Bayesian Neural Network Estimation of Next-To-Leading-Order Cross Sections

by

René Alexander Ask

THESIS

for the degree of

MASTER OF SCIENCE

Faculty of Mathematics and Natural Sciences
University of Oslo

Spring 2022

Bayesian Neural Network Estimation of Next-To-Leading-Order Cross Sections

René Alexander Ask

© 2022 René Alexander Ask

Bayesian Neural Network Estimation of Next-To-Leading-Order Cross Sections

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

This is my abstract.

Acknowledgments

Acknowledgments yo

Contents

Introduction **1**

1 The Physics Problem **3**
- 1.1 Computation of Beyond the Standard Model Cross Sections 3
- 1.2 Bayesian Regression as a Substitute 4

2 Bayesian Formulation of Machine Learning **7**
- 2.1 The Core of Machine Learning 7
  - 2.1.1 Loss Functions 7
  - 2.1.2 Regularization 8
  - 2.1.3 Optimization 8
- 2.2 Bayes' theorem 8
- 2.3 Bayesian Framework for Machine Learning 9
- 2.4 Bayesian Inference 10

3 Markov Chain Monte Carlo **13**
- 3.1 Expectation Values and the Typical Set 13
  - 3.1.1 The Typical Set 13
  - 3.1.2 The Target Density and Bayesian Applications 14
- 3.2 Markov Chains and Markov Transitions 14
  - 3.2.1 Ideal Markov Chains 15
  - 3.2.2 Pathologies 15
  - 3.2.3 Geometric Ergodicity and Convergence Diagnostics 15
- 3.3 Metropolis-Hastings 15
  - 3.3.1 The Proposal Distribution 16
- 3.4 Gibbs Sampling 17

4 Hamiltonian Monte Carlo **19**
- 4.1 Hamiltonian Dynamics 19
  - 4.1.1 Leapfrog integration 20
- 4.2 Generating a Proposal State 21
- 4.3 The Potential Energy Function in Bayesian Machine Learning Applications 23
- 4.4 Limitations of Hamiltonian Monte Carlo 23

5 Adaptive Hamiltonian Monte Carlo **25**
- 5.1 The No-U-Turn Sampler 25
  - 5.1.1 Stopping Conditions and Selection of Candidate States 26
  - 5.1.2 Computational Cost 28

6 Bayesian Neural Networks **31**
6.1 Neural Networks 31
6.1.1 Basic Mathematical Structure 31
6.1.2 Backpropagation 32
6.1.3 Regularization in Neural Networks 33
6.2 Activation Functions 34
6.2.1 Sigmoid and Tanh 34
6.2.2 ReLU 34
6.2.3 Swish 34
6.3 Bayesian learning of Neural Networks using Monte Carlo Samplers 34
6.3.1 What *is* Bayesian learning of Neural Networks? 35
6.3.2 The Potential Energy Function of Neural Networks 35
6.3.3 Practical Training of Bayesian Neural Networks 36
6.3.4 Training Algorithm of Bayesian Neural Networks 36

7 Numerical Experiments **39**
7.1 The Dataset 39
7.1.1 The Features and Targets 39
7.1.2 Data Transformations 40
7.1.3 Data Splitting 40
7.2 Methodology 41
7.2.1 Implementation 41
7.2.2 Selection of Models and Hyperparameters 42
7.2.3 Performance Metrics 42
7.2.3.1 Coefficent of Determination 43
7.2.3.2 Standardized Residuals 43
7.3 Results and Discussion 44
7.3.1 Computational Performance 44
7.3.1.1 CPU v. GPU Performance 44
7.3.1.2 Prediction Time 45
7.3.1.3 Loading Times 46
7.3.2 Posterior Distribution of Weights 47
7.3.3 Benchmarks of Hyperparameters 48
7.3.3.1 The Effect of Number of Warm-up Steps 48
7.3.3.2 The Effect of Pretraining 49
7.3.3.3 Effect of Number of Parameters 50
7.3.4 Predictive Distributions 50

Conclusion **64**

Appendices **65**

Appendix A **67**
A.1 Appendix 1 title 67

List of Figures

5.1 The figure shows an example of a trajectory generated by the NUTS sampler. The top diagram displays the projection onto position space with the momenta drawn in as arrows. The bottom diagram shows the resulting balanced binary tree. The tree structure is drawn onto the trajectory as well. The numbering displays the order in which the states are generated by Leapfrog integration. The black node is the initial node. The first doubling is forwards in time and yields the rightmost node of the first binary tree. The second doubling is backwards in time and is initiated from the black node, yielding a new tree of height 2 where the left subtree is the new states (the yellow nodes). The next doubling is also backwards in time, and the Leapfrog integrator is initiated from the tail (the leftmost yellow node) performing four Leapfrog steps generating a subtree which becomes the left half of the next tree (blue nodes). The final doubling in the figure is forwards in time with $L = 8$ Leapfrog steps taken from the orange node (which was the rightmost leaf of the tree before the final doubling) which yields the green nodes. The figure is a modified version of a diagram in [1]. 26

7.1 The values of the cross sections $\sigma_{\tilde{\chi}_1^0\tilde{\chi}_1^0}$ are shown projected onto the axis of masses $m_{\tilde{\chi}_1^0}$. The data is taken from the training data. 40
7.2 The values of the cross sections $\sigma_{\tilde{\chi}_1^0\tilde{\chi}_1^0}$ are shown projected onto the axes of mixing angles N_{1j} for $j = 1, 2, 3, 4$. The data is taken from the training data. 41
7.3 The figure shows the relative measured execution time used per sample using $L = 512$ Leapfrog steps, as a function of number of parameters. The CPU measurements are done using an 8-core M1 CPU (Apple Silicon). The GPU measurements are made with an NVIDIA Tesla P100 GPU. 45
7.4 The figure shows the average prediction time to compute a prediction given a single input x using the models in table 7.1. The average time used is measured in ms and is averaged over 1000 randomly sampled points. The measured time includes computation of the sample mean and sample error. 46
7.5 The figure shows the average prediction time using the built-in GPU on an M1 Apple Silicon system-on-chip to compute a prediction given a single input x using the models in table 7.1. The average time used is measured in ms and is averaged over 1000 randomly sampled points. The measured time includes computation of the sample mean and sample error. 47
7.6 The figure shows the histograms of measured loading times in seconds using the models in table 7.1. The measurements were performed using `time.perf_counter` from Python using an M1 Apple Silicon system-on-chip. The time measurements consist of 1000 measurements for each model. 48
7.7 The figure shows the projection of the empirical distribution onto the planes spanned by $(W^1_{2,4}, W^1_{2,5})$ on the left and onto the plane spanned by $(W^1_{3,7}, W^1_{3,6})$ on the right, using the samples from model 3 in table 7.1. The distributions are approximated using kernel density estimation. 52

7.8 The figure at the top shows the measured time in seconds per sample using HMC as a function of Leapfrog steps L using a model with 561 parameters. The figure at the bottom shows the time in seconds per sample with the same sampler with a fixed number of Leapfrog steps $L = 512$ as a function of number of parameters in the BNN model. 53

7.9 The figure shows the computed R^2-scores in both log space and target space as a function number of warm-up steps (20% burn-in and 80% adaptation) achieved with HMC and NUTS. The architecture of the BNN model used is 5-20-20-1 with $\tanh(x)$ used as the activation function in the hidden layers. We performed 2500 pretraning steps with a batch size of 32 using the ADAM optimizer. In total 1000 neural networks were sampled with 10 steps between each stored sample. When HMC was used, we ran with a fixed number of Leapfrog steps $L = 512$. When the NUTS sampler was used, we allowed for a maximum of $L = 4096$ Leapfrog steps (a maximum tree depth of 12). 54

7.10 The figure shows the standardized residuals computed on the testset. The model architechture used is a model with layers 5-20-20-1 with $\tanh(x)$ as the hidden activation function. In the top figure, we have used the HMC sampler with a fixed number of Leapfrog steps $L = 512$. In the bottom figure, we have used the NUTS sampler with a maximum tree depth of 12 corresponding to a maximum of $L = 2^{12} = 4096$ Leapfrog steps. The remaining important hyperparameters were 2500 pretraining epochs with a batch size of 32 using the ADAM optimizer. In total a 1000 neural networks were sampled in each case with a thinning-amount of 10 steps between each sample. 55

7.11 The figure shows the average number of Leapfrog steps L as a function of number of warm-up steps used by the NUTS sampler when sampling the models shown in the bottom of figure 7.10. We have included a few more measurements to showcase how fluctuating the average number can be. 56

7.12 The figure shows the computed R^2-scores of a model with the architecture 5-20-20-1 with $\tanh(x)$ as the hidden activation function. In this case the varying number of the number of epochs run with pretraining starting from 32 all the way up to 8192. The batch size used was 32 with the ADAM optimizer. The number of warm-up steps was 1000 (200 of which were burn-in steps and 800 were adaptation steps). We fixed the Leapfrog steps to $L = 512$ using the HMC sampler. As usual we sampled 1000 neural networks with 10 steps between each sample. 57

7.13 The figure shows the standardized residuals of a model with the architecture 5-20-20-1 with $\tanh(x)$ as the hidden activation function. In this case the varying number of the number of epochs run with pretraining starting from 32 all the way up to 8192. The batch size used was 32, the number of warm-up steps was 1000 (200 of which were burn-in steps and 800 were adaptation steps). We fixed the Leapfrog steps to $L = 512$ using the HMC sampler. The ADAM optimizer was used for the pretraining phase. As usual we sampled 1000 neural networks with 10 steps between each sample. 58

7.14 The figure shows the R^2-score computed on the training and test data as a function of number of nodes n in the hidden layer of models with architechture 5-n-1, yielding a total of $5n + 1$ parameters. The hidden layer activation used was $\tanh(x)$. The models were trained with 1000 warm-up steps (20% burn-in and 80% adaptation), gathering 1000 neural networks with 10 steps between each sample. We used 2500 pretraining epochs with a batch size of 32. When using the HMC sampler, we fixed the number of Leapfrog steps to $L = 512$. When using NUTS, we set a maximum of $L = 4096$ Leapfrog steps. 59

7.15 The figure shows the distribution of the standardized residuals computed on the test data using the models listed in table 7.1. The Normal distribution is drawn in with a dotted black line for benchmarking reference. The figure is meant to illustrate the performance of the models with respect to the number of parameters in the models. The models were trained with 2500 warm-up steps (20% burn-in and 80% adaptation), gathering 1000 neural networks with 10 steps between each sample. We used 1000 pretraining epochs with a batch size of 32. The kernel used was the NUTS kernel with a maximum of $L = 4096$ Leapfrog steps. 60

7.16 The figure shows the predictive distribution estimated by use of model 3 in table 7.1 for to randomly chosen points from the test set. The red line shows the true target and the black line shows the predicted sample mean obtained from the distribution. The figure on top demonstrates a case where the sample mean is approximately the same as the target, while the figure at the bottom demonstrates a case where the true target lies entirely outside the predictive distrbution. 61

7.17 The figure shows the predictive distribution estimated by use of model 3 in table 7.1 computed on all datapoints in the training, validation and test data. The black line shows the 95% line. The crosses indicate measured data with training data shown in blue, validation data shown in orange and test data shown in green. The y-axis shows the percentage of all targets lie on the interval $[\mu - k\sigma, \mu + k\sigma]$ for $k = 1, 2, 3, 4, 5$ where μ is the sample mean and σ^2 is the sample variance of the predictive distribution. 62

List of Tables

7.1 The table shows the models used in this section. For each model, 1000 sampled networks were sampled to collectively represent each BNN model. We used 2500 warm-up steps (20% burn-in and 80% adaptation). We skipped 10 samples for each sampled network. We performed 1000 pretraining epochs with a batch size of 32 using the ADAM optimizer. The kernel used for each model was the NUTS kernel with a maximum of $L = 4096$ Leapfrog steps. The number of nodes per layer is shown in the “Layers” column. For each hidden layer, we used $\tanh(x)$ as the activation function. The final layer used an identity activation. 42

List of Algorithms

3.1 Metropolis-Hastings 16
3.2 Gibbs sampling 17
4.1 Leapfrog Integration 20
4.2 Vectorized Leapfrog Integration 21
4.3 Hamiltonian Monte Carlo 23
5.1 The NUTS Sampler 28
6.1 Backpropagation: Forward pass 33
6.2 Backpropagation: Backward pass 33

Introduction

Motivation, context and problem.

- Contrast variational inference to Bayesian inference with MCMC methods.

Outline of the Thesis

In chapter 1, we explore the extensive computational cost needed to compute next-to-leading-order cross sections and how Bayesian regression models can serve as a viable substitute for direct calculation of these. In chapter 2, we survey the notion of Bayesian machine learning and how one in general constructs a probabilistic model using Bayes' theorem. In chapter 3 we provide an overview of important ideas for Monte Carlo Markov chains (MCMC) in continuous sample spaces. In chapter 4 we build upon this and show how to construct the first main sampler used in this called Hamiltonian Monte Carlo. In chapter 5 we explore ways to dynamically tune parameters used in the sampler to avoid tedious hand-tuning. In chapter 6, we bring all these topics together, culminating in a training algorithm for Bayesian neural networks using MCMC samplers to sample directly from the exact posterior of the probabilistic model. In chapter 7, we explain the remainder of the methodology and investigate problems such as computational cost, performance on various hardware platforms, reliability of predictions and uncertainty measurements and tuning of the training of Bayesian neural networks.

Chapter 1

The Physics Problem

In this chapter, we shall motivate the need for Bayesian machine learning regression models to replace deterministic methods in high-energy physics in the search for Beyond the Standard Model (BSM) physics. We will start off with a brief survey of the conventional way to compute cross sections, its need for precision and the inherent problems involved. We will end the chapter with a discussion of how Bayesian regression can provide a substitute for the standard way to compute cross sections.

1.1 Computation of Beyond the Standard Model Cross Sections

The Standard Model of particle physics (SM) is a successful fundamental theory that describes the fundamental particles of nature and their interactions. Despite its success, however, it has a few limitations on its own which has led physicists to propose extentions to the model to explain physics that the SM cannot. One such family of extensions is called *supersymmetry*. Theories like this are known as BSM models.

In order to test whether a particular symmersymmetric extension to the SM is valid, one has to search through large (high-dimensinoal) parameter spaces where the parameters themselves somewhat simplified represent the properties of the particles in the model. The technical aspect is to rather *exclude* regions of parameter space which cannot explain observed data. To this end, theoretical physicists must compute what is known as a cross section σ. These are roughly speaking the probability that a particular event occurs in a particular collider experiment. The total number of such events is given by the *event equation*

$$n = \sigma \epsilon A \mathcal{L}, \tag{1.1}$$

where ϵ represents the efficiency of the experimental apparatus, A represents the acceptance and $\mathcal{L}$ is the integrated luminosity of the data taken in the search or experiment, i.e. the amount of data. The job of the theoretical physicist is to compute σ, as all the other quantities can be inferred or measured from the experimental setup used.

We may further decompose the total number events as

$$n = s + b, \tag{1.2}$$

where b is called the *background* which is the portion of the events explained by the SM. Here s represents a portion of n which cannot be explained by SM, but rather the new BSM model, and is called the *signal*. Strictly speaking, the model proposed may only explain a subset of the total events. On a more technical level, the event equation can be divided into several *cuts*. A cut defined a range of an experimentally measured quantity where anything outside of it is excluded. A *signal region* consists of a set of cuts. For a signal region i, the event equation reads

$$n_i = \sigma \epsilon_i A_i \mathcal{L}. \tag{1.3}$$

All but the cross section and the integrated luminosity depend on the signal region.

The computation of σ in eq. (1.1) needs to be carried out to a high accuracy to yield a greater exlusion power. To explain why, consider the Poisson likelihood

$$\mathcal{L}(n|s,b) = \int_0^\infty \frac{[\xi(s+b)]^n e^{-\xi(b+s)}}{n!} P(\xi) \mathrm{d}\xi, \tag{1.4}$$

where ξ is a rescaling parameter and $P(\xi)$ is its probability distribution which is peaked at $\xi = 1$. Its width is defined by

$$\sigma_\xi^2 = \frac{\sigma_s^2 + \sigma_b^2}{(s+b)^2}, \tag{1.5}$$

where σ_s is the systematic uncertainty of the signal predictions s and σ_b is the systematic uncertainty of the background b. The particular form of $P(\xi)$ given a width σ_ξ is typically chosen to be Gaussian or log-normal [2]. To compare s and b correctly to n we must evaluate eq. (1.4). If σ_s is large, this will increase the width σ_ξ which yields a larger value of the likelihood for all points ξ. The consequence is less exclusion power achieved by the statistical analysis for the experimental data from which n was measured.

Computation of cross sections involves computation in quantum field theory of terms in a perturbation expansion which are of the form

$$\sigma = \alpha^2 \sigma_{\mathrm{LO}} + \alpha^4 \sigma_{\mathrm{NLO}} + \text{higher order terms}, \tag{1.6}$$

where α is a small parameter, σ_{LO} is the leading order (LO) term and σ_{NLO} is the next-to-leading order (NLO) term. For supersymmtric models, computation of the cross section used in the event equation is in practice carried out using `Prospino` [3]. It is a software developed to compute cross sections up to the (NLO) term. This computation is exceedingly expensive and can take up to the order of hours for a single tuple of input parameters [4]. This computational expense significantly hampers the investigation of parameter regions of BSM models. The search for new physics is thus halted, not by lack of possible BSM models to explain the discrepancies between the SM predictions and the observed data, but instead by the computational cost to perform the search itself. But the necessity for high accuracy in the computed cross sections used with the event equation forces the theoretical physicist to carry them out regardless, to progress in the search for new physics.

1.2 Bayesian Regression as a Substitute

Regression models are widely employed in problems where direct calculation of a target $y \in \mathbb{R}^d$ from an independent variable $x \in \mathbb{R}^p$ (which we usually call the features) is either too expensive to be considered tractible or the relationship between x and y is difficult to capture from first principles. The typical strategy is to represent the relationship between x and y with a mathematical function imbued with a collection of free parameters which are adjusted according to some “learning” algorithm that given a large number of examples is able to correctly predict the targets of unseen examples. This is what is referred to as *supervised machine learning*. The strategy has proved to be an efficient one, employing what we may coin as *black box* algorithms where we learn a mathematical function which is able to calculate the target given its independent variable without any intrinstic knowledge of the fundamental relationship between the two.

It comes with a major drawback, however. Assessing the accuracy of the prediction is difficult if the target is unknown. This is where *Bayesian regression* comes into the picture. Mathematical models trained within the Bayesian regression framework provides a natural way to not only predict a target y but also yield a corresponding uncertainty in its prediction, given an example of x. The resulting model produces a distribution of targets instead of a single prediction. This allow for a more thorough statistical analysis of the quality of its predictions, which is necessary if the regression model is to be used as a reliable substitute for direct calculations of NLO cross sections.

In this thesis, we propose to perform Bayesian regression using neural networks to substitute direct calculations of NLO cross sections. Neural networks are universal function approximators [5] and are thus a robust mathematical model to employ for regression tasks that may need a large number of free parameters to learn the relationship between the targets and the features. Neural networks trained within the Bayesian framework is referred to as Bayesian neural networks (*roll credits*). Due to the large number of free parameters found in neural networks, using them in Bayesian regression tasks is a considerable challenge. The vast majority of their usage in the literature employ approximate strategies to infer parameters of the model. The main reason for this is that modern machine learning libraries such as TensorFlow or PyTorch provide highly optimized and modular frameworks for neural network models, and research have been conducted to create Bayesian alternatives which spend approximately the same amount of time learning its parameters per training example. Given a set of data examples, the distributions of the model parameters in the neural network are parameterized with a surrogate distribution, i.e. a Gaussian distribution for each parameter. The parameterization is adjusted when shown training examples to "learn" an approximation to the true distribution of the model parameters. Once a parameterization is learned, the model can be used to computed a predictive distribution of a target given an example of the independent variable. This is achieved by drawing samples from the learned distribution, usually by use of Markov chain Monte Carlo (MCMC) methods. The disadvantage is that the surrogate distributions are typically treated as a product of an independent distribution for each parameter of the neural network, which cannot properly capture the multi-modal nature of the exact high-dimensional distribution its parameters embody. The multi-modality is a result of the correlation between each parameter and is an inescapable consequence of the model class. Thus drawing samples of its parameters independently of each others will not properly encapsulate their underlying distribution. To make matters worse, the multi-modal distribution of its parameters also suffer from what is known as indentifiability because a particular neural network can undergo specific permutations or rescaling of its parameters, which yield many equivalent sets of parameters that all produce identical predictions. Treating its parameters as independent will therefore be a crude approximation. These features make Bayesian inference of its parameters in regressions tasks an obstacle that exacts a heavy toll, also in the case of drawing its parameters from its exact distribution, which is what our task will be.

We will explore the properties of Bayesian neural networks where its parameters are sampled from the *exact* posterior using MCMC methods. Important problems to investigate is the computational cost of the methods on different types of hardware such as a CPU and a GPU. The ability to correctly predict targets and yield reliable uncertainty estimates are especially imperative to replace direct calculations of NLO cross sections. Exploring the exact distribution of neural network parameters is also of interest to evaluate the degree to which its distribution can be approximated with parameterized surrogate distributions. This will be *some* of the main concerns in this thesis.

In the next chapter, we shall formalize the notion of a Bayesian regression and Bayesian machine learning precisely.

Chapter 2

Bayesian Formulation of Machine Learning

In this chapter we will introduce the notion of *Bayesian machine learning* (Bayesian ML). We will start from the classical view of ML and reformulate it in terms of Bayesian concepts. We will only concern ourselves with so-called supervised ML models used to solve supervised regression tasks as it is the only class of problems of interest in this thesis. We will first introduce the core of ML and its constituent ingredients. From this we transition to Bayes' theorem and a Bayesian framework for ML. Finally we discuss Bayesian inference.

2.1 The Core of Machine Learning

The basic conceptual framework of a supervised machine learning problem is as follows. Assume a dataset D is a sequence of N datapoints $D = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, where $x^{(i)} \in \mathbb{R}^p$ is the set of *features* and $y^{(i)} \in \mathbb{R}^d$ is the *target.* The next ingredient is to assume the targets can be decomposed as

$$y = f(x) + \delta, \tag{2.1}$$

for some true function $f : \mathbb{R}^p \to \mathbb{R}^d$ (also known as the *ground truth*), where $\delta \in \mathbb{R}^d$ is introduced to account for random noise. The objective is to learn $f(x)$ from the dataset. To this end, we choose a *model class* $\hat{f}(x;\theta)$ parameterized by a model parameters $\theta \in \mathbb{R}^m$, combined with a procedure to infer an estimate of the parameters $\hat{\theta}$ such that the model is as close to $f(x)$ as possible. Formally, this means choosing a *metric* $\mathcal{L}$ to quantify the error, called a *loss* function (or a *cost* function, but we will adopt the former term in line with the terminology used in the TensorFlow framework), and minimize it with respect to the parameters of the model to obtain $\hat{\theta}$ using an optimization algorithm. For brevity, we will denote the output of a model class as $\hat{y}^{(i)} \equiv \hat{f}(x^{(i)};\theta)$.

2.1.1 Loss Functions

For regression problems, two loss functions $\mathcal{L}$ are commonly chosen. The first is the *residual sum of squares* (RSS) given by

$$\mathcal{L}_{\text{RSS}} \equiv \text{RSS} = \sum_{i=1}^{N} \left\| y^{(i)} - \hat{y}^{(i)} \right\|_2^2, \tag{2.2}$$

where $\|\cdot\|_2$ denotes the L^2-norm. The second is the the *mean squared error* (MSE), defined as

$$\mathcal{L}_{\text{MSE}} \equiv \text{MSE} = \frac{1}{N} \sum_{i=1}^{N} \left\| y^{(i)} - \hat{y}^{(i)} \right\|_2^2. \tag{2.3}$$

For optimization purposes, they yield equivalent optimal parameters $\hat{\theta}$, at least in principle.

2.1.2 Regularization

With datasets of limited size, *overfitting* can pose a problem, yielding models that generalize poorly because they become overly specialized to the dataset on which $\hat{\theta}$ is inferred. The implication is that the predicted target on unseen data is unlikely to be correct. This occurs especially if the model is too complex. One strategy to overcome this, is to tack on a regularization term to the loss-function. By *regularization*, we mean an additional term that limits the size of the allowed parameter space. Hence, regularization imposes a constraint on the optimization problem.

The two most commonly used regularization terms are L^2-regularization, which adds a term to the loss function as

$$\mathcal{L} = \mathcal{L}_0 + \frac{\lambda}{2}\|\theta\|_2^2, \tag{2.4}$$

where λ is the so-called *regularization strength*, which is what we call a *hyperparameter*, and $\mathcal{L}_0$ is a loss function with no regularization term. The second is L^1-regularization, which yields a loss

$$\mathcal{L} = \mathcal{L}_0 + \frac{\lambda}{2}\|\theta\|_1. \tag{2.5}$$

The terms *penalize* large values of θ, effectively shrinking the allowed parameter space. The larger the value of the regularization strength λ, the smaller the allowed parameter space becomes.

More generally, we can decomposed our full loss function as

$$\mathcal{L}(x, y, \theta) = \mathcal{L}_0 + R(\lambda_1, \dots, \lambda_r, \theta), \tag{2.6}$$

where $R(\theta)$ is a linear combination of L^p-regularization terms where λ_i are the expansion coefficients which are all treated as hyperparameters. L^p-regularization terms is defined by the L^p-norm

$$\|x\|_p = \left(|x_1|^p + \cdots + |x_m|^p\right)^{1/p}, \quad x \in \mathbb{R}^m. \tag{2.7}$$

In practice, we typically use a single form of L^p-regularization but nothing stops us from constructing complicated regularization terms in theory.

2.1.3 Optimization

Once a model class and loss function is chosen, an *optimizer* or *optimization algorithm* must be chosen. By this, we mean an algorithm that uses the loss function and the model class, and minimizes the loss with respect to the model parameters to yield an estimate of $\hat{\theta}$. Regardless of which optimization algorithm we employ, we seek

$$\hat{\theta} = \arg\min_\theta \mathcal{L}. \tag{2.8}$$

In this thesis, optimization plays a smaller role in the inference of model parameters than in classical ML because we do not seek a single estimate $\hat{\theta}$ in most Bayesian applications. We shall nevertheless utilize such algorithms for some parts but for another purpose. One of the most popular optimizers in the deep learning community is ADAM [6] which we will mainly use when optimization is needed.

2.2 Bayes' theorem

Our goal is to reformulate ML in terms of Bayesian concepts. The backbone of Bayesian ML is *Bayes' theorem* [7]. The theorem can be formulated as

$$p(\theta|D) = \frac{p(D|\theta)p(\theta)}{p(D)}, \tag{2.9}$$

where D is observed data and θ denotes the parameters of the model. Here $p(\theta)$ is called the *prior* distribution and embodies our prior knowledge of θ before any new observations are considered. $p(D|\theta)$ is called the *likelihood* function and provides the relative probability of observing D for a fixed value of θ. It need not be normalized to unity, which is why it only provides relative "probabilities". The *posterior* distribution $p(\theta|D)$ models our belief about θ after the data D is observed. Finally, $p(D)$ is called the *evidence* which we may regard as the normalization constant of the posterior such that posterior integrates to unity over parameter space. In the context of Bayesian ML, the evidence will not be an interesting quantity as it will not turn up as part of any algorithms. Moreover, it is typically intractible for sufficiently large parameter spaces. It is therefore common to write Bayes' theorem as

$$p(\theta|D) \propto p(D|\theta)p(\theta), \tag{2.10}$$

which we too shall adopt.

2.3 Bayesian Framework for Machine Learning

The Bayesian framework for ML differs somewhat in approach to its classical counterpart. We define a model class in the same way as before. Choosing a loss function is substituted with choosing a likelihood function and a prior. Minimization of the loss function is replaced with maximization of the likelihood function or the posterior distribution. In fact, The Bayesian framework introduces several ways to infer an estimate for the optimal model parameters [8].

1. *Maximum Likelihood Estimation* (MLE): The optimal parameters $\hat{\theta}$ are inferred by

$$\hat{\theta} = \arg\,\max_\theta\, p(D|\theta), \tag{2.11}$$

meaning we choose $\hat{\theta}$ as the mode of the likelihood function. This is equivalent to maximizing the log-likelihood (since log is a monotonic function), i.e.

$$\hat{\theta} = \arg\,\max_\theta \log p(D|\theta). \tag{2.12}$$

2. *Maximum-A-Posteriori* (MAP): This estimate of $\hat{\theta}$ is defined as

$$\hat{\theta} = \arg\,\max_\theta\, p(\theta|D), \tag{2.13}$$

meaning we choose $\hat{\theta}$ as a mode of the posterior distribution.

3. *Bayes' estimate*: The estimate of $\hat{\theta}$ is chosen as the expectation of the posterior,

$$\hat{\theta} = \mathbb{E}_{p(\theta|D)}[\theta] = \int \mathrm{d}\theta\; \theta p(\theta|D). \tag{2.14}$$

The connection between classical and Bayesian ML can be understood from what follows. First, let us assume that each datapoint $(x^{(i)}, y^{(i)})$ is identically and independently distributed (i.i.d.). The likelihood function can then generally be written as

$$P(D|\theta) = \prod_{i=1}^{N} P(y^{(i)}|x^{(i)}, \theta). \tag{2.15}$$

For regression tasks, the standard choice of likelihood function is the *Gaussian*

$$p(y|x,\theta) = \exp\left(-\frac{1}{2\sigma^2}\left\|y - \hat{f}(x;\theta)\right\|_2^2\right), \tag{2.16}$$

where σ is some hyperparameter typically chosen to be the same for every datapoint (x, y). For the full dataset, we get

$$p(D|\theta) = \prod_{i=1}^{N} \exp\left(-\frac{1}{2\sigma^2}\left\|y^{(i)} - \hat{f}(x^{(i)};\theta)\right\|_2^2\right). \tag{2.17}$$

Now, consider the definition of MLE from eq. (2.11). It instructs us to maximize the expression in eq. (2.17). If we rewrite the likelihood function a bit

$$p(D|\theta) = \exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{N}\left\|y^{(i)} - \hat{f}(x^{(i)};\theta)\right\|_2^2\right), \tag{2.18}$$

we can observe that maximization of the likelihood function simply amounts to minimization of the RSS and hence of the MSE, as can be seen by comparison with the expressions in eq. (2.2) and eq. (2.3).

We can go even further, by considering the MAP estimate. Let us introduce a Gaussian prior on the parameters such that

$$p(\theta) \propto \exp\left(-\frac{\lambda}{2}\|\theta\|_2^2\right). \tag{2.19}$$

The posterior obtained from Bayes' theorem in eq. (2.10) by combining the prior introduced in eq. (2.19) and the likelihood function in eq. (2.17) is

$$p(\theta|D) \propto p(D|\theta)p(\theta) \propto \prod_{i=1}^{N} \exp\left(-\frac{1}{2\sigma^2}\left\|y^{(i)} - \hat{f}(x^{(i)};\theta)\right\|_2^2\right)\exp\left(-\frac{\lambda}{2}\|\theta\|_2^2\right), \tag{2.20}$$

which we can rewrite as

$$p(\theta|D) \propto \exp\left(-\left[\frac{1}{2\sigma^2}\sum_{i=1}^{N}\left\|y^{(i)} - \hat{f}(x^{(i)};\theta)\right\|_2^2 + \frac{\lambda}{2}\|\theta\|_2^2\right]\right). \tag{2.21}$$

Maximization of this expression is equivalent to minimization of RSS or MSE with a L^2-regularization term tacked on which can be seen by comparison with eq. (2.4). Obviously, we are missing a factor $1/N$ in front of the likelihood term which can be thought of as baked into the σ parameter. The natural generalization is that the posterior can be expressed as

$$p(\theta|D) \propto \exp\left(-\mathcal{L}\right), \tag{2.22}$$

for any loss function as in eq. (2.6). For a purpose that comes much later when we discuss Hamiltonian Monte Carlo, we can invert eq. (2.22)

$$\mathcal{L} = -\log Z - \log p(D|\theta) - \log p(\theta), \tag{2.23}$$

for some appropriate normalization constant Z. Assuming that the dataset consists of observations that are i.i.d, we get

$$\mathcal{L} = -\log Z - \sum_{i=1}^{N} p(y^{(i)}|x^{(i)},\theta) - \log p(\theta). \tag{2.24}$$

Equation (2.24) will play an important role later on.

2.4 Bayesian Inference

We have seen that there is a straight forward connection between the Bayesian framework and the classical view of ML by looking at estimators $\hat{\theta}$. In regression tasks, however, we are seldom interested in a single estimate of the model parameter. Instead we seek to obtain the posterior distribution from which we can infer other quantities. In applications where the model class is sufficiently complex, direct computation of the posterior is not feasible. Instead, we must settle with an approximate posterior distribution which we construct using Monte Carlo Markov chains (MCMC) methods. The discussion of such methods is allocated to chapter 3. For now we assume that there exists a way to generate samples $\theta \sim p(\theta|D)$. We approximate the posterior by sampling a set of model parameters $\{\theta^{(1)}, \ldots, \theta^{(n)}\}$ where $\theta^{(t)} \sim p(\theta|D)$, yielding an *empirical* posterior distribution.

We will primarily use the posterior to compute two classes of mathematical objects. The first is the *predictive distribution* of a target y^* given an input x^*. The predictive distribution can be expressed as

$$p(y^*|x^*, D) = \int \mathrm{d}\theta \; p(y^*|x^*, \theta) p(\theta|D). \tag{2.25}$$

Equation (2.25) is generally intractible since we cannot exactly compute the posterior. The predictive distribution is therefore approximated by generating a set of predictions using the empirical posterior distribution. That is, we indirectly sample from $p(y^*|x^*, D)$ by computation of $\hat{f}(x^*; \theta^{(t)})$ for $t = 1, \ldots, n$. In other words, the empirical predictive distribution is generated as follows.

$$\begin{aligned} \theta^{(t)} &\sim p(\theta|D), \\ f(x^*; \theta^{(t)}) &\sim p(y^*|x^*, \theta). \end{aligned} \tag{2.26}$$

The second class is expectation values with respect to the posterior distribution, which for a target function $f(\theta)$ is defined as

$$\mathbb{E}_{p(\theta|D)}[f] = \int \mathrm{d}\theta \; f(\theta) p(\theta|D). \tag{2.27}$$

An important example of eq. (2.27) is the expectation value of the predictive distribution, which will be the expectation of the model class with respect to the posterior

$$\hat{y} \equiv \mathbb{E}_{p(\theta|D)}[\hat{f}(x; \theta)] = \int \mathrm{d}\theta \; \hat{f}(x; \theta) p(\theta|D). \tag{2.28}$$

Equation (2.27) must be approximated since we cannot hope to evaluate the posterior $p(\theta|D)$. Even if we could, we will be working with sufficiently large parameters spaces such that the integral itself is intractible in any case. Approximation of expectation values is done using MCMC methods which is the subject of the next chapter.

Chapter 3

Markov Chain Monte Carlo

In this chapter, we will discuss fundamental ideas pertaining to *Markov Chain Monte Carlo* (MCMC) methods. We shall confine the discussion to continuous sample spaces which is the kind needed in this thesis. We will commence with a discussion of expectation values and an important notion called the *typical set*. We will then define and discuss Markov chains and Markov transitions after which we shall discuss Metropolis-Hastings sampling and its limitations. Finally we will look at Gibbs sampling. We will adopt a geometric view where possible to provide a natural transition to Hamiltonian Monte Carlo and the No-U-Turn sampler in the two following chapters.

3.1 Expectation Values and the Typical Set

Consider a *target probability density* $\pi(\theta)$ and an m-dimensional sample space Θ where $\theta \in \Theta$. Consider $f(\theta)$ to be an arbitrary smooth function of θ. The *expectation value* of $f(\theta)$ with respect to the density $\pi(\theta)$ is then defined as

$$\mathbb{E}_\pi[f] = \int \mathrm{d}\theta\ \pi(\theta) f(\theta). \tag{3.1}$$

We shall interchangably refer to expectation values simply as *expectations*. We will call the function f we seek to compute the expectation of as the *target function*. For all but a few simple densities, evaluation of eq. (3.1) is impossible analytically. To complicate things further, numerical evaluation with numerical integration techniques of the expectation in high-dimensional spaces quickly becomes computationally infeasible as the dimensionality increases, due to limited computational resources. Even worse, we may not even be able to write down the expression of $\pi(\theta)$ explicitly. Fortunately, it is unlikely that the entire sample space contribute significantly to the expectation. If we could somehow pick out the points in sample space that *does* contribute, only knowing $\pi(\theta)$ up to a normalization constant, we could make approximate computations of expectations tractible.

For most purposes, we are interested in the expectation of more than a single target function. For example, in Bayesian applications, we are often interested in both the mean and variance of a quantity which introduces the need for several target functions. Thus the numerical method should not depend on the target function in question. Instead the focus should be laid on the contribution from $\pi(\theta)\mathrm{d}\theta$ to the integrand. The objective of MCMC methods is to efficiently sample points from regions of sample space where this quantity is non-neglible. This region of sample space is called the *typical set* [9].

3.1.1 The Typical Set

For simplicity, we can divide a sample space into three regions with respect to the target density $\pi(\theta)$.

1. High-probability density region. These are regions in the neighborhood of a mode of the target density. In general, as the dimensionality increases, the contribution from $\pi(\theta)\mathrm{d}\theta$ becomes neglible here unless the volume in the region is significant enough.
2. The typical set. This refers to the regions in which $\pi(\theta)\mathrm{d}\theta$ provides a non-neglible contribution to any expectation. This may be thought of as the high-probability region of the sample space since $\pi(\theta)\mathrm{d}\theta$ is proportional to probability of a volume $\mathrm{d}\theta$ in the neighborhood of θ.
3. Low-probability density regions. These are regions far away from any mode of the density. This region, too, will generally yield neglible contributions to the integrand even if the volume is large.

Although the notion of a typical set can be formalized precisely, we will intentionally operate with this somewhat imprecise definition. For our purposes, it suffices to use it merely as a conceptual notion to evaluate the quality of the samples generated by an MCMC chain.

3.1.2 The Target Density and Bayesian Applications

In the chapter on Bayesian ML, we mentioned that we could not compute the evidence term of Bayes' theorem in realistic applications and thus were only concerned with a proportionality relationship $p(\theta|D) \propto p(D|\theta)p(\theta)$. Thus any MCMC methods we are interested in cannot require that the $\pi(\theta)$ is normalized to unity. We only require that the density is smooth and that

$$0 < \int \mathrm{d}\theta \; \pi(\theta) < \infty. \tag{3.2}$$

Sometimes we may refer to the target density as the *target distribution*. In Bayesian applications, we assume that $\pi(\theta) = p(D|\theta)p(\theta)$ such that $p(\theta|D) \propto \pi(\theta)$.

3.2 Markov Chains and Markov Transitions

Since direct evaluation of eq. (3.1) in most applications is intractible, we seek to approximately evaluate it by generating samples $\theta^{(t)}$ from the typical set using *Markov chains*. A Markov chain is a sequence of points $\theta^{(1)}, \theta^{(2)}, ..., \theta^{(n)}$ generated sequentially using a random map called a *Markov transition*. A Markov transition is a conditional probability density $T(\theta'|\theta)$ that yields the probability of transition from a point θ to θ'. The Markov transition is also called a *Markov kernel* which is a special case of a *transition kernel*. The latter is the term we will adopt because it is the term used by TensorFlow Probability.

An arbitrary transition kernel is not useful because the generated Markov chain is unlikely to have any relation to the target distribution of interest. To generate a useful Markov chain, we must use a transition kernel that preserves the target distribution. The condition that ensures this is

$$\pi(\theta) = \int d\theta' \pi(\theta')T(\theta|\theta'). \tag{3.3}$$

The condition is formally called *detailed balance*. The interpretation of the condition is that the Markov chain is reversible.

We can start from any θ and use the transition kernel to produce a set of new states. The distribution generated by the Markov chain should be distributed according the target distribution regardless of which point we used to generate the chain from, given a long enough chain. A more important fact is that as long as this condition is satisfied, the Markov chain will converge to and stay within the typical set.

The standard approach to approximate eq. (3.1) is then with the MCMC *estimator*

$$\hat{f}_N = \frac{1}{N}\sum_{t=1}^{N} f(\theta^{(t)}). \tag{3.4}$$

For large enough N, the estimator can be shown to converge to the true expectation such that $\lim_{N\to\infty} \hat{f}_N = \mathbb{E}_\pi[f]$. Obviously, the knowledge that the estimator will asymptotically converge to the true expectations is of limited use when restricted to a practical computation in which only a finite chain can be generated. We must therefore understand the properties of finite Markov chains so we can efficiently use them to approximate eq. (3.1).

3.2.1 Ideal Markov Chains

In order to understand the behaviour of finite Markov chains, we should first consider the behaviour of ideal Markov chains. An ideal Markov chain can be divided into three phases.

1. A convergence phase. The Markov chain is initiated from some point θ and the initially generated sequence lies in a region outside the typical set. Estimators evaluated using this part of the sequence are highly biased, meaning inclusion of these points will lead to an estimator that lies relatively far away from the true expectation.

2. An exploration phase. The Markov chain has reached the typical set and begins its first "traversal" of it. In this phase, estimators will rapidly converge towards the true expectations.

3. A saturation phase. At this point, the Markov chain has explored most of the typical set and convergence of the estimators slow down significantly.

The ideal evaluation of estimators thus only use the parts of the Markov chain generated in the second and third phase, discarding the the chain generated in the first phase. The notion of discarding the chain from the first phase is called *burn-in* or *mixing*. To most efficiently approximate eq. (3.1), we should really only use points generated in the exploration phase. Using points from the saturation phase does not hurt the estimators but yield diminishing returns with respect to computational resources.

3.2.2 Pathologies

Unfortunately, many target distributions embody typical sets with pathological regions where *any* transition kernel that obey eq. (3.3) is not sufficient to *efficiently* explore the typical set. Geometrically, this can be regions in the typical set in which the target distribution rapidly changes. The pathological regions can be completely ignored by the chain for much of the exploration, leading to poor convergence and thus biased estimators. However, as long as the transition kernel satisfies detailed balance, we know for a fact that the estimators *must* converge eventually. Consequentially, the Markov chain will be stuck near pathological regions for long periods to compensate before it rapidly explores other parts of the typical set. This behaviour can be repeated, which makes estimators oscillate. Regardless of when the MCMC chain is terminated, the estimator will likely be biased due to this oscillating behaviour.

3.2.3 Geometric Ergodicity and Convergence Diagnostics

Generation of ideal Markov chains is guaranteed if the transition kernel satisfies *geometric ergodicity* [10], a *Central Limit Theorem* for the MCMC estimators. However, in most cases it is impossible to check that the condition is satisfied. Instead one uses a statistical quantitiy known as the *potential scale reduction factor* $\hat{R}$ [11]. The ideal value is $\hat{R} = 1$. For values far away from this target, it is unlikely that geometric ergodicity is satisfied. The Rule-of-thumb is to assume convergence if $\hat{R} < 1.1$ [12].

3.3 Metropolis-Hastings

Construction of a transition kernel that ensures convergence to the typical set of the target distribution is a non-trivial problem in general. Fortunately, the Metropolis-Hastings algorithm provides a general solution that lets us construct a transition kernel with this property [13, 14]. The algorithm consist of two components; a proposal of a new state and a correction step called the *Metropolis correction*. Given a state θ, we propose a new state θ' by adding a random perturbation to the initial state. The correction step rejects a proposed state that moves away from the typical set of the target distribution and accepts proposals that stay within it. The proposed state is formally sampled from a *proposal distribution* $q(\theta'|\theta)$. The probability of accepting the proposed state given the initial state, fittingly called the *acceptance probability*, is

$$a(\theta'|\theta) = \min\left(1, \frac{q(\theta|\theta')\pi(\theta')}{q(\theta'|\theta)\pi(\theta)}\right). \tag{3.5}$$

A particularly neat feature of the acceptance probability in eq. (3.5) is that it can be calculated in Bayesian applications because the evidence term cancels out. These steps are summarized in algorithm 3.1.

Algorithm 3.1 Metropolis-Hastings

```
function MetropolisHastings($\theta$)
    Sample $\theta' \sim q(\theta'|\theta)$
    $a(\theta'|\theta) \leftarrow \min\left(1, \frac{q(\theta|\theta')\pi(\theta')}{q(\theta'|\theta)\pi(\theta)}\right)$
    Sample $u \sim \text{Uniform}(0,1)$.
    if $a(\theta'|\theta) \geq u$ then
        $\theta \leftarrow \theta'$                                    ▷ Accept transition
    else
        $\theta \leftarrow \theta$                                     ▷ Reject transition
    end if
    return $\theta$
end function
```

3.3.1 The Proposal Distribution

There are many valid choices of proposal distributions. A common choice is a Gaussian distribution $q(\theta'|\theta) = \mathcal{N}(\theta'|\theta, \Sigma)$, where Σ is the *covariance matrix* of the normal distribution used to generate the perturbation of the initial state. This is typically chosen to be the identity matrix $\Sigma = I$.

We will refer to the Metropolis-Hastings algorithm with this proprosal distribution as *random walk Metropolis*. More precisely, this means that a proposed state is given by

$$\theta' = \theta + \delta, \tag{3.6}$$

where $\delta \sim \mathcal{N}(0, \Sigma)$. This distribution is symmetric such that $q(\theta'|\theta) = q(\theta|\theta')$, implying that the acceptance probability reduces to

$$a(\theta'|\theta) = \min\left(1, \frac{\pi(\theta')}{\pi(\theta)}\right). \tag{3.7}$$

Hence, evaluation of the acceptance probability only require that we evaluate the target distribution at the initial state and the proposed state.

The random walk Metropolis algorithm does suffer from slow convergence to, and exploration of, the typical set in high-dimensional spaces. This can be understood because of the following. As we increase the dimension of the sample space, the volume outside of the typical set becomes increasingly larger than the volume of the typical set itself. This implies with increasing probability that a random perturbation of an arbitrary initial state will cause the proposed state to lie outside the typical set for a fixed covariance matrix. We can compensate for this flaw by reducing the values of Σ_{ij}, but this will slow down exploration of the sample space. The slow exploration also leads to a Markov chain where consecutive samples embody a relatively large measure of correlation. The quality of the resulting Markov chain tarnishes and successive samples must be discarded in order to properly evaluate eq. (3.1). This process of discarding correlated successive samples in a Markov chain is called *thinning*. Fortunately, there exists a solution; *gradient-informed* exploration of the sample space, manifested in the form of *Hamiltonian Monte Carlo*. This algorithm is a special case of a Metropolis-Hastings algorithm in which the proposal distribution $q(\theta'|\theta)$ is a special one utilizing Hamiltonian dynamics and Gibbs sampling to produce a new proposal state θ'. This is the topic of the next chapter.

3.4 Gibbs Sampling

The final standard MCMC algorithm we need is the *Gibbs* sampler. It plays a small part of the sampling in HMC and so we should therefore briefly discuss it. It is a MCMC sampling method used for multi-variate probability densities, and so is only meaningful to discuss for $d > 1$ dimensions. Suppose $\theta^{(t)}$ represents the parameters at iteration t. The next sample $\theta^{(t+1)}$ in the Markov chain is drawn according to some chosen conditional distribution p depending on the previous and current sample as follows

$$\theta_i^{(t+1)} \sim p(\theta_i|\theta_1^{(t+1)}, \dots \theta_{i-1}^{(t+1)}, \theta_{i+1}^{(t)}, \dots, \theta_m^{(t)}). \tag{3.8}$$

We may summarize this as a function in algorithm 3.2 which given an initial state $\theta^{(t)}$ returns a new state $\theta^{(t+1)}$ sampled according to eq. (3.8).

Algorithm 3.2 Gibbs sampling

```
function Gibbs($\theta^{(t)}$)
    for $i = 1, \dots, d$ do
        Sample $\theta_i^{(t+1)} \sim p(\theta_i|\theta_1^{(t+1)}, \dots \theta_{i-1}^{(t+1)}, \theta_{i+1}^{(t)}, \dots, \theta_m^{(t)})$.
    end for
return $\theta^{(t+1)} = \left(\theta_1^{(t+1)}, \dots, \theta_m^{(t+1)}\right)$.
end function
```

Chapter 4

Hamiltonian Monte Carlo

In this chapter, we will explore the details of Hamiltonian Monte Carlo. It is a Markov chain Monte Carlo method that uses gradient-informed steps to generate a proposal state for Metropolis correction. This is achieved by usage of Hamiltonian dynamics which allow gradient-informed exploration by treating the model parameters as "coordinates" of a fictitous physical system, and introducing auxilliary variables representing its momenta. The coordinates and momenta are required to obey a particular set of coupled differential equations called Hamilton's equations. The differential equations cannot in general be solved exactly and are instead simulated. The particular kind of numerical method used to achieve this is called the Leapfrog integrator. At the end of a simulation, a new set of coordinates and momenta will be generated, which is regarded as the proposal state to undergo Metropolis correction. If accepted, we keep the proposed coordinates as the next parameter in the Markov chain. Otherwise, the initial coordinates assume this role. The auxilliary momenta is discarded and resampled on each iteration as they play no important role for the actual Markov chain.

We begin by presenting Hamiltonian dynamics and the Leapfrog integrator. Once established we show how the framework is used to construct an MCMC method. Next, we will see how to apply the method to Bayesian machine learning models before we finalize the chapter with a discussion on some limitations of the method.

4.1 Hamiltonian Dynamics

Hamiltonian dynamics [15] is a formulation of classical mechanics that allows us to compute the time evolution of a physical system. The fundamental mathematical object of the theory is the *Hamiltonian* H which governs the time evolution of the *coordinates* $q = (q_1, \ldots, q_d)$ and *momenta* $p = (p_1, \ldots, p_d)$ of the system. The $2d$-dimensional space defined by the points (q, p) is called *phase-space*. The precise relationship is formulated by *Hamilton's equations*

$$\frac{\mathrm{d}q_i}{\mathrm{d}t} = \frac{\partial H}{\partial p_i}, \qquad \frac{\mathrm{d}p_i}{\mathrm{d}t} = -\frac{\partial H}{\partial q_i}, \quad \text{for} \quad i = 1, \ldots, d. \tag{4.1}$$

The objective is to use the $2d$ coupled differential equations in eq. (4.1) to find $(q(t), p(t))$ given some initial condition $(q(0), p(0))$ where t represents time. A system governed by Hamilton's equations is called a *Hamiltonian system*. For the purpose of constructing a MCMC method, we need not consider the most general theory of Hamiltonian dynamics and we will therefore refrain from doing so. We shall confine our focus to Hamiltonians which can be decomposed as

$$H(q, p) = V(q) + K(p), \tag{4.2}$$

where V is the *potential energy* and K is the *kinetic energy* of the system. The particular kind of Hamiltonian in eq. (4.2) corresponds to the total energy of the system. A key feature is that this

Hamiltonian is *conserved* through time. This observation follows from

$$\frac{\mathrm{d}H}{\mathrm{d}t} = \sum_i \left(\frac{\mathrm{d}q_i}{\mathrm{d}t}\frac{\partial H}{\partial q_i} + \frac{\mathrm{d}p_i}{\mathrm{d}t}\frac{\partial H}{\partial p_i} \right) = \sum_i \left(\frac{\partial H}{\partial p_i}\frac{\partial H}{\partial q_i} - \frac{\partial H}{\partial q_i}\frac{\partial H}{\partial p_i} \right) = 0. \tag{4.3}$$

Thus any solution $(q(t), p(t))$ will be confined to a hyperplane defined by the Hamiltonian and the initial condition.

Evolving a Hamiltonian system from some initial point $(q(0), p(0))$ is in general a non-trivial task. Exact solutions can only be computed for simple systems. To arm ourselves with a robust MCMC method, then, we must employ a numerical method to approximate the solutions. To this end, there is a class of numerical methods called *symplectic integrators* that take advantage of the underlying geometry enforced by Hamilton's equations which allow accurate solutions over long time periods at a lower computational cost than typical higher-order methods such as fourth-order Runge-Kutta. The particular symplectic integrator used in HMC is called the *Leapfrog integrator* which we shall discuss next.

4.1.1 Leapfrog integration

The Leapfrog integrator [16] is used in HMC to integrate eq. (4.1) to generate new proposal states. First assume that we *discretize* the time-coordinate t into discrete time coordinates defined by an initial time t_0 and a *step size* ϵ which defines the distance between each time coordinate. The k-th time coordinate can be generated by

$$t_k = t_0 + k\epsilon. \tag{4.4}$$

To please mathematicians, we introduce functions $\hat{q}$ and $\hat{p}$ to represent the discretized approximations to the exact solution $(q(t), p(t))$. From an initial point $(q(t_0), p(t_0))$, we simulate the system to obtain approximate values of the exact solution at discrete times $t_1, \dots, t_n$.

Consider a single Leapfrog step from a point $(\hat{q}(t), \hat{p}(t))$. Its approximation to $(q(t+\epsilon), q(t+\epsilon))$ is then computed as formulated in algorithm 4.1.

Algorithm 4.1 Leapfrog Integration

```
function Leapfrog($V, q, p, \epsilon$)
    for $i = 1, \dots, d$ do
        $p_i' \leftarrow p_i - \frac{\epsilon}{2}\frac{\partial V(q)}{\partial q_i}$
        $q_i' \leftarrow q_i + \frac{\epsilon}{m_i} p_i'$
        $p_i' \leftarrow p_i' - \frac{\epsilon}{2}\frac{\partial V(q')}{\partial q_i}$
    end for
    return $(q', p')$
end function
```

Note the introduction of the *masses* m_i. For now they may simply be regarded as some constants belonging to the Hamiltonian system. When used in HMC, it is common to set all masses $m_i = 1$ from which we can formulate the algorithm in vectorized form, as seen in algorithm 4.2.

Algorithm 4.2 Vectorized Leapfrog Integration

```
function VectorizedLeapfrog($V, q, p, \epsilon$)
    $p' \leftarrow p - \frac{\epsilon}{2}\nabla_q V(q)$
    $q' \leftarrow q + \epsilon p'$
    $p' \leftarrow p' - \frac{\epsilon}{2}\nabla_q V(q')$
    return $(q', p')$
end function
```

4.2 Generating a Proposal State

Our next objective is to understand how we connect an arbitrary target distribution $\pi(\theta)$ to Hamiltonian dynamics. In this section we will weave these together and show how we generate a new proposal state θ' which will undergo a Metropolis correction. The results discussed in this section can be understood as representing the proposal distribution $q(\theta'|\theta)$ used during the Metropolis-Hastings step, as we summarized in algorithm 3.1.

The fundamental assumption we make is that the target distribution can be expressed in terms of a canonical distribution over coordinate space

$$\pi(q) \propto \exp\left\{-V(q)\right\}, \tag{4.5}$$

where q represents the model parameters θ. We will stick to this convention to avoid confusion and utilize the formulation of Hamiltonian dynamics discussed hitherto. Once we want to apply it in a Bayesian ML context, we simply replace $q \to \theta$. From eq. (4.5), we can find the potential energy function in terms of the target distribution

$$V(q) = -\log \pi(q), \tag{4.6}$$

up to a constant. Hence once the target distribution is known, we use eq. (4.6) to obtain the potential energy of the system.

We now turn to the problem of constructing the Hamiltonian so we can utilize Hamilton's equations. To achieve this, we must introduce *auxilliary* momenta p so we can define a kinetic energy function and evolve the system through what we may regard as *fictitious time* t. The momenta are sampled from some distribution of our own choice. We can proceed in the same way as we did with the potential energy function and express the momentum distribution in terms of a canonical distribution over momentum space

$$\pi(p) \propto \exp\left\{-K(p)\right\}, \tag{4.7}$$

such that

$$K(p) = -\log \pi(p), \tag{4.8}$$

up to a constant. The commonly chosen expression for kinetic energy is the one found in classical physics

$$K(p) = \sum_{i=1}^{d} \frac{p_i^2}{2m_i}, \tag{4.9}$$

from which the canonical distribution is inferred to be

$$\pi(p) \propto \exp\left\{-\sum_{i=1}^{d} \frac{p_i^2}{2m_i}\right\} = \prod_{i=1}^{d} \exp\left\{-\frac{p_i^2}{2m_i}\right\}. \tag{4.10}$$

Hence, with the kinetic energy from eq. (4.9), we sample each momentum independently from a Gaussian distribution with zero mean and variance $\sigma_i^2 = m_i$.

Now that we understand how we specify the potential energy for a given target distribution and the kinetic energy of the auxilliary momenta, we can formulate the full canonical distribution over phase-space as

$$\pi(q, p) = \pi(q)\pi(p) \propto \exp\left\{-V(q)\right\} \exp\left\{-K(p)\right\} = \exp\left\{-H(q, p)\right\}. \tag{4.11}$$

We are naturally just interested in generating a new coordinate q'. Using Hamilton's equations with the Hamiltonian implied by eq. (4.11), we can simulate the fictitious Hamiltonian system using Hamilton's equation in eq. (4.1) to generate a new state (q', p'). The proposal state is then obtained by the projection map $(q', p') \mapsto q'$.

As stated in the beginning of this section, we may regard the details outlined here as an elaborate explanation of the proposal distribution $q(\theta'|\theta)$. The final keypoint to consider is how we can make it symmetric so that we only need to evaluate $\pi(q', p')$ at the Metropolis step using eq. (3.7). It can be shown that we only need two additional steps. We must randomly choose to sample forwards or backwards in time. The second step is the negate the momenta at the end of the generation of the state, $p \mapsto -p$. The acceptance probability can then be computed as

$$a = \min\left(1, \frac{\pi(q', p')}{\pi(q, p)}\right) = \min\left(1, \exp\left\{-\left[H(q', p') - H(q, p)\right]\right\}\right). \tag{4.12}$$

But this should always be evaluated to $a = 1$ if H is indeed conserved. But the catch is that the dynamics is only approximated using the Leapfrog integrator. The best the integrator can do is conserve H on average, with its value oscillating about the initial value.

Before we summarize the algorithm in a neat manner, we shall briefly outline it conceptually.

1. Given an initial state q, we randomly sample the auxilliary momenta p from the distribution in eq. (4.10) to generate an initial condition (q, p) to use with Hamilton's equations.

2. We randomly choose to simulate the system forwards or backwards in time by sampling a variable $v \sim \text{Uniform}(\{-1, 1\})$ from which the step size is set as $v\epsilon$. Forwards in time is represented by $v = 1$ and backwards in time is represented by $v = -1$.

3. Perform L Leapfrog steps using algorithm 4.1 for a total trajectory length of ϵL to produce a proposal point (q', p').

4. Perform a Metropolis-Hastings correction on the proposal state to accept or reject it.

5. Project the phase-space point onto coordinate space and return q' if accepted, or q if rejected, in the previous step.

This essentially summarizes the practical steps of HMC. The introduction of randomly simulating forwards and backwards in time is to ensure that the algorithm is reversible and obeys the detailed balance condition discussed in chapter 3. To please mathematicians once more, we must really reverse the sign of the final momenta as well, but since we shall use a Gaussian distribution, changing the sign of the momenta makes no difference to the value of the kinetic energy. To generate a Markov chain by this procedure, we simply feed the returned coordinate state back in to the machinery and reiterate. The HMC scheme is summarized in algorithm 4.3.

Algorithm 4.3 Hamiltonian Monte Carlo

```
function HMCstep($q, H, L, \epsilon$)
    Sample $p \sim \mathcal{N}(0, \mathrm{diag}(m_1, \ldots, m_d))$    ▷ Sample auxilliary momenta
    Sample $v \sim \mathrm{Uniform}(\{-1, 1\})$.    ▷ Randomly choose direction in time.
    $(q', p') \leftarrow (q, p)$    ▷ Initialize the initial state.
    for $l = 1, ..., L$ do    ▷ Simulate Hamiltonian system for $L$ Leapfrog steps.
        $(q', p') \leftarrow$ Leapfrog($q', p', v\epsilon$)
    end for
    $a = \min\left(1, \exp\left\{-\left[H(q', p') - H(q, p)\right]\right\}\right)$    ▷ Compute acceptance probability
    Sample $u \sim U(0, 1)$    ▷ Uniform distribution on $(0, 1)$.
    if $a \geq u$ then    ▷ Perform Metropolis-Hastings correction
        $q \leftarrow q'$    ▷ Accept proposed state.
    else
        $q \leftarrow q$    ▷ Reject proposed state.
    end if
    return $q$
end function
```

4.3 The Potential Energy Function in Bayesian Machine Learning Applications

We seek to use HMC in a Bayesian ML application. It is therefore important to discuss a general way to construct the potential energy function in such applications. First, recall from chapter 2 in eq. (2.22) that the posterior could in general be written as

$$p(\theta|D) \propto \exp\left\{-\mathcal{L}(\theta)\right\}, \tag{4.13}$$

where $\mathcal{L}$ was some loss function in the classical ML sense. However, we do not need the evidence term and simply sample from the target distribution $\pi(\theta) = p(D|\theta)p(\theta)$ instead. Comparison with eq. (4.5) makes it clear that the potential energy function simply is $\mathcal{L}$. Combining this with eq. (2.24), lets us conclude that the general expression for the potential energy is

$$\mathcal{L} = -\log p(D|\theta) - \log p(\theta), \tag{4.14}$$

up to a constant. If we assume all N datapoints are i.i.d. we can recast it as eq. (2.24), that is

$$\mathcal{L} = -\sum_{i=1}^{N} \log p(y^{(i)}|x^{(i)}, \theta) - \log p(\theta). \tag{4.15}$$

4.4 Limitations of Hamiltonian Monte Carlo

Although HMC is effective at exploring the state space we wish to sample from, it suffers from the need to hand-tune the trajectory length ϵL. Poor choices of ϵ and L can lead to poor results. On one hand, if the trajectory length is too short, exploration of the state space will be limited which makes HMC behave like a random-walk. Suppose we fix the trajectory length to a finite, but sufficiently large value. If the step size ϵ is too large, it can lead to instabilities in the leapfrog integrator, while if its chosen to be too small, it will perform far too many iterations to make the algorithm worthwhile. Tuning these parameters requires preliminary runs for the problem at hand and analysis of so-called *trace statistics*, which essentially measures the quality of the generated Markov chain.

In the next chapter, we will look at algorithms that adaptively sets the trajectory length of HMC, namely the No-U-Turn sampler combined with dual-averaging of the step size, which allows us to overcome these limitations and more effectively sample from the target distribution without the need for hand-tuning and analysis of trace statistics, or reliance on heuristics.

Chapter 5

Adaptive Hamiltonian Monte Carlo

Hamiltonian Monte Carlo is considered a state-of-the-art sampler that efficiently explores sample space by producing large jumps to successive states with low correlation, but suffers the need for manual tuning of the trajectory length ϵL. In this chapter, we will explore improvements that adaptively adjust the trajectory length. This is achieved by means of adapting both the number of Leapfrog steps L using an improved sampler called the *No-U-Turn* (NUTS) sampler, and an adaptive scheme for setting the step size ϵ using a *dual averaging* algorithm. We will closely follow the treatment in the original paper [1] but adapt the notation to be consistent with the rest of this thesis.

We will start off with a discussion on how to adapt the number of Leapfrog steps using NUTS. At a high-level, NUTS starts from an initial state (q, p) and simulates the Hamiltonian dynamics of the system. This is done in the following way. Leapfrog steps are performed either forwards or backwards in time, first with a single Leapfrog step, then two Leapfrog steps, then followed by four Leapfrog steps and so on. This reiteration of the simulation is performed until the the path traced out in position space starts to double back towards itself. The states traced out can be regarded as a *balanced binary tree* $\mathcal{B}$ where each node represents a phase-space state produced by the Leapfrog integrator during the simulation. The next state of the Markov chain is sampled at random from these nodes.

We will end the chapter with the dual averaging scheme for adaptively setting the step size used with Leapfrog integrator. The algorithm is a modified version of a dual averaging scheme presented by Nesterov in [17].

5.1 The No-U-Turn Sampler

The No-U-Turn sampler generates a set of states we may regard as a balanced binary tree which we represent with the set $\mathcal{B}$. We shall explain the way it is built by starting from an initial point and building up the tree gradually before we generalize the procedure. An example of a trajectory in a 2-dimensional position space generated by NUTS is shown in figure 5.1. The initial state (q, p) is defined as the the node of the tree of depth $j = 0$. We sample a direction at random in time, either forwards $(v_0 = 1)$ or backwards $(v_0 = -1)$ and perform a single Leapfrog step to produce a new state (q', p') using the step size ϵv_0. This state represents its own little subtree of height $j = 0$ which is to be combined with the initial node to form a tree of height $j = 1$. If $v_0 = 1$, the new node is placed as the right half of the new tree. Conversely, if $v_0 = -1$, the new node is placed as the left half of the new tree. We repeat, but this time we double the number of Leapfrog steps to $L = 2$. We randomly sample the direction once more. If forwards in time $(v_1 = 1)$, we initiate the Leapfrog integrator from rightmost node of the current tree (which represents the head of the trajectory in position space). If backwards in time $(v_1 = -1)$, we feed the state of the leftmost node to the Leapfrog integrator (which represents the tail of the trajectory in position space) and integrate backwards in time. The new states produced with the Leapfrog integrator becomes the nodes of a subtree of height $j = 1$ which will be combined with the current tree. Again, if $v_1 = 1$, we place the new subtree as the right half of the combined tree. If $v_1 = -1$, it is placed as the left half of the combined tree. This procedure is carried out repeatedly. We draw a direction in time at random, and perform twice as many Leapfrog steps as the prior iteration from the rightmost node if forwards in time or the leftmost node if backwards in time to extend the trajectory further. More precisely, given a tree of height j,

1. Sample a direction $v_j \sim \text{Uniform}(\{-1, 1\})$ in time. Set the step size in the Leapfrog integrator as $\epsilon \to \epsilon v_j$.
2. Perform 2^j Leapfrog steps from the rightmost node if $v_j = 1$ or from the leftmost node if $v_j = -1$.
3. The new generated tree of height j is combined with the current tree of height j, producing a combined tree of height $j + 1$. If $v_j = 1$, the newly generated tree becomes the right half of the combined tree. If $v_j = -1$, it becomes the left half of the combined tree.

From a practical perspective, we cannot apply these steps repeatedly *ad-infinitum* of course. At some point, we must stop the doubling of the tree and select a node which from which we obtain the position state to take the next place in the Markov chain. How this is solved is what we shall consider next.

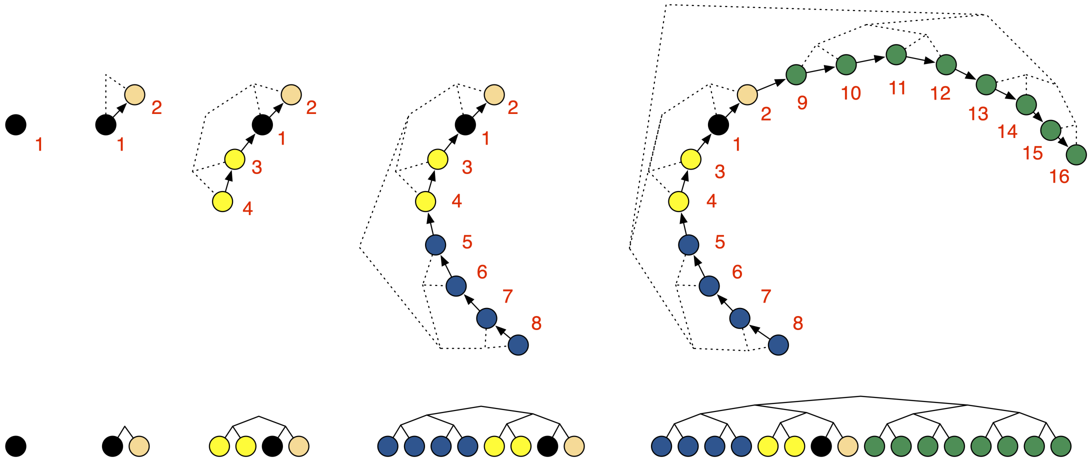


Figure 5.1: The figure shows an example of a trajectory generated by the NUTS sampler. The top diagram displays the projection onto position space with the momenta drawn in as arrows. The bottom diagram shows the resulting balanced binary tree. The tree structure is drawn onto the trajectory as well. The numbering displays the order in which the states are generated by Leapfrog integration. The black node is the initial node. The first doubling is forwards in time and yields the rightmost node of the first binary tree. The second doubling is backwards in time and is initiated from the black node, yielding a new tree of height 2 where the left subtree is the new states (the yellow nodes). The next doubling is also backwards in time, and the Leapfrog integrator is initiated from the tail (the leftmost yellow node) performing four Leapfrog steps generating a subtree which becomes the left half of the next tree (blue nodes). The final doubling in the figure is forwards in time with $L = 8$ Leapfrog steps taken from the orange node (which was the rightmost leaf of the tree before the final doubling) which yields the green nodes. The figure is a modified version of a diagram in [1].

5.1.1 Stopping Conditions and Selection of Candidate States

We seek a way to stop the doubling procedure that automatically does so when continuing is no longer beneficial from a computational standpoint. Let us first consider a point we stressed in chapter 4. If Hamilton's equations are solved exactly, the generated trajectory is confined to a hyperplane. Thus for an exact solution, the trajectory would at some point double back onto itself. This will likely also happen when we only approximate the solution with the Leapfrog integrator. If we continue the doubling procedure when the generated trajectory begins to double back on itself (perform a "U-turn"), we will revisit regions of parameter space that are already part of the binary tree, thus wasting computational resources. To avoid this, we ought to check during each doubling if such a "U-turn" occurs and terminate if it does. Consider an initial state (q, p) and let (q', p') be a point produced by the Leapfrog integrator (which is treated as a function of time). Then the change in the Euclidean distance between the positions are

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\|q'-q\|_2^2}{2} = (q'-q)^T\frac{\mathrm{d}}{\mathrm{d}t}(q'-q) = (q'-q)^T p'. \tag{5.1}$$

If eq. (5.1) evaluates to a negative number, continuing the simulation for an infinitesimal time $\mathrm{d}t$ will decrease the distance between the points which is how we can detect an occurence of a "U-turn". The way the NUTS sampler does this, is to consider the leftmost and rightmost node of any subtree of the current tree. Let (q^+, p^+) be the rightmost node and (q^-, p^-) the leftmost node of any subtree. Then if

$$(q^+ - q^-)^T p^+ < 0 \quad \text{or} \quad (q^+ - q^-)^T p^- < 0, \tag{5.2}$$

is fulfilled for any of the subtrees, it terminates the doubling of the tree. This is the so-called *No-U-Turn condition*. We must consider two distinct cases where the stopping condition in eq. (5.2) is met.

1. Consider a tree of height j. If, during the doubling to create the tree of height $j+1$, a "U-turn" is detected within any of the subtrees of the new tree of height j, all of its states are discarded and the tree before the doubling is taken as the final tree. That is, if eq. (5.2) is met for any of the subtrees of the new tree, we must discard their states. The reason for this is that if we were to begin the doubling from any of these states, the No-U-Turn condition is met before we can rebuild $\mathcal{B}$ and thus we violate reversibility and inadvertently detailed balance.

2. Consider now the combined tree after doubling. Naturally, none of the subtrees will satisfy the No-U-Turn condition because

   (a) The tree before doubling had not triggered the termination of the doubling procedure. Hence, none of its subtrees nor the full tree satisfy eq. (5.2).

   (b) The new tree, which is the other half of the combined tree, did not trigger a termination either so none of its subtrees nor the full tree satisfy eq. (5.2).

   The only part of the combined tree that can satisfy the No-U-Turn condition at this is point is the rightmost and leftmost nodes of the entire tree. If eq. (5.2) is met in this case, we terminate the doubling but no state must necessarily be discarded. After all, since the full tree is built and none of subtrees satisfy the No-U-Turn condition, we can start from any state and find a unique set of directions $\{v_j\}$ from which we can rebuild the entire tree before the No-U-Turn condition is satisfied.

There is another case in which we want to stop the doubling procedure. If at any point, the error of the simulation becomes too large, the states produced during the doubling process is likely to lie in a low probability region of parameter space. Continuing the simulation at this point will likely waste computational resources as states integrated from these low probability states will yield new states in a low probability region. Let (q', p') be any state in the tree (including the initial state) and denote the initial state as (q, p). The doubling is terminated if

$$H(q', p') - H(q, p) + \log\Lambda \geq \Delta_{\max}, \tag{5.3}$$

where $\Lambda \sim \text{Uniform}(0, 1)$ is sampled in the beginning of the tree building (and is the slice variable used during Metropolis correction to accept or reject a state) and $\Delta_{\max}$ is a tolerance which the authors of the original paper recommends to be set to $\Delta_{\max} = 1000$ to allow the tree building to continue if the error introduced by the Leapfrog integrator is moderate. Equation (5.3) essentially states that if the energy difference becomes too large, we terminate the tree building. The tree produced during this final doubling must be discarded and the final tree becomes the tree prior to doubling. The reasoning is the same as before; we cannot initiate the Leapfrog integrator from the states in this tree and rebuild $\mathcal{B}$ as the stopping condition in eq. (5.3) is met before the full tree can be rebuilt.

Once the tree $\mathcal{B}$ is built, the NUTS sampler selects a *candidate set* $\mathcal{C}$ from the tree where all of its elements, which we define as *candidate states*, must satisfy

$$\frac{\pi(q', p')}{\pi(q, p)} = \exp\{-[H(q', p') - H(q, p)]\} > \Lambda, \tag{5.4}$$

which is the same Metropolis correction that is employed in HMC [18]. The next state in the Markov chain is drawn randomly from $\mathcal{C}$. The selected state (q', p') is projected onto q' which is the parameter of interest that is next in line in the Markov chain. We have defined a function `NUTSstep` in algorithm 5.1 which generates the next state q' in the Markov chain given a prior state q, a Hamiltonian H and a step size ϵ.

Algorithm 5.1 The NUTS Sampler

```
  function NUTSstep($q, H, \epsilon$)
      Sample $p \sim \mathcal{N}(0, I)$
      Sample $\Lambda \sim$ Uniform$(0, 1)$
      Set the initial tree $\mathcal{B} \leftarrow \{(q, p)\}$
      for $j \geq 1$ do
          Sample $v_j \sim$ Uniform$(\{-1, 1\})$
          if $v_j = 0$ then
              Perform $2^j$ Leapfrog steps from the rightmost node of the current tree. Assign to $\mathcal{B}'$
          else
              Perform $2^j$ Leapfrog steps from the leftmost node of the current tree. Assign to $\mathcal{B}'$
          end if
          if For any subtree in $\mathcal{B}'$, eq. (5.2) is satisfied then
              Terminate building of tree and discard $\mathcal{B}'$
          else
              $\mathcal{B} \leftarrow \mathcal{B} \cup \mathcal{B}'$
      end if
  end for
end function
```

5.1.2 Computational Cost

The No-U-Turn sampler introduces additional operations to keep track of whether any of the stopping conditions are met. Equation (5.2) requires $2^{j+1} - 2$ evaluations of inner products for a tree of height j, two inner products per subtree. In addition, eq. (5.3) requires $2^j - 1$ evaluations of the Hamiltonian, and its gradient must be calculated an equal amount of times to perform Leapfrog integration similar to what is required by HMC. The additional cost of the inner products are, however, neglible for sufficiently complex models and/or large datasets as the evaluation of the Hamiltonian and its gradient will be the dominating computational cost. Another added computational cost is the memory footprint introduced by storing the balanced binary tree. In its naive form, the memory footprint requires the order $\mathcal{O}(2^j)$ states. A more efficient solution can be found by observing that the uniform distribution over the candidate set $\mathcal{C}$ can be rewritten as

$$p(q, p|\mathcal{B}, \mathcal{C}) = \frac{1}{|\mathcal{C}|} = \frac{|\mathcal{C}_{\text{subtree}}|}{|\mathcal{C}|} \frac{1}{|\mathcal{C}_{\text{subtree}}|}, \tag{5.5}$$

where $|\cdot|$ denotes the *cardinality* or the number of elements in the set and $\mathcal{C}_{\text{subtree}} \subseteq \mathcal{C}$ is the candidate states in a subtree of the subset of the full tree corresponding to the candidate set. Equation (5.5) states that the uniform probability over $\mathcal{C}$ can be rewritten as the probability of selecting a subtree $\mathcal{C}_{\text{subtree}}$ from $\mathcal{C}$ times the probability of drawing a state at random from that subtree. A tree of height j consists of two subtrees of height $j-1$. From each subtree for $j > 0$, draw a state (q, p) from each subtree with probability $1/|\mathcal{C}_{\text{subtree}}|$ to represent that tree and a give it a weight proportional to how many states of the total candidate set that belonged to that particular subtree. Starting from the initial tree of height $j > 0$, this can be performed during the doubling process for each new subtree that is generated to avoid explicit storage. The storage requirement is thus brought down to an order of $\mathcal{O}(j)$ position-momentum states, which significantly reduces the memory footprint.

Chapter 6

Bayesian Neural Networks

6.1 Neural Networks

In this chapter, we will finally discuss the main topic of this thesis, *Bayesian neural networks* (BNNs). We will start off introducing the mathematical formalism of neural networks. We will then discuss the *backpropagation* algorithm, which is the standard algorithm used to compute the gradient of the model with respect to a specified loss. We will then end the chapter with how Bayesian learning of neural networks work. Fortunately, most of the groundwork is already laid, so we need only a mathematical description of the model and a Bayesian interpretation of it. We will stay general and assume a set of inputs $x \in \mathbb{R}^p$ and corresponding targets $y \in \mathbb{R}^d$. These serve as the training data on which the neural network is trained. We will adopt the terminology used by the TensorFlow framework [19] to help make the transition from mathematics to code easier.

6.1.1 Basic Mathematical Structure

A neural network is most generally defined as a non-linear function $f : \mathbb{R}^p \to \mathbb{R}^d$ built up as follows.

- A set of L layers. Consider the ℓ'th layer. It consists of n_ℓ nodes all of which has a one-to-one correspondence to a real number. The conventional representation is with a real-valued vector $a^\ell \in \mathbb{R}^{n_\ell}$ called the *activation* of layer ℓ.
- For convenience, the layer with $\ell = 1$ is often called the *input layer* and the layer with $\ell = L$ is referred to as the *output layer*. The layers in between for $\ell = 2, ..., L-1$ are called the *hidden layers*. Although this distinction is merely conceptual and does not change the mathematics one bit, it provides useful categories for discussion later on.
- Each layer ℓ is supplied with a (possibly) non-linear function $\sigma_\ell : \mathbb{R}^{n_{\ell-1}} \to \mathbb{R}^{n_\ell}$. In other words, it defines a mapping $a^{\ell-1} \mapsto a^\ell$. The complete neural network function can thus be expressed as
$$f(x) = (\sigma_L \circ \sigma_{L-1} \circ \cdots \circ \sigma_\ell \circ \cdots \circ \sigma_2 \circ \sigma_1)(x). \tag{6.1}$$
- To each layer, we assign a *kernel* $W^\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ and a *bias* $b^\ell \in \mathbb{R}^{n_\ell}$. Together, these parameters are called the *weights* of layer ℓ.
- The complete set of neural network parameters $(W, b) \equiv \{(W^\ell, b^\ell)\}_{\ell=1}^{L}$ are called the weights of the network. They serve as the *learnable* or *trainable* parameters of the model.
- Finally, we introduce the *logits* $z^\ell \in \mathbb{R}^{n_\ell}$ of layer ℓ.
- The permutation of number of layers, number of nodes per layer and activation functions are collectively called the *architecture* of the neural network.

The activation in layer ℓ is computed through the recursive equation:

$$a_j^\ell = \sigma_\ell \left(\sum_k W_{jk}^\ell a_k^{\ell-1} + b_j^\ell \right) \equiv \sigma_\ell(z_j^\ell), \quad \text{for} \quad j = 1, 2, ..., n_\ell. \tag{6.2}$$

A special case of eq. (6.2) applies to $\ell = 1$ where $a^0 = x \in \mathbb{R}^p$ is assumed.

6.1.2 Backpropagation

The standard approach to train a neural network is by minimization of some loss function by employing the backpropagation algorithm [20] to compute its gradient with respect to its trainable parameters recursively. The algorithm boils down to four equations. Consider $\mathcal{L}$ as the loss function. The first of the four equations quantifes the change in the error with respect to the logits z_j^L in the output layer,

$$\Delta_j^L = \frac{\partial \mathcal{L}}{\partial z_j^L}, \tag{6.3}$$

but for convenience we will simply regard this as the “error” in the output layer (and use the same term for Δ_j^ℓ). For example, in the case where $\mathcal{L} = \text{RSS}$, we get

$$\Delta_j^L = \frac{\partial \mathcal{L}}{\partial z_j^L} = a_j^L - y_j, \tag{6.4}$$

for a single datapoint y, so the use of the term is largely appropriate. Fundamentally, they denote the gradient of the error with respect to the quantities defined with respect to the neural network model. The second equation allows us to compute the error at layer ℓ given we know the error at layer $\ell + 1$,

$$\Delta_j^\ell = \left(\sum_k \Delta_k^{\ell+1} W_{kj}^{\ell+1} \right) \sigma_\ell'(z_j^\ell). \tag{6.5}$$

The final two equations relate these errors to the gradient of the loss function with respect to the model parameters. For the kernels, we have

$$\frac{\partial \mathcal{L}}{\partial W_{jk}^\ell} = \frac{\partial \mathcal{L}}{\partial z_j^\ell} \frac{\partial z_j^\ell}{\partial W_{jk}^\ell} = \Delta_j^\ell a_k^{\ell-1}. \tag{6.6}$$

For the biases, the gradients are

$$\frac{\partial \mathcal{L}}{\partial b_j^\ell} = \frac{\partial \mathcal{L}}{\partial z_j^\ell} \frac{\partial z_j^\ell}{\partial b_j^\ell} = \Delta_j^\ell. \tag{6.7}$$

With these four equations, we can fit the neural network using minimization techniques such as stochastic gradient descent or more complex methods such as ADAM (pages 13-19 in [8]). Although not the focus of this thesis, we might use these methods in conjunction with HMC to speed up convergence to the stationary distribution. Furthermore, the computation of gradients in combination with HMC or NUTS is achieved with the backpropagation algorithm as we know from chapter 4 where $\mathcal{L}$ coincides with the potential energy function whose gradient is necessary to employ these samplers.

We are now equipped to write down the backpropagation for a single datapoint. It’s built up of a *forward pass* which takes an input x and applies the recursive eq. (6.2) which produces a model prediction $\hat{y} = a^L$. The second part of the algorithm is the *backward pass* which based on the prediction $\hat{y}$ and the target y, computes the gradient of the loss function $\mathcal{L}$ with respect to the model parameters. The forward pass of the neural network is summarized algorithm 6.1.

Algorithm 6.1 Backpropagation: Forward pass

```
procedure ForwardPass($x$)
    $a_j^0 = x_j$    for    $j = 1, \dots, p$                ▷ Initialize input
    for $\ell = 1, 2, .., L$ do
        for $j = 1, 2, .., n_\ell$ do
            $a_j^\ell \leftarrow \sigma_\ell \left( \sum_k W_{jk}^\ell a_k^{\ell-1} + b_j^\ell \right)$
        end for
    end for
end procedure
```

The backward pass of the algorithm is stated in algorithm 6.2.

Algorithm 6.2 Backpropagation: Backward pass

```
procedure BackwardPass($\mathcal{L}, x, y$)
    for $j = 1, 2, \dots, n_L$ do
        $\Delta_j^L \leftarrow \partial \mathcal{L} / \partial z_j^L$
        $\partial \mathcal{L} / \partial b_j^L \leftarrow \Delta_j^L$
        $\partial \mathcal{L} \Big/ \partial W_{jk}^L \leftarrow \Delta_j^L a_k^{L-1}$
    end for
    for $\ell = L - 1, \dots, 1$ do
        for $j = 1, \dots, n_\ell$ do
            $\Delta_j^\ell \leftarrow \left( \sum_k \Delta_k^{\ell+1} W_{kj}^{\ell+1} \right) \sigma'(z_j^\ell)$
            $\partial \mathcal{L} / \partial b_j^\ell \leftarrow \Delta_j^\ell$
            $\partial \mathcal{L} \Big/ \partial W_{jk}^\ell \leftarrow \Delta_j^\ell a_k^{\ell-1}$
        end for
    end for
end procedure
```

Note that in all practical implementations in this thesis, we utilize *automatic differentiation* provided by TensorFlow to compute the gradients.

6.1.3 Regularization in Neural Networks

As discussed in chapter 2, models with a large number of parameters are prone to overfit training data and generalize poorly as a consequence. Thus one typically tacks on an L^2-regularization term to the loss $\mathcal{L}_0$. Assuming that $\mathcal{L}_0$ is the RSS in eq. (2.2), the form of the full loss function for a neural network model becomes

$$\mathcal{L} = \frac{1}{2} \sum_i \left\| \hat{y}^{(i)} - y^{(i)} \right\|_2^2 + \frac{\lambda_W}{2} \sum_\ell \left\| W^\ell \right\|_2^2 + \frac{\lambda_b}{2} \sum_\ell \left\| b^\ell \right\|_2^2, \tag{6.8}$$

where λ_W and λ_b are regularization strengths for the kernels and biases respectively. The L^2-norm $\|\cdot\|_2$ is the standard Euclidean norm in the case of a vector. For a matrix, we mean the following. Let

$A \in \mathbb{R}^{m\times n}$. The matrix norm $\|\cdot\|_2$ is then given by *Fröbenius norm*

$$\|A\|_2 = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|A_{ij}|^2}. \tag{6.9}$$

6.2 Activation Functions

There are many common activation functions σ with various strengths and weaknesses used in modern neural networks. We shall briefly mention a few for completeness.

6.2.1 Sigmoid and Tanh

The sigmoid activation function is given by

$$\sigma(x) = \frac{1}{1+\exp(-x)}. \tag{6.10}$$

It was a very common choice in neural networks early on, likely due to its simple derivative. It has a significant drawback, however. Looking at eq. (6.10), we can easily deduce that $\sigma(\infty) = 1$ and $\sigma(-\infty) = 0$, and since its derivative is of the form $\sigma'(x) = \sigma(x)(1-\sigma(x))$, the gradient computed with backpropagation vanishes if $|x| \to \infty$. This significantly hampers the progress during optimization.

A popular alternative to the sigmoid function is the hyperbolic tangent given by

$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}. \tag{6.11}$$

This function is very similar to sigmoid in the sense that its derivative vanishes for inputs of large magnitude and so may suffer from the same issues as sigmoid does.

6.2.2 ReLU

To overcome the vanishing gradient problem, an activation function called the Rectifying Linear Unit (ReLU) became widely adopted, which is given by

$$\sigma(x) = x^+ = \max(0, x). \tag{6.12}$$

6.2.3 Swish

Recently, an activation function to replace ReLU was proposed in [21] known as *swish* or SiLU which was shown to outperform ReLU in deep neural networks on a number of challengig datasets. The activation function is given by

$$\sigma(x) = x \cdot \text{sigmoid}(x) = \frac{x}{1+\exp(-x)}. \tag{6.13}$$

6.3 Bayesian learning of Neural Networks using Monte Carlo Samplers

So far, we have discussed neural networks as a model class whilst ignoring the issue of what it really means to do Bayesian learning of neural networks, in other words, what it means to *train* BNNs. We have intentionally left it somewhat ambigious what this really means because as it turns out, its meaning can be quite different depending on how Bayesian inference is performed. In this section we will clarify precisely what it means to train BNN using MCMC samplers such as HMC and NUTS. We shall then discuss practical aspects of the training which we shall put to practice in chapter 7.

6.3.1 What *is* Bayesian learning of Neural Networks?

The way Bayesian learning of neural networks manifest itself depends on the way in which we do Bayesian inference of the probabilistic model. We are concerned with inference of model parameters from the posterior using MCMC methods and will therefore obtain samples where each such sample consist of the weights of an entire neural network. More precisely, if we gather N samples with a chosen sampler, we will obtain N entire neural networks all sampled from the posterior to explain the observed data. Thus, what we mean by a *trained* BNN in this sense is that we have sampled a set of neural networks that collectively represent the BNN.

As we discussed at the end of chapter 2, we are mainly interested in the predictive distribution $p(y|x, W, b)$ of an output y given an input x. We can approximate this distribution by constructing an empirical distribution by feeding x through all N sampled neural networks to obtain N predicted targets $\hat{y}$ using eq. (2.26). The second quantity of interest is expectations of target functions dependent on the model parameters. We can approximate any such expectation with an MCMC estimator as in eq. (3.4) using all N networks to evaluate the target function.

6.3.2 The Potential Energy Function of Neural Networks

We now turn to the Bayesian formulation of the neural network model for use with the samplers used in this thesis. Assume that we have picked an architecture for a neural network and wish to train it in the Bayesian sense. For both HMC and NUTS, we need only specify a potential energy function for our model. The samplers take care of the rest. Assume we are dealing with a dataset $D = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$ where all N points are independent and identically distributed. Equation (4.15) instructs us to specify a prior for the weights of the network, and a likelihood function that depends on the target and the model output, in order to fully specify the potential energy function. Common practice is to choose priors that are either Gaussian or Laplacian. We will operate with Gaussian priors, i.e.

$$P(W^\ell) \propto \exp\left(-\frac{\lambda_W}{2}\left\|W^\ell\right\|_2^2\right) \quad \text{and} \qquad P(b^\ell) \propto \exp\left(-\frac{\lambda_b}{2}\left\|b^\ell\right\|_2^2\right). \tag{6.14}$$

We will not worry too much about the choice of priors as the term in the potential energy function that corresponds to the likelihood will be much larger in practice. The Gaussian priors serve roughly the same purpose as L^2-regularization does in classical ML.

The likelihood for regression from eq. (2.17) formulated in terms of a neural network $\hat{f}(x^{(i)}; W, b)$ is

$$p(D|W, b) = \exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{N}\left\|y^{(i)} - \hat{f}(x^{(i)}; W, b)\right\|_2^2\right), \tag{6.15}$$

where σ is treated as a hyperparameter. This is not the only valid choice for a likelihood function but it is the common choice since it can be identified with the Euclidean L^2-norm and its “neat” mathematical properties.

Combining the priors and the likelihood with eq. (2.24) yields the potential energy function

$$\mathcal{L} = \frac{1}{2\sigma^2}\sum_{i=1}^{N}\left\|y^{(i)} - f(x^{(i)}; W, b)\right\|_2^2 + \frac{\lambda_W}{2}\sum_{\ell=1}^{L}\left\|W^\ell\right\|_2^2 + \frac{\lambda_b}{2}\sum_{\ell=1}^{L}\left\|b^\ell\right\|_2^2, \tag{6.16}$$

up to a constant. As we discussed in chapter 4, the potential energy function also happens to be the typical loss function with L^2-regularization used in the classical ML which is why we denote it as $\mathcal{L}$. At this point, we have set up all the machinery we need to train BNNs. Our next topic of discourse is the practice of doing so.

6.3.3 Practical Training of Bayesian Neural Networks

Training BNNs in practice requires us to specify a fairly large number of hyperparameters to obtain a set of models. These are

1. **Neural network architecture**. We need to specify its number of layers, number of nodes and activation function per layer. Once the BNN is trained, we store this information along with the model for future usage. The stored weights themselves will encode how many layers and nodes the model has but the activation functions must be stored in addition.

2. **Number of results**. We must specify how many neural networks we want to sample and store. Because the weights must be stored in its entirety, we are forced to worry about the amount of disk space that is required to do so. For a fixed allocated disk space, we can obviously store a larger set of samples if the model is simple. As complexity increases, the number of samples we can store will necessarily decrease.

3. **Number of warm-up steps**. We must decide how long we want to run the MCMC chain before we start storing results. If amount of disk space was no obstacle, this step would be considered entirely optional as we could simply store every single sample and make a thorough analysis of the chain's quality to determine when proper mixing is obtained. In practice, with TensorFlow's framework, we can make a predetermined set of burn-in steps to avoid unnecessary RAM usage. In conjunction with a predetermined number of burn-in steps, we must also set a number of adaptation steps to dynamically set the step size used with the Leapfrog integrator. We shall the call total number of burn-in steps and adaptation steps as the number of *warm-up steps*.

4. **Amount of thinning**. Since successive samples most likely will be correlated, we can specify how many samples we simply skip once we start gathering samples, i.e. after the burn-in period. Again, we could ignore this and do this manually with the chain but doing so becomes a question of amount of available VRAM, RAM and disk space.

5. **Hyperparameters specific to the samplers**. The samplers themselves carry their own hyperparameters. In the case of HMC, we must specify a fixed number of Leapfrog steps L. If we use the NUTS sampler, we must specify the maximum tree depth. Moreover, we must determine how much of the computing resources we allocate to adapting the step size used in the Leapfrog integrator.

6. **Amount of pretraining**. An attempt to accelerate convergence of the MCMC chain can be achieved by pretraining the neural network using minimization methods with the backpropagation algorithm to bring the weights closer to a minima of the potential energy function (i.e. the loss function used in classical ML). Then the point estimate obtained at the end of the training is used as a starting point for the MCMC chain.

6.3.4 Training Algorithm of Bayesian Neural Networks

In this section we shall turn our attention to an actual training algorithm for BNNs. Assume we pick a sampler S that represents either HMC or NUTS and a specified permutation of the hyperparameters discussed in the last section. In practice we can summarize a training algorithm as follows.

1. Initialize the weights of the model from the specified priors, i.e.

$$\text{Sample} \quad W^\ell \sim p(W^\ell) \quad \text{and} \quad \text{sample} \quad b^\ell \sim p(b^\ell) \quad \text{for} \quad \ell = 1, \dots, L. \tag{6.17}$$

2. Minimize the potential energy function $\mathcal{L}$ with respect to the weights of the model using an optimizer of your choice to obtain a point estimate for use as the initial state of the Markov chain.

3. Initialize the Markov chain for a finite set of burn-in steps to achieve mixing using S. A proportion of the initial burn-in steps are used for step size adaptation, while the remaining are used for mixing.

4. Gather samples by applying S repeatedly, replacing the current weights of the model by the ones returned by S.

Chapter 7

Numerical Experiments

Six technical chapters have culminated to the main event, the numerical experiments and their results. This is where we learn if BNNs can deliver on the promise of substituting direct calculations of cross sections. We will begin with a description of the dataset and the data transformations made prior to training and their potential implications for the predictive performance of the trained BNNs. We will then explain the methodology employed including the implementation details, the selection of BNN models and hyperparameters and performance metrics. Once discussed, we will explore the results and their consequences.

7.1 The Dataset

In this section, we will give a brief description of the dataset. Moreover, we will discuss the data transformations prior to training and its implications on the accuracy of the predictions.

For the predictions presented in this chapter, we have restricted our investigations to use the dataset for a single particle process throughout. This will make it easier to compare the performance of BNNs across different configurations to better understand the strengths and weaknesses of the different choices made when training BNNs.

7.1.1 The Features and Targets

We will focus on a particular neutralino-neutralino. Neuralinos are denoted by the symbol $\tilde{\chi}_i^0$ for $i = 1, 2, 3, 4$. Each neutralino carries its own *mass* $m_{\tilde{\chi}_i}$ and a set of *mixing angles* N_{ij}. For each neutralino i, there are four mixing angles for $j = 1, 2, 3, 4$. The two possible cases of interest then would be a process with two identical neutralinos, in which case the input features are of the form

$$x = (m_{\tilde{\chi}_i^0}, N_{i1}, N_{i2}, N_{i3}, N_{i4}), \tag{7.1}$$

or a process where there are two distinct neutralinos i and k, such that the input features would have the form

$$x = (m^0_{\tilde{\chi}_i}, N_{i1}, N_{i2}, N_{i3}, N_{i4}, m_{\tilde{\chi}_k^0}, N_{k1}, N_{k2}, N_{k3}, N_{k4}). \tag{7.2}$$

The targets of the dataset are NLO cross sections of the form

$$\sigma_{\tilde{\chi}_i^0 \tilde{\chi}_k^0} = \text{LO} + \text{NLO}. \tag{7.3}$$

In our case we will focus on processes that results in $\tilde{\chi}_1^0 \tilde{\chi}_1^0$, meaning the input features have the form in eq. (7.1). The masses $m_{\tilde{\chi}_i^0}$ can take on both postive and negative values in the dataset. In figure 7.1, we show the NLO cross sections $\sigma_{\tilde{\chi}_1^0 \tilde{\chi}_1^0}$ projected onto the axis of masses $m_{\tilde{\chi}_1^0}$ and in figure 7.2 we show their values projected onto the axes of the mixing angles N_{1j} for $j = 1, 2, 3, 4$. Note in particular that the cross sections span several orders of magnitude which necessitates a data transformation to reliably perform regression analysis with the dataset. We can also note a few outliers most of which yield neglible cross section values, but we should be aware of that adverse effect of these can affect the trained BNN models. There is a certain asymmetry in some of the features as well, where there are largely more cross sections on the right side of

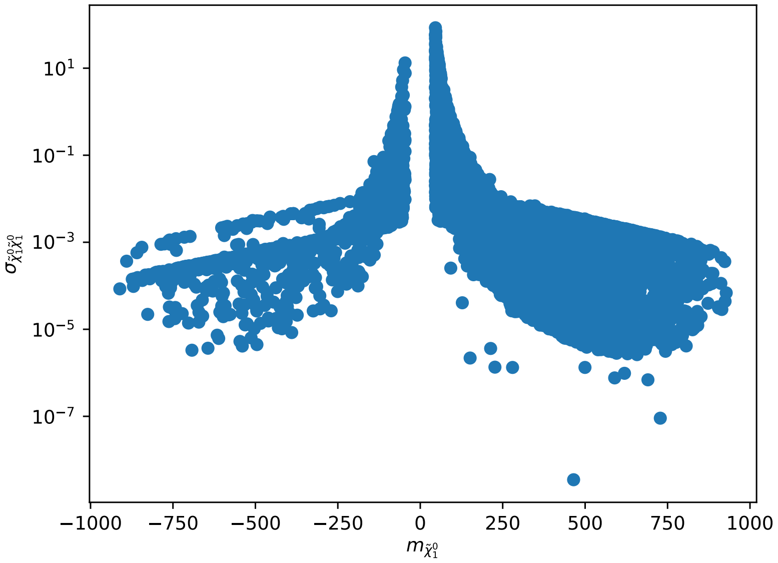


Figure 7.1: The values of the cross sections $\sigma_{\tilde{\chi}_1^0\tilde{\chi}_1^0}$ are shown projected onto the axis of masses $m_{\tilde{\chi}_1^0}$. The data is taken from the training data.

7.1.2 Data Transformations

We shall briefly discuss how the training data is transformed before training. The targets in the dataset of NLO cross sections can span several orders of magnitude. For practical training of BNNs, this would require model parameters that also span several orders of magnitude. The result will usually be overflow and thus unsucessful training of the models. Therefore, we have chosen to map the targets using the base-10 logarithm, i.e. $y \mapsto \log_{10}(y)$. More generally, we could choose any base-a logarithm. A practical consideration here is that once the model is trained, any prediction it produces must be transformed back using the inverse mapping to produce a cross section. As we increase the value of a, the precision the model's prediction decreases. Thus a small error in log-space can result in a large error in what we may refer to as the target space, the larger the value of a is. We will explore the performance both in log space using the transformed data and in target space by applying the inverse mapping to the predictions.

7.1.3 Data Splitting

Data splitting is a common strategy in machine learning to avoid biased model selection and obtain reliable estimates of the performance of the trained models. The conventional way is to split the dataset $\mathcal{D}$ into three subsets:

1. A training set $\mathcal{D}_{\text{train}}$. This dataset usually contain the largest chunk of the dataset and is used to train the models.

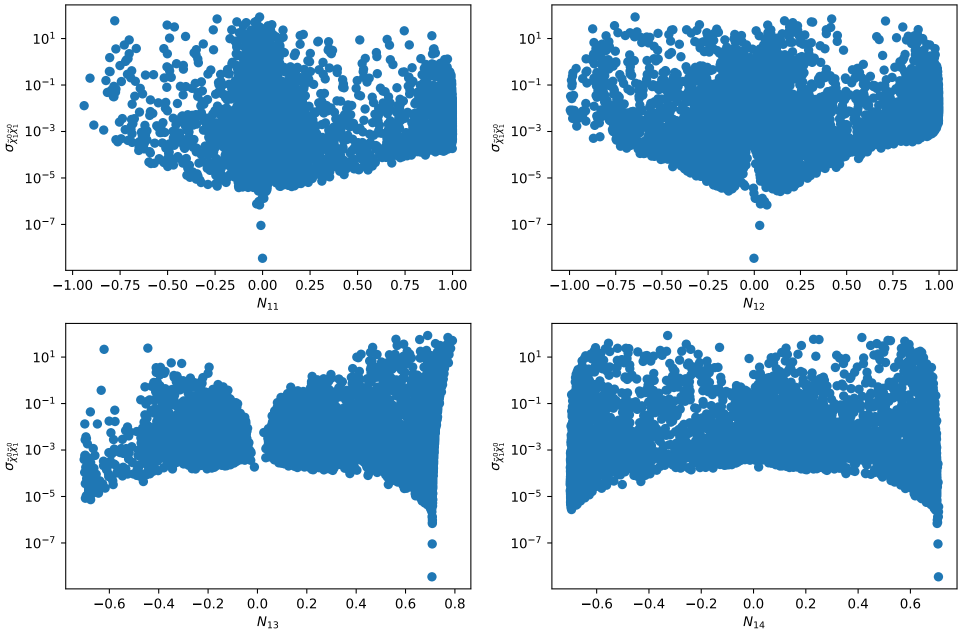


Figure 7.2: The values of the cross sections $\sigma_{\tilde{\chi}_1^0\tilde{\chi}_1^0}$ are shown projected onto the axes of mixing angles N_{1j} for $j = 1, 2, 3, 4$. The data is taken from the training data.

2. A validation set $\mathcal{D}_{\text{val}}$. This dataset is typically the smallest of the bunch and is sometimes used in classical machine learning problems to perform cross-validation or similar methods. The results measured here are typically used to select hyperparameters of the model.

3. A test set $\mathcal{D}_{\text{test}}$. This partition is slightly larger than the validation set and is used as an out-of-sample check to measure the performance of a model.

We have selected to use a division of 80% training data, 5% validation data and 15% test data. In practice though, the notion of a validation set does not lend itself as easily to Bayesian ML tasks and we may in practice therefore use both the validation and test data as the test set.

7.2 Methodology

In this section, we shall explain the methodology used to train and test the BNNs explored in this thesis. We will explain the implementations, selection of models and hyperparameters and performance metrics used.

7.2.1 Implementation

We have utilized the Python libraries `TensorFlow 2.7.0` and `TensorFlow-Probability 0.15.0` to implement the BNNs. The implementation itself is available at `LEGGTILURLHER`. Unfortunately, BNNs trained with HMC or NUTS has not been of interest for majority of the deep learning community. The main focus has been with use of surrogate distributions to develop algorithms that when employed using GPUs spend roughly the same amount of time per epoch as training of classical neural networks.

As a result no implementations of BNNs for these kind of samplers have been implemented directly into the either framework. Luckily, `TensorFlow-Probability` provide general purpose implementations of the samplers we have discussed hithertho which only requires us to define the *unnormalized negative log-target density* (which is the negative potential function -$\mathcal{L}$) to utilize the them. Consequentially, we created our own class that facilitates the usual kind of conveniences shipped with `TensorFlow` such as the ability to automatically save, load or print the model architecture to screen, to name as few. The class and its functionality is well-documented and made with the intention to be reused, expanded and modified.

Both `TensorFlow` and `TensorFlow-Probability` handle execution on NVIDIA GPUs automatically with minimal effort on the user side, which we have utilized to generate our results. We will also provide measurements that indicate the expected speedup gained from using a GPU instead of a CPU for training of BNNs. Some of the results are also generated using the built-in GPU on an M1 Apple Silicon system-on-chip (SoC), since a port that supports execution on this device is available. In any case where computational performance is measured, we will make it crystal clear on what hardware the calculations are performed.

7.2.2 Selection of Models and Hyperparameters

In order to better understand the behaviour of BNNs, we have chosen to train a set of models whose details are listed in table 7.1. Each model consists of 1000 sampled neural networks. Each model is trained with $\tanh(x)$ as the activation function on the hidden layers, while the output layer uses an identity activation. We will refer this table whenever a model or a set of models selected from it is used. Otherwise, we will state the model architecture used and its hyperparameters explicitly.

Table 7.1: The table shows the models used in this section. For each model, 1000 sampled networks were sampled to collectively represent each BNN model. We used 2500 warm-up steps (20% burn-in and 80% adaptation). We skipped 10 samples for each sampled network. We performed 1000 pretraining epochs with a batch size of 32 using the ADAM optimizer. The kernel used for each model was the NUTS kernel with a maximum of $L = 4096$ Leapfrog steps. The number of nodes per layer is shown in the “Layers” column. For each hidden layer, we used $\tanh(x)$ as the activation function. The final layer used an identity activation.

Model number	Layers	Number of parameters
1	5-50-1	351
2	5-50-50-1	2901
3	5-50-50-50-1	5451
4	5-50-50-50-50-1	8001
5	5-50-50-50-50-50-1	10551

7.2.3 Performance Metrics

In this section, we will discuss the performance metrics used to benchmark and measure the performance of the models trained in this thesis. Due to the inherent probabilistic nature of the models trained, any output the model produces will be a distribution from which we can calculate a sample mean and variance. We will introduce a metric to measure the performance of the mean predictions of the BNNs called coefficient of determination and a metric to assess the BNNs ability to yield reliable uncertainties in its predictions called standardized residuals.

7.2.3.1 Coefficent of Determination

The *coefficent of determination*, or the R^2-*score*, is used to assess the quality of the predictions of a model in supervised regression tasks. For a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, it is given by

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=1}^{N}(y^{(i)} - \bar{y})^2}, \tag{7.4}$$

where $y^{(i)}$ denotes the target of an input $x^{(i)}$ and $\hat{y}^{(i)}$ denotes the model prediction and $\bar{y}$ denotes the sample mean of the targets

$$\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y^{(i)}. \tag{7.5}$$

The R^2-score lies in the range $R^2 \in (-\infty, 1]$ where the larger the value, the better the model predicts. The score is interpreted as the proportion of the variance in the targets that can be predicted from the inputs. The reason we select this metric is that it provides a more reliable interpretation of the prediction quality of our models than other metrics such as mean squared error (MSE), root mean squared error (RMSE) and the mean absolute error (MAE) which can only be used to compare the predictions of models relative to each other [22]. Their values all lie in the range of $[0, \infty)$ where a perfect prediction would yield zero. The lack of an upper-bound on these alternative metrics make them difficult to interpret in a vacuum, a weakness the R^2-score does not suffer from. If $R^2 < 0$, the model performs poorly, while of $R^2 \in [0, 1]$, the model explains the variation in the data with $R^2 = 0$ meaning the model cannot explain any of the variance in the targets around their sample mean. A score of $R^2 = 1$ means a perfect prediction.

Note that when we use BNNs to compute the R^2-score, we will replace $\hat{y}^{(i)}$ in eq. (7.4) with the sample mean of the predictive distribution computed by the BNNs.

7.2.3.2 Standardized Residuals

Standardized residuals is a transformation of a model prediction given an input feature x^* and a target y^*. The mapping is defined as

$$z(x^*) = \frac{y^* - \hat{y}_{\text{mean}}(x^*)}{\hat{\sigma}(x^*)}, \tag{7.6}$$

where $\hat{\sigma}(x^*)$ is the square-root of the sample variance of the model predicitions and $\hat{y}_{\text{mean}}(x^*)$ is the sample mean of the predictions and. The mapping in eq (7.6) resembles the mapping of a random variable $x \sim \mathcal{N}(\mu, \sigma^2)$ onto the standard Normal distribution $z \sim \mathcal{N}(0, 1)$.

As we discussed in chapter 2, the fundamental assumption made is that any target y can be decomposed as

$$y = f(x) + \delta, \tag{7.7}$$

for some true function $f(x)$ and a random noise $\delta \sim \mathcal{N}(0, 1)$, i.e it is distributed according to a standard Normal distribution. But the noise in the data produced by `Prospino` is neglible, which means that $y \approx f(x)$. The regression error obtained through the sample variance of the model predictions must therefore be dominated by the variance of the predictive distribution computed by the model itself. Let σ_z^2 denote the variance of the distribution of the standardized residual. If $\sigma_z > 1$, the model will be considered overconfident in its predictions since the sample variance of the model's predictions are smaller than the variance of the targets around the mean prediction. On the other hand, if $\sigma_z \leq 1$, we consider the model to yield reliable uncertainty estimates. In this case, the model is not considered to be "underconfident" but rather "conservative".

7.3 Results and Discussion

In this section we present the results from various numerical experiments and discuss their implications. We start off with measurement of computational performance with a focus on training time, prediction time and loading times. We then investigate the posterior distribution projected onto a 2-dimensional planes of the BNN weights to investigate their property with a focus on whether typical approximations made in the literature provides a sufficient representation of the exact posterior distribution. Following this, we present the effect of various hyperparameters pertaining to the training of BNNs, including the performance achieved when using HMC and NUTS. Finally, we explore the predictive distributions of various trained BNNs.

7.3.1 Computational Performance

In this section we will explore the computational performance of BNN models, both on modern CPUs and GPUs. We shall measure the training time which with the way we perform Bayesian inference of the BNN parameters most appropriately translates to the time used per generated sample with HMC and NUTS. Another measurement of great interest is the time a trained BNN model used to compute predictions as this will be their central usage once trained. But because we must store the full BNN model on disk, the loading times play an important role as well. We shall therefore measure both the execution time of predictions and loading times separately. We will perform wall clock measurements using `time.perf_counter` provided by the `time` module of the standard library shipped with Python 3. It yields a measurement in fractional seconds of the clock with the highest available resolution on the system.

7.3.1.1 CPU v. GPU Performance

In this thesis we are primarily concerned with creation of an optimized alternative to direct calculations of cross sections. An important consideration is at which available commercially available hardware platform the most speedup can be achieved. In this section we pin the performance of available CPU and GPU hardware against each other. First note that `TensorFlow` and its framework supports OpenMP-type parallelization on the CPU, that is, parallelization where a single node consisting of several cores, or in the case case of a CPU that supports hyperthreading, a set of virtual threads exceeeding the number of physical cores that evenly divide the workload within the compute node, using a shared memory parallelization strategy similar to what a GPU does. Unfortunately, multi-device parallelization either on the CPU or the GPU lacks support, particularly in the case of non-synchronous execution. Thus in this section we will only compare performance of the single-node performance that is achievable with either a CPU or GPU. We will thus contrast the performance achieved by a single CPU node with several (virtual) cores available to the perfomence obtained by employing the most taxing workload on a GPU. In figure 7.3, we demonstrate the significant speedup that can be achieved with GPU accelerated sampling when using `TensorFlow-Probability` and its implementation of samplers. Here we have used HMC and a fixed $L = 512$ Leapfrog steps with XLA (Accelerated Linear Algebra) compilation enabled on the GPU. This is a highly optimized linear algebra execution engine that can significantly speed up code written with `TensorFlow` run on the GPU [23]. The time measurements per sample was in the order of magnitude of seconds for the most complex models tested. An important aspect for practical utilization of BNNs will be ease of training and accelerated sampling with NUTS or HMC, which is automatically achieved if the system can detect an NVIDIA GPU. Moreover, the full training time can be estimated by a few preliminary runs where execution time per sample is measured. This analysis is slightly more difficult when using NUTS. One can however set a maximum tree depth J which translates to a maximum of $L = 2^J$ Leapfrog steps when using the implementation in `Tensorflow-Probability`. As we argued in chapter 5, the additional computational cost added by NUTS per Leapfrog step is neglible given a sufficiently complex model and/or large dataset, so measurement of the time per sample used by HMC with a fixed L provides a sensible estimate of the maximal time used by NUTS per sample if the maximum tree depth is set to the corresponding number of Leapfrog steps used. Thus one can simply perform the measurements using the implementation of HMC and estimate an upper-bound on the computational time.

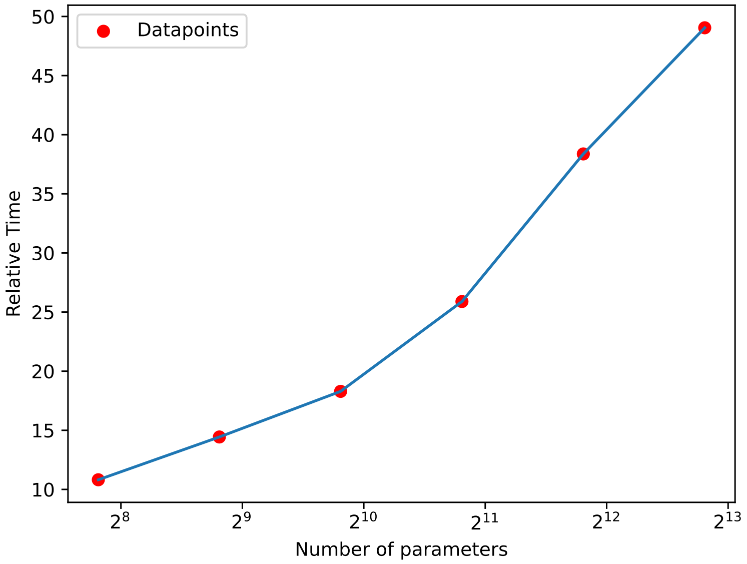


Figure 7.3: The figure shows the relative measured execution time used per sample using $L = 512$ Leapfrog steps, as a function of number of parameters. The CPU measurements are done using an 8-core M1 CPU (Apple Silicon). The GPU measurements are made with an NVIDIA Tesla P100 GPU.

7.3.1.2 Prediction Time

As we discussed in the introduction, the execution time's order of magnitude when using `Prospino` is in the order of hours. If BNNs are to serve as a viable alternative to these calculations, it must at least significantly reduce the time it takes to compute predictions. In figure 7.4, we show the average execution time to compute predictions using all models in table 7.1. For each model, we randomly generated input points of correct dimension and computed predictions for up to 4096 input points simultaneously. The execution times appear proportional to the number of input points provided for each model, which perhaps is not all that surprising. We can crudely infer by inspection that increasing the order of magnitude by one does the same for the execution time. Still, the order of magnitude for a single input point is at the order of a millisecond which is a significant speedup over `Prospino` calculations. Both the sample mean of the predictions and the sample error is computed during the measurement. The measurements were performed on an M1 Apple Silicon CPU using `perf_counter` from the module `time` provided by the standard library of Python.

The performance degradation that the computations in figure 7.4 suffers is inherently due to the limited vectorization capability of the CPU's computing units when performing matrix multiplications in the forward pass of the individual neural network models. The computation itself is performed with all 1000 sampled networks simultaneously, and so one might hypothesize that more specialized computing units may be able to handle several input points while applying all sampled networks at the same time. As it turns out, GPUs are excel at executing matrix multiplication and even more fortunate, `TensorFlow` has added support for the built-in GPU on Apple Silicon system-on-chips. In figure 7.5 we can see the execution times achieved using the GPU to perform the same computations as before. In this case the order magnitude remains more or less the same in all the tests. Thus, computing predictions on several points can benefit greatly if the execution is employed on a GPU. Note, however, that the measured execution time of "model 1" is slightly slower than for more points which likely is due to the overhead introduced by using the GPU for such a simple model. Care must thus be taken when considering what type of hardware the computations should be performed with.

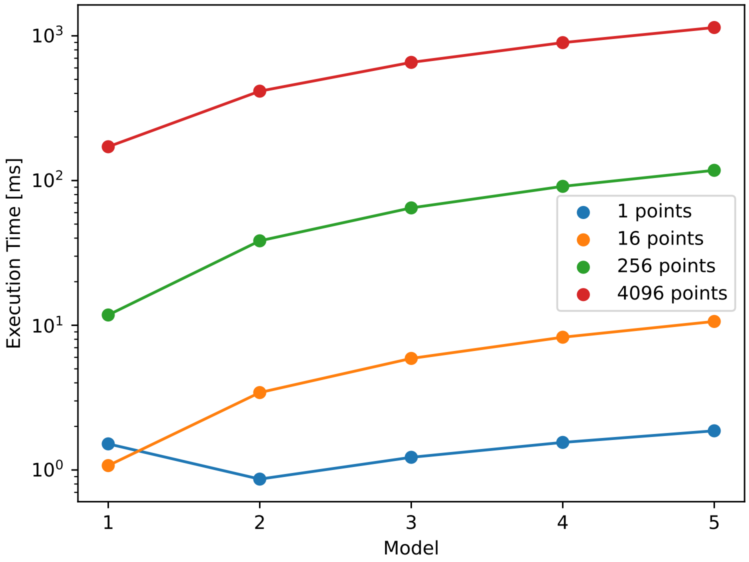


Figure 7.4: The figure shows the average prediction time to compute a prediction given a single input x using the models in table 7.1. The average time used is measured in ms and is averaged over 1000 randomly sampled points. The measured time includes computation of the sample mean and sample error.

7.3.1.3 Loading Times

Even if we have demonstrated a substantial speedup for predictions using BNNs, we have thus far ignored the fact that empirical distribution representing the weights of the BNN is stored on disk which typically means an solid state drive (SSD) with modern computing hardware. The memory bandwidth between the SSD and the faster forms of memory such as RAM, cache and registers becomes a potential bottleneck for performance. Although cache and registers introduce fast memory transfer of stored data to the computing units of the CPU, they typically boast a fairly limited capacity. Thus loading in the entire BNN model might not be viable and we may observe that once we need to models with a large number of parameters, the loading times dominate the computational cost involved with computing predictions. This added computional cost stems from the transfer of data back and forth between the RAM, and the cache and registers. An additional problem is that if the BNN model is simply used for a single prediction at a time, it might simply be loaded a single time before it is dumped from working memory all together. In this case, the initial load may dominate the computational cost all together.

In figure 7.6 we show the resulted loading times measured using an M1 Apple Silicon system-on-chip and `time.perf_counter` from Python. The memory allocated to the BNN models were deallocated manually using the `del` operator provided by Python to ensure that each load of the model of the model was from the SSD. The models loaded in are the ones listed in table 7.1. Given the order of magnitude of the loading times displayed in the figure is approximately the same order of magnitude

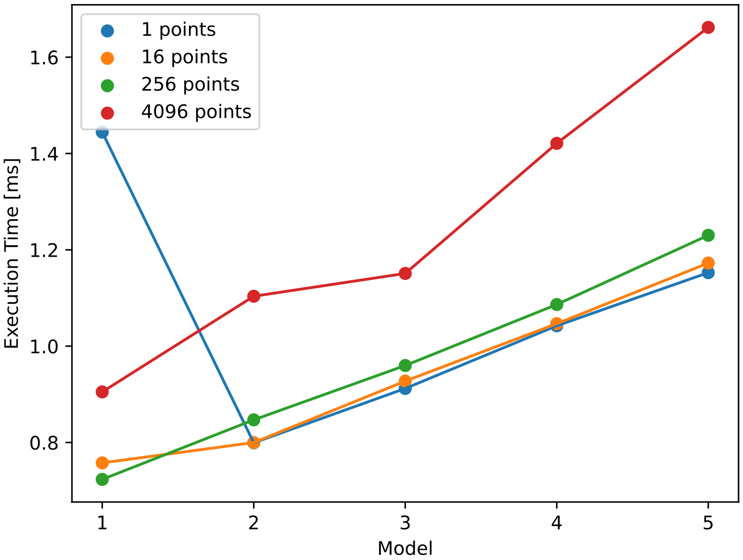


Figure 7.5: The figure shows the average prediction time using the built-in GPU on an M1 Apple Silicon system-on-chip to compute a prediction given a single input x using the models in table 7.1. The average time used is measured in ms and is averaged over 1000 randomly sampled points. The measured time includes computation of the sample mean and sample error.

as the execution time, we have demonstrated that BNNs can provide a serious substitute for `Prospino` calculations from a purely computational perspective. It remains to be seen if the predictions themselves are reliable enough for this substitution to be adopted, which we will explore in later sections.

7.3.2 Posterior Distribution of Weights

An important problem to consider is if we can even justify the use of Monte Carlo samplers to sample from the exact posterior instead of using the approximation employed by variational inference with a parameterized surrogate posterior which is the most ubiquitous method of training BNNs in the literature. The surrogate distribution is usually a factorized normal distribution of the form

$$q \propto \prod_{i,j,\ell} \mathcal{N}(\mu_{ij}^{\ell}, (\sigma_{ij}^{\ell})^2)\mathcal{N}(\mu_j^{\ell}, (\sigma_j^{\ell})^2), \tag{7.8}$$

meaning for each parameter in the model, we assume its posterior distribution can be written as an independent Gaussian distribution with a mean μ_{jk}^{ℓ} and a standard deviation σ_{ij}^{ℓ} for the kernels, and μ_j^{ℓ} and σ_j^{ℓ} for the biases. The method sports some fairly obvious advantages like the fact that one can perform *online training*, i.e. continue training once new data becomes available starting from an earlier *checkpoint* by using q obtained during earlier training as the prior. The way we have trained BNNs in this thesis does not permit this form of training because we cannot formulate a prior based on the empirical distribution we have sampled. Thus we cannot use the weights of the model that we have already sampled to continue training. We must start over entirely and discard the empirical distribution we obtained with the prior dataset.

It has been widely discussed that BNN posteriors are typically found to be multi-modal [24]. We demonstrate this observation in figure 7.7. We can observe that the projection onto the planes shown

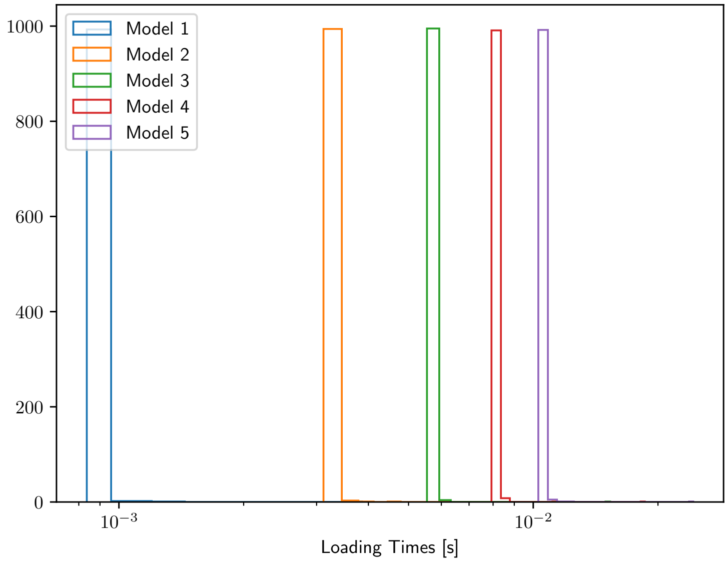


Figure 7.6: The figure shows the histograms of measured loading times in seconds using the models in table 7.1. The measurements were performed using `time.perf_counter` from Python using an M1 Apple Silicon system-on-chip. The time measurements consist of 1000 measurements for each model.

there indicate that the posterior distribution indeed is multimodal and unlikely to be approximated well with a parameterized surrogate distribution like the one in eq. (7.8).

7.3.3 Benchmarks of Hyperparameters

7.3.3.1 The Effect of Number of Warm-up Steps

As we discussed in section 6.3.3, we must set a predetermined number of warm-up steps, i.e. number of burn-in steps and number of adaptation steps when using `TensorFlow Probability`'s samplers. Conventional wisdom would have us believe that increasing the number of burn-in steps increases the probability that the Markov chain has converged to the stationary distribution of the posterior. Moreover, the literature has shown that NUTS performs at least as good as or better than HMC with an equivalent number of maximum Leapfrog steps or more as the results in [1] demonstrated. In our case we have split the number of warm-up steps to 20% burn-in steps and 80% adaptation steps, respectively. This is a heuristic recommended by the `TensorFlow Probability` developers.

To investigate the effect of the number of warm-up steps, we employed a model with a 5-20-20-1 architechture using $\tanh(x)$ as the activation function on the hidden layers. We fixed the number of pretraining epochs to 2500 with a batch size of 32 using the ADAM optimizer. We then trained several models using various number of warm-up steps with both HMC and NUTS. When trained with HMC, we fixed the number of Leapfrog steps to $L = 512$. When trained with NUTS, we set a maximum tree depth of 12 corresponding to a maximum of $L = 4096$ Leapfrog steps. In figure 7.9 we show the achived R^2-scores of the different configurations both in log space and target space as function of the number of warm-up steps used. In log space, the models consistently achieve scores $R^2 \approx 1$ which suggest they on average correctly predict the targets in the test data. Transforming the predictions and targets back to target space, however, lead to som fairly poor results as we increase the number of warm-up steps with HMC outperforming the models trained with NUTS in all but one case. We removed the point at 32 warm-up steps for the NUTS sampler as it achieved a poor score of $R^2 = -1306$ in this case. It is difficult to explain this discrepancy as the R^2-score in log space is good. But the general observation to note here is that the performance of the models degrade once we transform back to target space which is expected as we discussed in section 7.1.2 on the effect of the data transformations. At least we now have an empirical confirmation that suggests that reducing the base of the logarithm used to transformm the data should be investigated further.

In figure 7.10 we display the standardized residuals in log space for both samplers over all trained configurations. The standardized residuals demonstrate that the claims discussed in the beginning of this section may not be general enough to apply to BNNs as the models trained with HMC performs better almost regardless of how many warm-up steps that are performed. When using NUTS, the performance of the model trained with a substantial amount of warm-up steps appear to degrade as opposed to improve consistent with their R^2-scores. The standardized residual of HMC lies consistently inside the Normal distribution for the bulk of the distribution albeit with longer tails, while the NUTS sampler produced rather few models that achieve the same. These results, then, actually indicate that we may be better off running the training procedure of BNNs with a fixed L, only adapting the step size. Even better, we may get by with a fairly small number of warm-up steps and a fairly small L. But the claim that NUTS performs at least as well as HMC with an equivalent number of Leapfrog steps begs the question, then, how many Leapfrog steps did NUTS use on average? In figure 7.11, we show the average number of Leapfrog steps the NUTS sampler used during the generation of the Markov chain as a function of number of warm-up steps. Clearly, NUTS used more Leapfrog steps on average than $L = 512$, except in a single case. The NUTS sampler introduces the need for a slightly more complicated analysis than a one-to-one comparison like this though. Recall from chapter 5, that the NUTS sampler generates samples by running HMC forwards and backwards in time at random until a stopping condition is met. At this point the sampler selects one of the acceptable states that does not violate detailed balance at random. Thus the selected position (model parameter) may lie close to the initial position while HMC automatically selects the final position of the simulation. Thus for the same number of Leapfrog steps, we should expect HMC to generate larger jumps in parameter space when using the same step size.

7.3.3.2 The Effect of Pretraining

Pretraining a model before initiating the Markov chain is typically suggested as a means to accelerate convergence to the stationary distribution by minimizing a selected loss function with respect to the model parameters to obtain a point estimate. The point estimate is then used as the initial point of the Markov chain. This may help but as we discussed in chapter 3, the typical set, the set which we seek to sample from, may not lay particularly close to the mode of the loss function, or as we know it as, the potential energy function. Thus we have good reasons to challenge this recommendation and verify that it indeed improves the performance of the BNN models we sample. We trained a BNN model with the architechture 5-20-20-1 with $\tanh(x)$ as the activation function in the hidden layers. We fixed the number of warm-up steps to 1000 of which 800 were used for adaptation of the step size and 200 were used for burn-in. We fixed the number of Leapfrog steps to $L = 512$ using the HMC sampler. As usual, we sampled 1000 neural networks with 10 steps between each sampled network. From the figure, we can observe that the performance of the model increases as we increase the number of pretraining epochs which gives us empirical grounds for initiating the Markov chain from the point estimate obtained. As in the previous section, we see can not a dramatic decrease in performance when we transform the predictions back to target space and compute the R^2-score.

In figure 7.13, we show the computed standardized residuals in log space. The figure demonstrates a pretty noticable improvement as the amount of pretraining increases, up to a point. Once we surpass 2048 epochs of pretraining, we see a slight degradation of the model performance with a larger spread in the residual distribution. But we can rest assured that pretraining can be used to increase the performance of the trained BNN when everything else is held fixed, and should thus be applied as part of the training procedure. Note that the model used here consists of a rather small number of parameters. We performed pretraining of the models on a GPU which yielded less than 10% GPU utilization, while the MCMC sampling of the model parameters required $\sim 90\%$ GPU utilization. Hence pretraining with models of this size is better performed on either a built-in GPU on an SoC such as the one shipped with an Apple Silicon SoC. Else, the pretraining should be employed on a multi-core CPU device.

7.3.3.3 Effect of Number of Parameters

Increasing the number of parameters of the BNN model may help capture the underlying process from the data to a larger degree. The typical problems posed by the *bias-variance trade-off* [8] (pages 12-13) does not play as significant a role here since the trained model can compute a sample variance along with its prediction. This does not mean it is immune, however, as the model is still sampled according to the nuances found in the training data which in principle may be due to noise. Moreover, the training data may not be particularly representive of the distribution of the underlying process we are trying to learn from the training data, or it may contain outlies which the model gathers samples to explain. As explained in section 7.2.3, the dataset produced by `Prospino` contains very little noise and thus specializing the model to the inherent noise is not the issue. We did, however, see outliers in figure 7.1 and figure 7.2. If a model has a large number of parameters, then, the training may produce a BNN model that generalizing poorly to the unseen test data because it attempts to account for the outliers. Moreover, the dataset we use is fairly small ($\sim$ 16000 datapoints in total), which can exacerbate the effect.

In figure 7.14 we show the computed R^2 score on the training and test data as a function of number of nodes n placed in the hidden layer of models with the architecture 5-n-1. The hidden activation used was $\tanh(x)$. The training was carried out with 1000 warm-up steps with the usual division of 20% burn-in steps and 80% allocated to adapdation of the step size in the Leapfrog integrator. We performed 2500 pretraining steps with a batch size of 32 for each model. When using the HMC sampler, we fixed the number of Leapfrog steps to $L = 512$. For the NUTS sampler, we allowed a maximum of $L = 4096$ Leapfrog steps. We sampled 1000 networks in each cases, skipping 10 networks between each stored sample. The models trained with the HMC sampler increase in performance up until a maximum after which the performance degrades, which may be explained by the fixed number of Leapfrog steps in an increasingly higher-dimensional parameter space. Moreover, the dataset used is fairly small for $7n + 1$ parameters in total as n increases (which in the highest case of $n = 2^13 = 8192$ parameters imply 57345 parameters per sampled neural network).

In figure 7.15, we show the computed standardized residual distribution of the models listed in table 7.1, which gives us an idea of the performance the BNNs have on this dataset as a function of the number of parameters. We can note that model 3 and 4 performs fairly well given our chosen benchmark measure, but that model 5's performance degrade somewhat, which is the model that contains the most parameters of the models tested.

7.3.4 Predictive Distributions

As we discussed in chapter 2, one of the primary objects we seek to compute in Bayesian ML is the predictive distribution $p(y^*|x^*, D)$ for a target y^* given an unseen input point x^* and a training dataset D. In figure 7.16, we show the predictive distribution computed with model 3 in table 7.1. In the figure on top, the sample mean approximates the true target well with a fairly small spread in the distribution which is a desirable outcome in most cases. There are, however, ill cases as well which we demonstrate in the figure at the bottom. Here the true target lies entirely outside the predictive distribution. Thus care must be taken to understand when a BNNs prediction is reliable and when it is not. We can deal with this problem by empirically counting how many targets $y \in [\mu - k\sigma, \mu + k\sigma]$ for $k = 1, 2, 3, 4, 5$ where μ represents the sample mean and σ^2 represents the sample variance of each predictive distribution computed by the BNN model. In principle there is no need for k to be an integer, and a finite grid of points $k \in (0s, \infty)$ can be used instead such that an arbitrary desired accuracy can be specified. We illustrate the results of this analysis in figure 7.17 performed on the training, validation and test data. Clearly there exists a small percentage of ill cases where the target lies far away from the predicted mean. The result does at least tell us that more than 95% of the targets lie within $\mu \pm 3\sigma$ over each dataset in their respective predictive distribution.

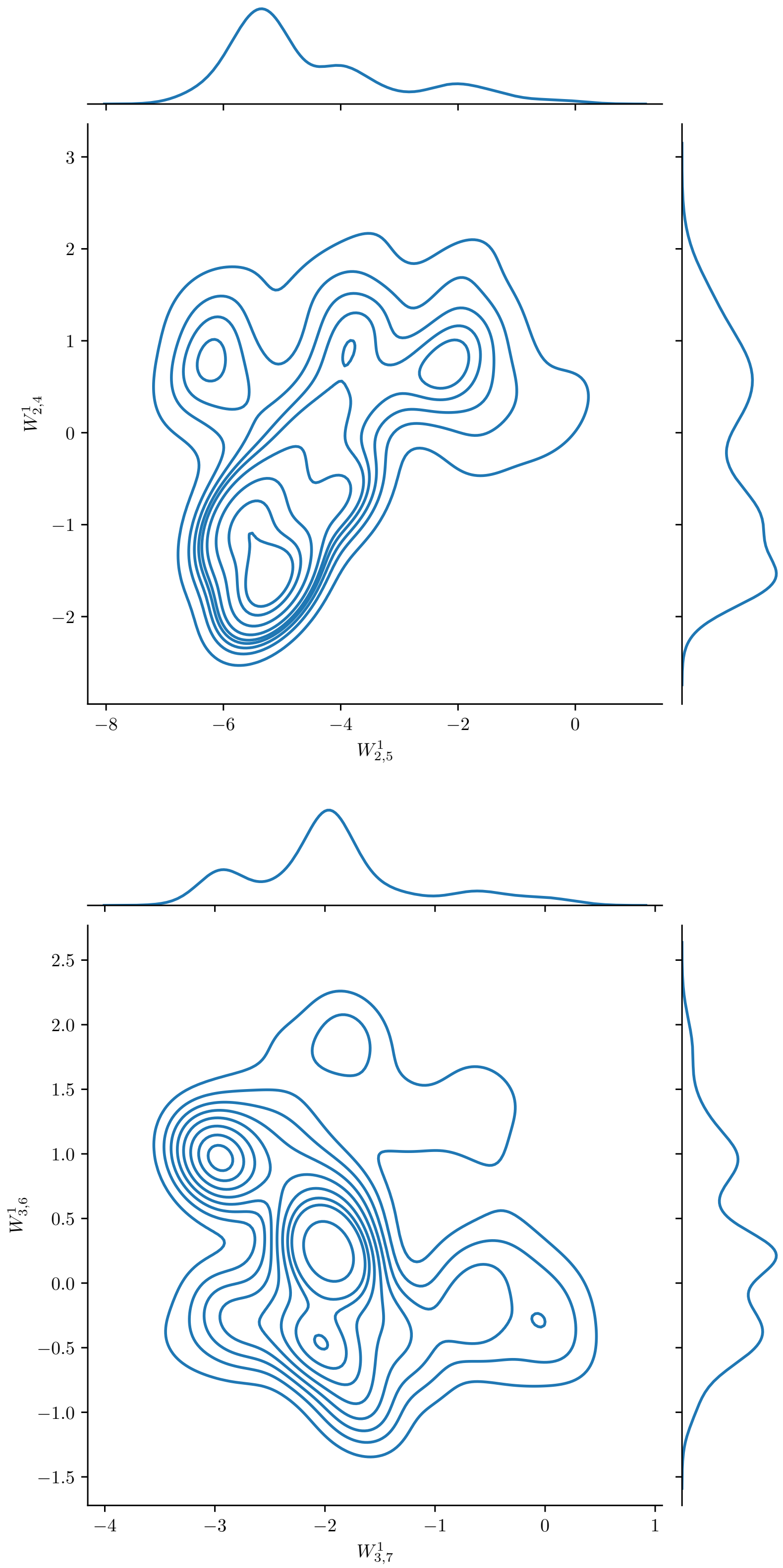
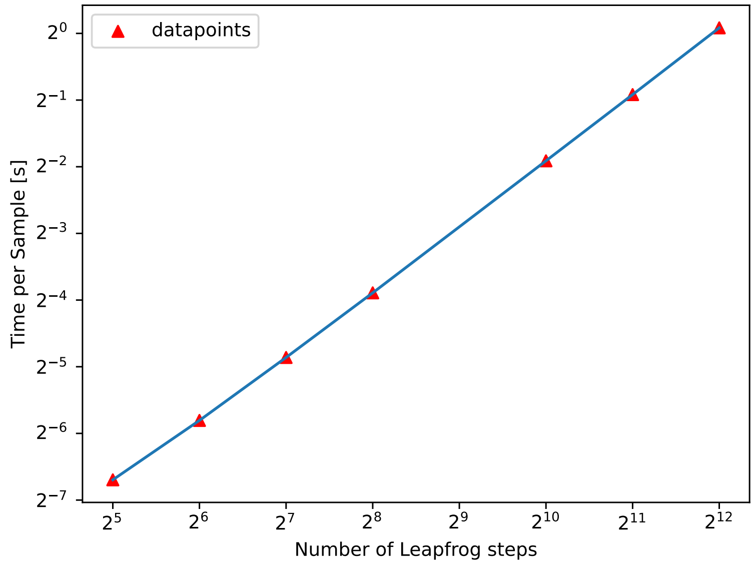


Figure 7.7: The figure shows the projection of the empirical distribution onto the planes spanned by $(W^1_{2,4}, W^1_{2,5})$ on the left and onto the plane spanned by $(W^1_{3,7}, W^1_{3,6})$ on the right, using the samples from model 3 in table 7.1. The distributions are approximated using kernel density estimation.



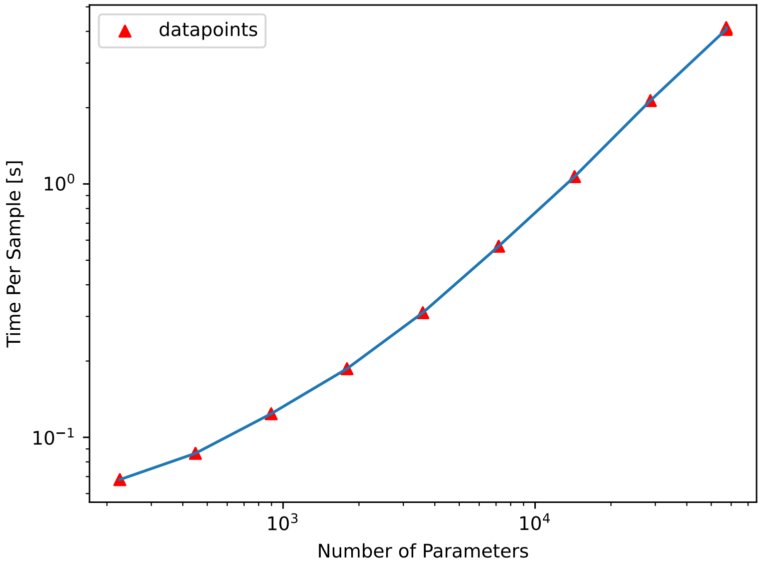


Figure 7.8: The figure at the top shows the measured time in seconds per sample using HMC as a function of Leapfrog steps L using a model with 561 parameters. The figure at the bottom shows the time in seconds per sample with the same sampler with a fixed number of Leapfrog steps $L = 512$ as a function of number of parameters in the BNN model.

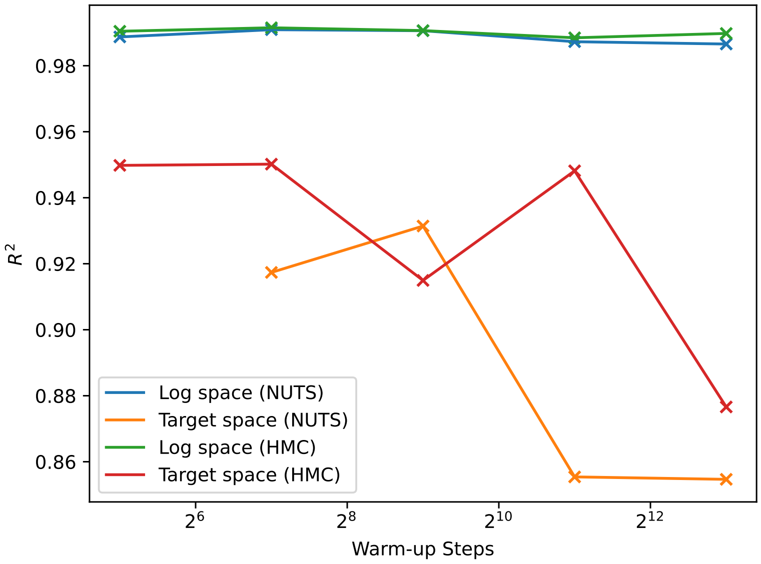
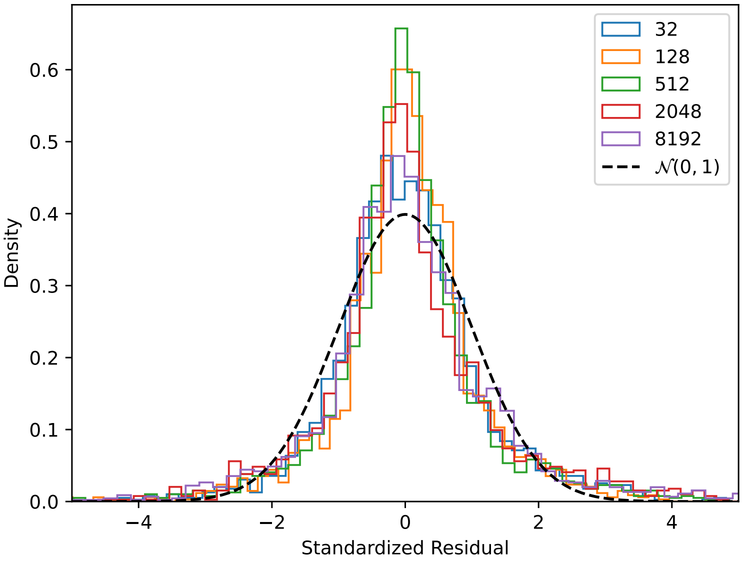


Figure 7.9: The figure shows the computed R^2-scores in both log space and target space as a function number of warm-up steps (20% burn-in and 80% adaptation) achieved with HMC and NUTS. The architecture of the BNN model used is 5-20-20-1 with $\tanh(x)$ used as the activation function in the hidden layers. We performed 2500 pretraning steps with a batch size of 32 using the ADAM optimizer. In total 1000 neural networks were sampled with 10 steps between each stored sample. When HMC was used, we ran with a fixed number of Leapfrog steps $L = 512$. When the NUTS sampler was used, we allowed for a maximum of $L = 4096$ Leapfrog steps (a maximum tree depth of 12).



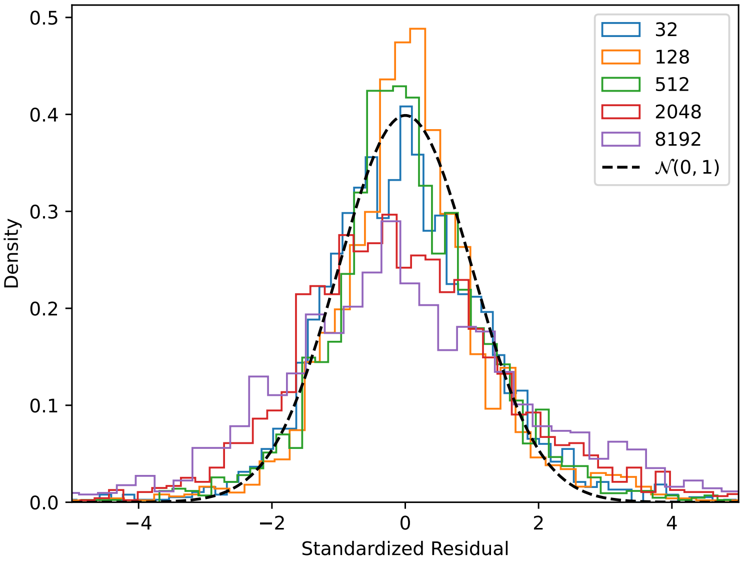


Figure 7.10: The figure shows the standardized residuals computed on the testset. The model architechture used is a model with layers 5-20-20-1 with $\tanh(x)$ as the hidden activation function. In the top figure, we have used the HMC sampler with a fixed number of Leapfrog steps $L = 512$. In the bottom figure, we have used the NUTS sampler with a maximum tree depth of 12 corresponding to a maximum of $L = 2^{12} = 4096$ Leapfrog steps. The remaining important hyperparameters were 2500 pretraining epochs with a batch size of 32 using the ADAM optimizer. In total a 1000 neural networks were sampled in each case with a thinning-amount of 10 steps between each sample.

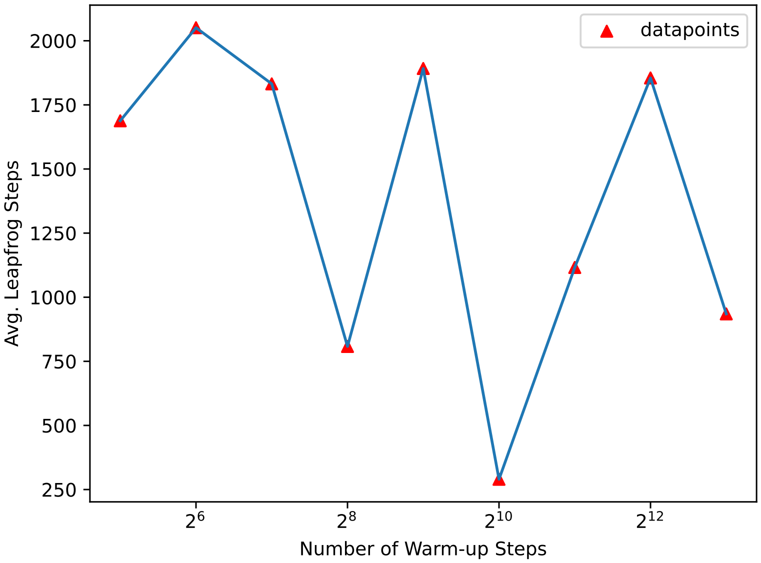


Figure 7.11: The figure shows the average number of Leapfrog steps L as a function of number of warm-up steps used by the NUTS sampler when sampling the models shown in the bottom of figure 7.10. We have included a few more measurements to showcase how fluctuating the average number can be.

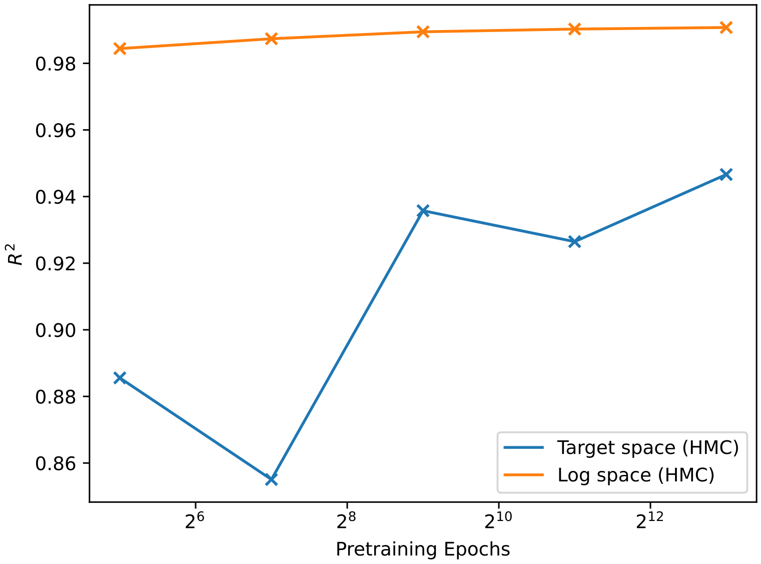


Figure 7.12: The figure shows the computed R^2-scores of a model with the architecture 5-20-20-1 with $\tanh(x)$ as the hidden activation function. In this case the varying number of the number of epochs run with pretraining starting from 32 all the way up to 8192. The batch size used was 32 with the ADAM optimizer. The number of warm-up steps was 1000 (200 of which were burn-in steps and 800 were adaptation steps). We fixed the Leapfrog steps to $L = 512$ using the HMC sampler. As usual we sampled 1000 neural networks with 10 steps between each sample.

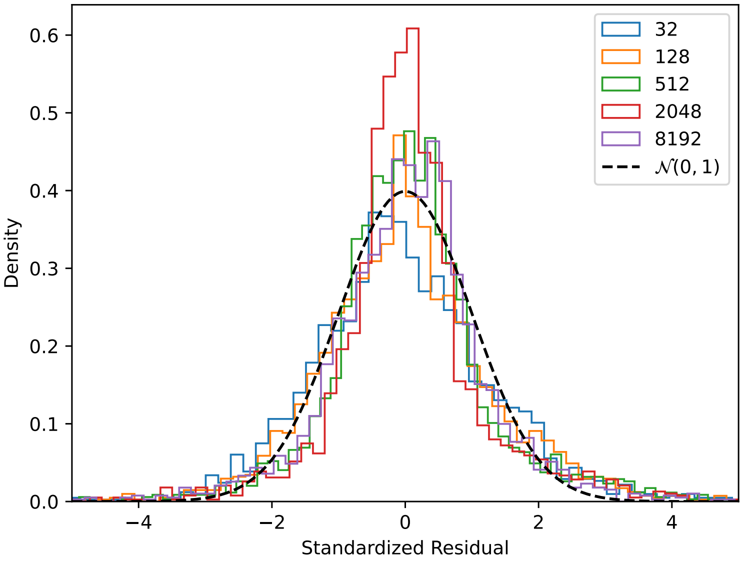


Figure 7.13: The figure shows the standardized residuals of a model with the architecture 5-20-20-1 with $\tanh(x)$ as the hidden activation function. In this case the varying number of the number of epochs run with pretraining starting from 32 all the way up to 8192. The batch size used was 32, the number of warm-up steps was 1000 (200 of which were burn-in steps and 800 were adaptation steps). We fixed the Leapfrog steps to $L = 512$ using the HMC sampler. The ADAM optimizer was used for the pretraining phase. As usual we sampled 1000 neural networks with 10 steps between each sample.

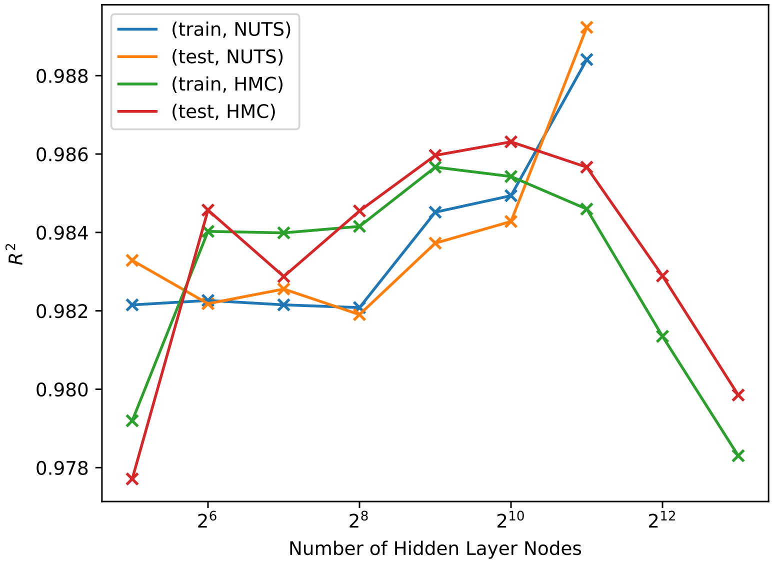


Figure 7.14: The figure shows the R^2-score computed on the training and test data as a function of number of nodes n in the hidden layer of models with architechture 5-n-1, yielding a total of $5n + 1$ parameters. The hidden layer activation used was $\tanh(x)$. The models were trained with 1000 warm-up steps (20% burn-in and 80% adaptation), gathering 1000 neural networks with 10 steps between each sample. We used 2500 pretraining epochs with a batch size of 32. When using the HMC sampler, we fixed the number of Leapfrog steps to $L = 512$. When using NUTS, we set a maximum of $L = 4096$ Leapfrog steps.

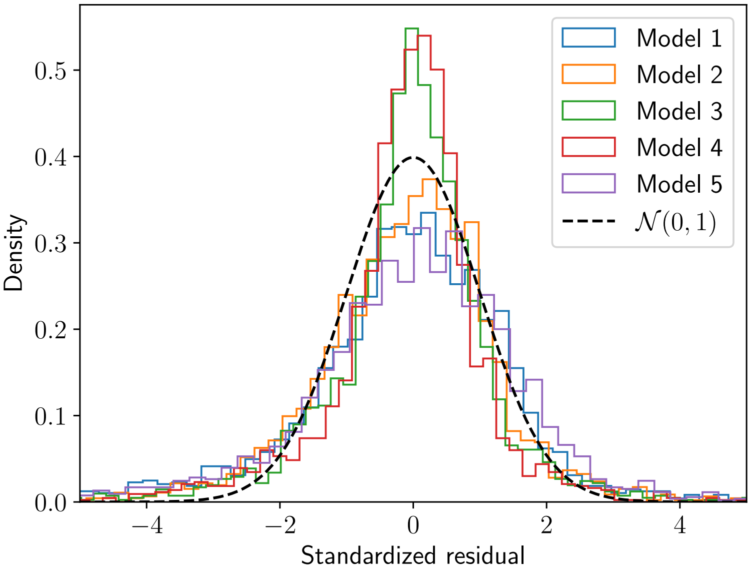


Figure 7.15: The figure shows the distribution of the standardized residuals computed on the test data using the models listed in table 7.1. The Normal distribution is drawn in with a dotted black line for benchmarking reference. The figure is meant to illustrate the performance of the models with respect to the number of parameters in the models. The models were trained with 2500 warm-up steps (20% burn-in and 80% adaptation), gathering 1000 neural networks with 10 steps between each sample. We used 1000 pretraining epochs with a batch size of 32. The kernel used was the NUTS kernel with a maximum of $L = 4096$ Leapfrog steps.

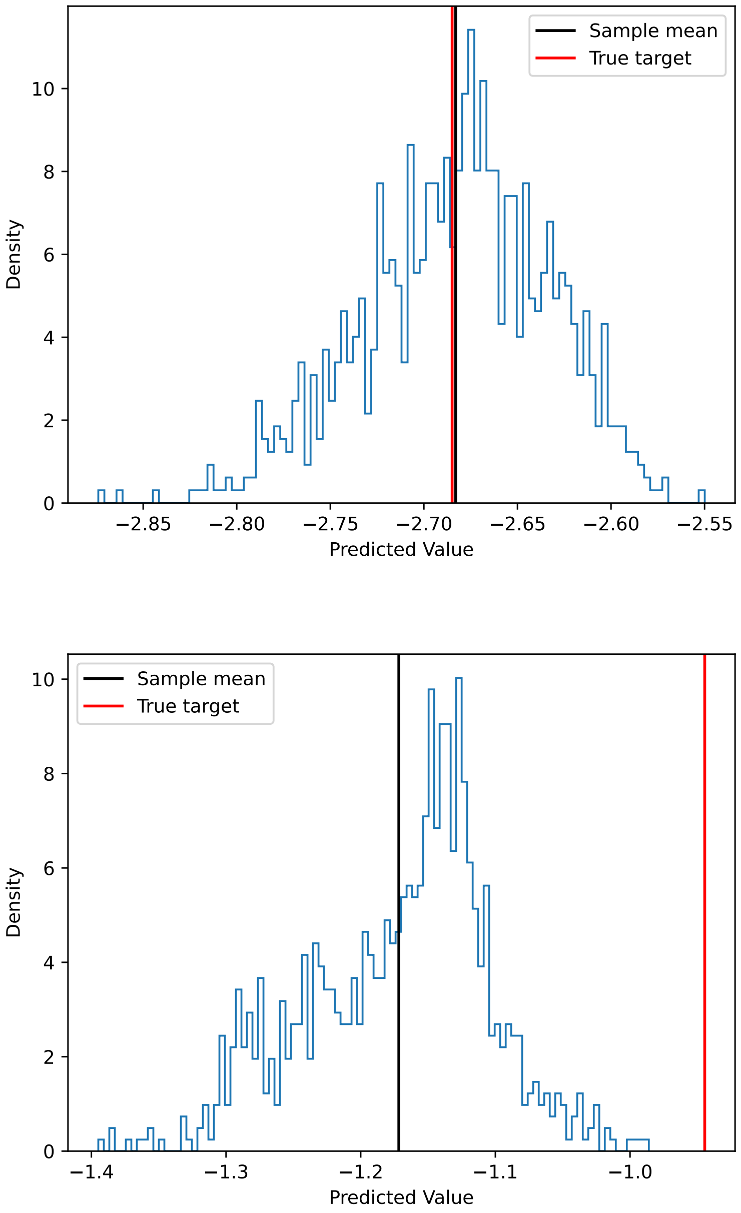


Figure 7.16: The figure shows the predictive distribution estimated by use of model 3 in table 7.1 for to randomly chosen points from the test set. The red line shows the true target and the black line shows the predicted sample mean obtained from the distribution. The figure on top demonstrates a case where the sample mean is approximately the same as the target, while the figure at the bottom demonstrates a case where the true target lies entirely outside the predictive distrbution.

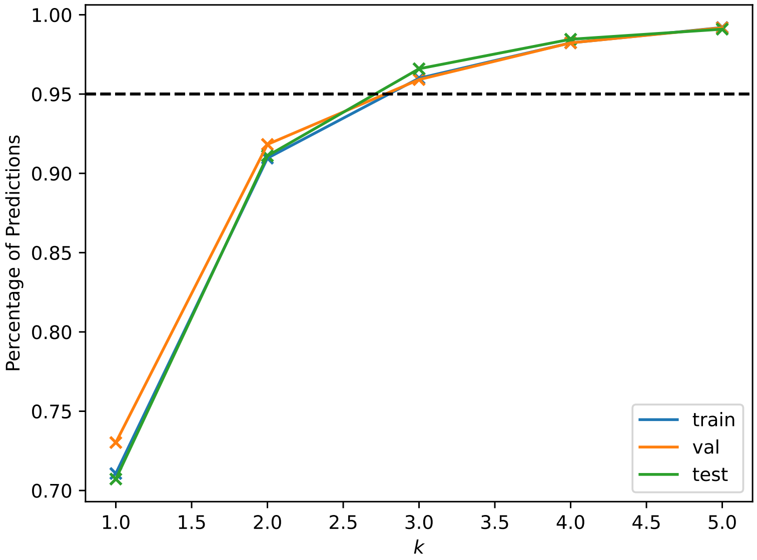


Figure 7.17: The figure shows the predictive distribution estimated by use of model 3 in table 7.1 computed on all datapoints in the training, validation and test data. The black line shows the 95% line. The crosses indicate measured data with training data shown in blue, validation data shown in orange and test data shown in green. The y-axis shows the percentage of all targets lie on the interval $[\mu - k\sigma, \mu + k\sigma]$ for $k = 1, 2, 3, 4, 5$ where μ is the sample mean and σ^2 is the sample variance of the predictive distribution.

Conclusion

Conclusion here.

Appendices

Appendix A

A.1 Appendix 1 title

Some appendix stuff.

Bibliography

[1] M. D. Hoffman and A. Gelman, *The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo*, 2011.

[2] C. Balázs, , A. Buckley, L. A. Dal, B. Farmer, P. Jackson et al., *ColliderBit: a GAMBIT module for the calculation of high-energy collider observables and likelihoods*, *The European Physical Journal C* **77** (nov, 2017) .

[3] W. Beenakker, R. Hoepker and M. Spira, *Prospino: A program for the production of supersymmetric particles in next-to-leading order qcd*, 1996. 10.48550/ARXIV.HEP-PH/9611232.

[4] A. Buckley, A. Kvellestad, A. Raklev, P. Scott, J. V. Sparre, J. V. den Abeele et al., *Xsec: the cross-section evaluation code*, *The European Physical Journal C* **80** (dec, 2020) .

[5] T. Nishijima, *Universal approximation theorem for neural networks*, 2021. 10.48550/ARXIV.2102.10993.

[6] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. 10.48550/ARXIV.1412.6980.

[7] J. L. Devore and K. N. Berk, *Modern Mathematical Statistics with Applications*, p. 80. Springer, 2018.

[8] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher et al., *A high-bias, low-variance introduction to Machine Learning for physicists*, *Physics Reports* **810** (may, 2019) 1–124.

[9] M. Betancourt, *A Conceptual Introduction to Hamiltonian Monte Carlo*, 2017. 10.48550/ARXIV.1701.02434.

[10] G. O. Roberts and J. S. Rosenthal, *General state space markov chains and MCMC algorithms*, *Probability Surveys* **1** (jan, 2004) .

[11] A. Gelman and D. B. Rubin, *Inference from Iterative Simulation Using Multiple Sequences*, *Statistical Science* **7** (1992) 457 – 472.

[12] S. Brooks, A. Gelman, G. L. Jonas and X.-L. Meng, eds., *Handbook of Markov Chain Monte Carlo*, ch. 6. Springer, 2018.

[13] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, *Equation of State Calculations by Fast Computing Machines*, *The Journal of Chemical Physics* **21** (1953) 1087–1092, [`https://doi.org/10.1063/1.1699114`].

[14] W. K. Hastings, *Monte Carlo sampling methods using Markov chains and their applications*, *Biometrika* **57** (04, 1970) 97–109, [`https://academic.oup.com/biomet/article-pdf/57/1/97/23940249/57-1-97.pdf`].

[15] J. S. Helbert Goldstein, Charles Poole, *Classical Mechanics, 3rd ed.*, ch. 2,8. Addison Wesley, 2000.

[16] C. M. Bishop, *Pattern Recognition and Machine Learning*, ch. 11, p. 551. Springer New York, 2006.

[17] Y. Nesterov, *Primal-dual subgradient methods for convex problems*, *Mathematical Programming* **120** (Aug, 2009) 221–259.

[18] J. Park and Y. F. Atchadé, *Markov chain monte carlo algorithms with sequential proposals*, 2019. 10.48550/ARXIV.1907.06544.

[19] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro et al., *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015.

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

[21] P. Ramachandran, B. Zoph and Q. V. Le, *Searching for activation functions*, *CoRR* **abs/1710.05941** (2017) , [`1710.05941`].

[22] D. Chicco, M. J. Warrens and G. Jurman, *The coefficient of determination r-squared is more informative than smape, mae, mape, mse and rmse in regression analysis evaluation*, *PeerJ. Computer science* **7** (Jul, 2021) e623–e623.

[23] A. Sabne, *XLA : Compiling Machine Learning for Peak Performance*, 2020.

[24] P. Izmailov, S. Vikram, M. D. Hoffman and A. G. Wilson, *What Are Bayesian Neural Network Posteriors Really Like?*, *CoRR* **abs/2104.14421** (2021) , [`2104.14421`].