap the advantages of updating the hyperparameters dynamically.

6.2.5 Why the Stepsize Heuristics are Bad

A possible explanation for why the hyperparameter stepsize heuristics do not work well is that we cannot use the current values of the hyperparameters to compute their stepsizes. As we are unable to get an estimate of them from the weights due to their reparameterization, we are forced to use their prior means, which are not necessarily very good estimates.

In retrospect, this should have been obvious. The backpropagation of the second derivative of the likelihoods multiplies together the hyperparameter variances of each layer of weights that it propagates derivatives through. For instance, the second derivative of the likelihood with respect to the input to hidden weight hyperparameter λ_u is proportional to $1/(\omega_u \omega_v)$, the product of the prior variances of the hidden to output weights and the input to hidden weights. In an 8 hidden unit run, our settings were such that $1/(\omega_u \omega_v) = 0.2$. Yet, from Fig. 5.8, it is clear that, as the Markov chain ranges over the posterior distribution, the product of these two variances can actually range up to $e^{12} \approx 1.6 \times 10^5$ or more. Clearly, 0.2 as an estimate of 1.6×10^5 is bad. This can lead to stepsizes which are $(0.2/e^{-12})^{-0.5} \approx 1000$ times larger than what they would have been if we had used the actual values of the hyperparameters.

The Dynamical A method does not suffer from this multiplicative effect of wrong hyperparameter estimates as its neural network function does not depend on the hyperparameters, so there is no need to do backpropagation of second derivatives. Indeed, the second derivative of the potential energy in Eqn. 4.9 is proportional to just the precision of the hyperparameter, so if our estimate of the hyperparameter is k times too small, the stepsize is only going to go up by a factor of $\sqrt{k}$. Furthermore, the parameters contain information about the value of the hyperparameter: the Dynamical A method estimates a hyperparameter as its posterior mean given its weights.

6.2.6 Other Implications of the Current Heuristics

The fact that the current heuristics uses the prior means of the hyperparameters as estimates of the hyperparameters themselves during stepsize calculations has the implication that, when the Dynamical B method is used, the prior means must be carefully selected to be close to values where the hyperparameters have high posterior probability. This allows the stepsizes to be accurate when moving about in areas of high posterior probability. If the prior means are badly set, exploration of the posterior is expected to become very inefficient.

A hybrid Monte Carlo method that computes bad stepsizes in some region of state space may usually be unable to enter that region because it rejects once a trajectory enters it. Furthermore, once having entered that region, it is obliged to stay in there a long time (through its high rejection rate in that region) in order to compensate for its inability to enter that region in the first place. This leads to high autocorrelations.

We might ask if the B method actually suffers from this problem. If it does, it is small, as an effect like this was not noticed: the marginal posterior distributions of the hyperparameters obtained by the B method matches those from the old (Fig. 5.8). However, this is no guarantee that it will work similarly well for other problems.

6.3 Conclusion

In this thesis, we have introduced a new way of learning the hyperparameters in a neural network model using Hamiltonian dynamics. We have presented two versions of the new method: the Dynamical A method, and the Dynamical B method, which is the former with weights reparameterized to enhance movement between large and small values of the hyperparameters. We have also developed performance evaluation methodologies that measure the rate of exploration of the posterior while accounting for the different compute times required for the different methods. The observation that some parts of state space might have a large effect on the results then led us to a stratified version of the performance measures. After extensive testing, we have found that these same regions of state space can cause Dynamical A to become very inefficient for the reasons that led us to formulate Dynamical B. However, we have also shown that Dynamical B does not show a measurable performance improvement over the old method.

The Dynamical A method as it currently stands suffers from expected inefficiencies, but it was hoped that Dynamical B would overcome them and yield better performance. Instead, there is currently no reason to recommend either new method over the old one.

We strongly believe that the Dynamical B method's Achilles' heel is the fact that its rejection rate increases with trajectory length. If longer trajectories can be achieved while keeping rejection rates low, it is expected that the Dynamical B method can become significantly faster. To achieve this, future work might focus on an improved parameterization of the weights and/or more accurate hyperparameter stepsize heuristics. Also, the stepsize heuristics can potentially be simplified to allow more leapfrog steps to be taken.

Ultimately, our efforts are being hampered by the fact that the volume of the state space under the posterior having large hyperparameter variance is huge and low density, while the region having small hyperparameter variance is small and very high density, and hybrid Monte Carlo does not move well between these two types of regions. Doing nothing about this leads to the Dynamical A method, which we have shown can have severe inefficiencies in certain regions. On the other hand, our effort to reparameterize the weights to tackle this problem leads to the hyperparameters being confounded with the weights in the computation of the network output, and that is the cause of our inability to have long trajectories while keeping rejection rates down. It may be that we are pushing against the inherent limitations of the hybrid Monte Carlo method here, but we nevertheless hope that further work will overcome the present difficulties.

Appendix A

Preservation of Phase Space Volume Under Hamiltonian Dynamics

Here, we show the well-known result that Hamiltonian dynamics keeps the Hamiltonian H as well as phase space volume constant.

Hamiltonian dynamics is characterized by:

$$\begin{aligned} \dot{q} &= \frac{\partial H}{\partial p} \\ \dot{p} &= -\frac{\partial H}{\partial q} \end{aligned} \tag{A.1}$$

where q and p are the state and the momentum variables respectively.

The following shows that Hamiltonian dynamics keeps H constant:

$$\begin{aligned} \frac{dH}{dt} &= \frac{\partial H}{\partial q}\dot{q} + \frac{\partial H}{\partial p}\dot{p} \\ &= \frac{\partial H}{\partial q}\left(\frac{\partial H}{\partial p}\right) + \frac{\partial H}{\partial p}\left(-\frac{\partial H}{\partial q}\right) \\ &= 0 \end{aligned} \tag{A.2}$$

To show that Hamiltonian dynamics conserves phase space volume, consider the phase space flow $(\dot{q}, \dot{p})$, which defines a vector field in the phase space (q, p). The fact that phase space volume is conserved is due to the fact that the divergence of this vector field is 0:

$$\begin{aligned}
\nabla(\dot{q}, \dot{p}) &= \frac{\partial}{\partial q}\dot{q} + \frac{\partial}{\partial p}\dot{p} \\
&= \frac{\partial}{\partial q}\left(\frac{\partial H}{\partial p}\right) + \frac{\partial}{\partial p}\left(-\frac{\partial H}{\partial q}\right) \\
&= \frac{\partial^2 H}{\partial q \partial p} - \frac{\partial^2 H}{\partial p \partial q} \\
&= 0
\end{aligned} \tag{A.3}$$

Appendix B

Proof of Thorem 2: Deterministic Proposals for Metropolis Algorithm

Here, we prove Theorem 2 from Section 2.3.1.

First, we need the following definition:

Definition 7 (Detailed Balance) *We say that the transition probabilities $T(x, A)$ defined for all points x and all sets A satisfies* **detailed balance** *with respect to the density $\pi(x)$ if, given any two sets A and B:*

$$\int_A \pi(x)T(x, B)dx = \int_B \pi(x)T(x, A)dx \tag{B.1}$$

In words, the detailed balance condition says that, in equilibrium, the probability of starting in A and moving to B in one transition is exactly equal to the probability of starting in B and moving to A.

Recall that to say that a Markov update leaves a distribution $\pi(x)$ invariant is to say that the total probability mass $\pi(A)$ in some arbitrary set A is unchanged by the Markov transition. That is:

$$\int_R \pi(x)T(x, A)dx = \pi(A) \tag{B.2}$$

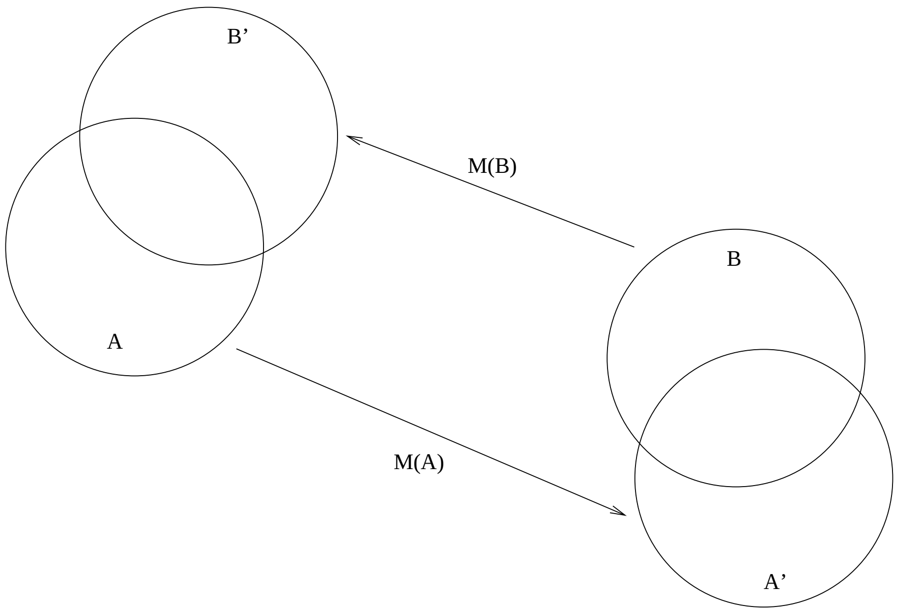


Figure B.1: A maps to A' under M, and B maps to B' under M.

where R is the state space.

It is easy to see that if a Markov transition satisfies detailed balance with respect to $\pi(x)$, then it leaves $\pi(x)$ invariant: we need only set B to R to see this. Thus, we only need to prove that the Metropolis algorithm with deterministic proposals satisfying the two conditions in Theorem 2 yields Markov updates that satisfy detailed balance with respect to the desired distribution $\pi(x)$. That is our aim in the below discussion.

Consider the deterministic mapping $M : R \rightarrow R$. Let $A \subseteq R$ and $B \subseteq R$. Under M, the image of A might in general have some part outside B, and the image of B might have some part outside A, as shown in Fig. B.1.

Assume that the mapping $M(\cdot)$ is the inverse of itself so that $M(M(x)) = x$, we must have that the $M(A \cap B') = B \cap A'$. Suppose not. Then, there is a point $x \in A \cap B'$ that maps into $\overline{B} \cap A'$, which we show as "$M(x)$" in Fig. B.2. ($M(x)$ must be in A' as $x \in A$.) But the fact that x is in B' means that it is the image under $M(\cdot)$ of some point y in B, so that $M(y) = x$. If indeed x maps on to the point "$M(x)$", then $M(M(y)) \neq y$, for y is in B but "$M(x)$" is not. Since this violates the assumption that $M(\cdot)$ is its own

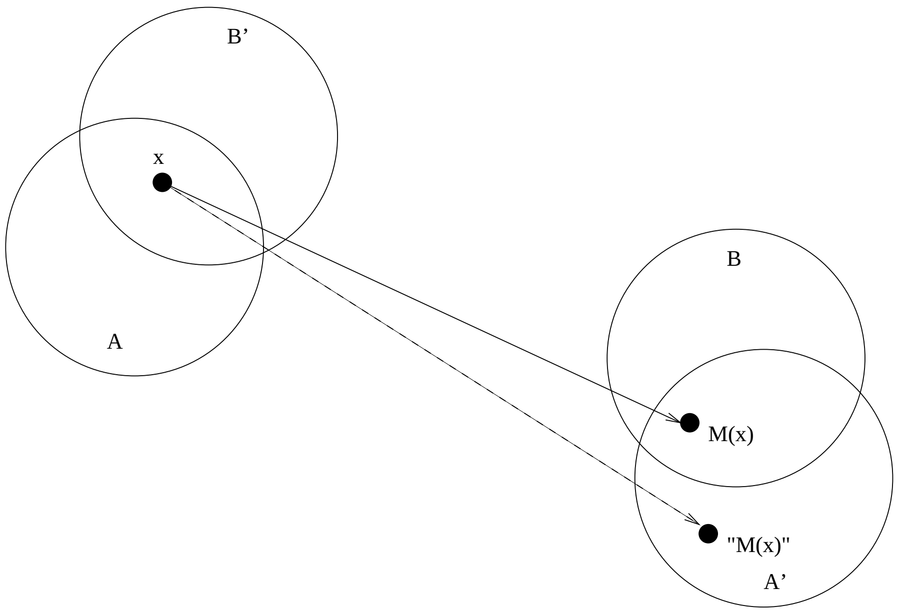


Figure B.2: $x \in A \cap B'$ must map onto $B \cap A'$.

inverse, we conclude that $M(A \cap B') = B \cap A'$. Note that $M(B \cap A') = A \cap B'$ also due to the self-inverting assumption on $M(\cdot)$.

With this fact in hand, we are ready to show how to achieve detailed balance for the Metropolis algorithm. We assume self-inverting deterministic proposals. Since the total probability mass flowing from A to B takes place in the flow from $A \cap B'$, we can rewrite the left hand side of Eqn. B.1:

$$\begin{aligned}\int_A \pi(x)T(x,B)dx &= \int_{A\cap B'} \pi(x)T(x,B)dx \\ &= \int_{A\cap B'} \pi(x)\min\left(1, \frac{\pi(M(x))}{\pi(x)}\right)dx \\ &= \int_{A\cap B'} \min[\pi(x), \pi(M(x))]dx\end{aligned} \tag{B.3}$$

where we have used the Metropolis transition probability $T(x,B) = \min[1, \pi(M(x))/\pi(x)]$.

We can then rewrite the right hand side of Eqn. B.1:

$$\begin{aligned}
\int_B \pi(y)T(y,A)dy &= \int_{B\cap A'} \pi(y)T(y,A)dy \\
&= \int_{B\cap A'} \pi(y)\min\left(1, \frac{\pi(M(y))}{\pi(y)}\right)dy \\
&= \int_{M(A\cap B')} \pi(y)\min\left(1, \frac{\pi(M(y))}{\pi(y)}\right)dy \\
&= \int_{M(A\cap B')} \min[\pi(y), \pi(M(y))]dy \\
&= \int_{A\cap B'} \min[\pi(M(x)), \pi(M(M(x)))]\left|\frac{\partial y}{\partial x}\right|dx \quad (\text{using } y = M(x)) \\
&= \int_{A\cap B'} \min[\pi(M(x)), \pi(x)]\left|\frac{\partial y}{\partial x}\right|dx
\end{aligned} \tag{B.4}$$

Comparing the expressions resulting from manipulating the left and the right hand sides of the detailed balance condition, we see that they are equal if the Jacobian $|\partial y/\partial x| = 1$. Thus, detailed balance with respect to $\pi(x)$ is achieved if the Metropolis algorithm is used with deterministic proposals that are reversible (self-inverting) and that have Jacobian 1, and we are done.

Appendix C

Preservation of Phase Space Volume Under Leapfrog Updates

In this Appendix, we show that leapfrog updates preserve phase space volume.

Consider the simultaneous update of 2 variables such that each update does not depend on the variable it updates, but depends on the value of the other variable:

$$\begin{aligned} x' &\leftarrow x + f(y) \\ y' &\leftarrow y + g(x) \end{aligned} \tag{C.1}$$

It can easily be shown that the Jacobian of such an update is $1 - f'(y)g'(x)$, so the update does not preserve phase space volume in general. On the other hand, consider two sequential updates where each update also depends on the variable updated by the other only, but the updates are performed one after another:

$$\begin{aligned} x' &\leftarrow x + f(y) \\ y' &\leftarrow y + g(x') = y + g(x + f(y)) \end{aligned} \tag{C.2}$$

The Jacobian of such an update can be shown to be identically equal to 1, and so it preserves phase space volume. This is simply a demonstration of the fact that each sequential update of a variable that does not depend on itself amounts to a shear in the direction of that variable. Thus, each sequential update preserves phase space volume, and indeed, a chain of such updates does as well.

The leapfrog updates are:

$$\begin{aligned}
p_i(t + \frac{\epsilon}{2}) &= p_i(t) - \frac{\epsilon}{2}\frac{\partial E}{\partial q_i}(\mathbf{q}(t)) \quad \text{for each } i = 1..d \\
q_i(t + \epsilon) &= q_i(t) + \epsilon\frac{p_i(t + \epsilon/2)}{m_i} \quad \text{for each } i = 1..d \\
p_i(t + \epsilon) &= p_i(t + \frac{\epsilon}{2}) - \frac{\epsilon}{2}\frac{\partial E}{\partial q_i}(\mathbf{q}(t + \epsilon)) \quad \text{for each } i = 1..d
\end{aligned} \tag{C.3}$$

The update for $\mathbf{p}$ appears to be simultaneous in that p_i is updated without reference to any newly-updated components of $\mathbf{p}$. However, in the leapfrog update, the update of each p_i does not actually depend on any other component of $\mathbf{p}$, so the update of each p_i can be viewed as being sequential and conserving phase space volume. The same applies to q_i.

Bibliography

C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

W. L. Buntine and A. S. Weigend. Bayesian back-propagation. *Complex Systems*, 5:603–643, 1991.

G. Cybenko. Approximation by superpositions of a sigmoid function. *Mathematics of Control, Signals and Systems*, 2:303–314, 1989.

L. Devroye. *Non-Uniform Random Variate Generation.* Springer-Verlag, 1986.

S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth. Hybrid monte carlo. *Physics Letters B*, 195(2):216–222, September 1987.

B. Efron and R. Tibshirani. *An Introduction to the Bootstrap.* Chapman and Hall, 1993.

W. Feller. *An Introduction to Probability Theory and Its Applications.* John Wiley and Sons, Inc., 1966.

D. J. C. MacKay. *Bayesian Methods for Adaptive Models.* PhD thesis, California Institute of Technology, 1991.

D. J. C. MacKay. Comparison of approximate methods for handling hyperparameters. *Neural Computation*, 11:1035–1068, 1999.

P. B. Mackenzie. An improved hybrid monte carlo method. *Physics Letters B*, 226:369–371, August 1989.

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*, 21:1087–1092, 1953.

P. Müller and D. R. Insua. Issues in bayesian analysis of neural network models. *Neural Computation*, 10:749–770, 1998.

R. M. Neal. Probabilistic inference using markov chain monte carlo methods. Technical Report CRG-TR-93-1, University of Toronto, September 1993.

R. M. Neal. *Bayesian Learning for Neural Networks*. Springer-Verlag, 1996.

J. S. Rosenthal. A review of asymptotic convergence of general state space markov chains. March 1999.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.