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

Motivation, context and problem.

Outline of the Thesis

Give outline of thesis

Chapter 1

The Physics Problem

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

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

for some true function $f : \mathbb{R}^p \to \mathbb{R}^d$ (also known as the *ground truth*), where $\epsilon \in \mathbb{R}^d$ 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 [1] 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* [2]. 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 [3].

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. A general discussion of such methods are 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 $\{\theta^{(1)},\dots,\theta^{(n)}\}$ 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 category of sample space focused on 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 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 probability density function $\pi(\theta)$ and a 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 expecation of as a *target function*. For all but a few simple low-dimensional densities, computing eq. (3.1) is impossible to evaluate analytically. For high-dimensional spaces, evaluating the expectation over the entire sample space is computationally infeasible. Moreover, it is unlikely that the entire sample space contribute significantly to the expectation. Efficiently evaluating the expectations should thus only require evaluation of the integrand in a subset of the sample space. 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 is 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*.

3.1.1 The Typical Set

MCMC methods are devised to efficiently sample points of 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.

2. The typical set. This refer 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 state θ in a volume $\mathrm{d}\theta$.

3. Low-probability density regions. These are regions far away from any mode of the density.

The first and third region will yield neglible contributions to an expectation. The point to stress here is that the interesting part of sample space to explore is not the high-probability region, and is not the part MCMC methods sample from. Instead, they sample from the region in which $\pi(\theta)\mathrm{d}\theta$ 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 stored from a MCMC chain.

3.1.2 The Target Density and Bayesian Applications

We initially defined $\pi(\theta)$ as a probability density function, implying that the density is integrated to unity. 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 a proportionality relationship $p(\theta|D) \propto p(D|\theta)p(\theta)$. Thus any MCMC methods we are interested cannot require that the $\pi(\theta)$ is normalized to unity. We only require that the density is smooth and that

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

For this reason, we shall refer to $\pi(\theta)$ as the *target density* or *target distribution* interchangably. 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 them 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 imposed to ensure this is expressed as

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

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. 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 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 will 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 are of limited use when restricted to a practical computation in which only a finite chain can be generated. We are thus more interested in the properties of finite Markov chains.

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*. 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 yields 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 obeys eq. (3.3) is not sufficient to efficiently explore the typical set. Geometrically, this can be regions in which the target distribution rapidly changed within the typical set. The pathological regions can be completely ignored by the chain for much of the exploration, leading to poor convergence and thus biased estimators. But 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 is 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 a certainty if the transition kernel satisfies *geometric ergodicity* [4]. 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* $\hat{R}$ [5]. The ideal value is $\hat{R} = 1$. For values far away from this target, it is unlikely that geometric ergodicity is satisfied. Rule-of-thumb is to assume convergence if $\hat{R} < 1.1$ [6].

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 [7, 8]. The algorithm consist of two components, a proposal of a new state and a correction step. 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

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

These steps are summarized in algorithm 1.

Algorithm 1 Metropolis-Hastings

```
procedure METROPOLIS-HASTINGS($\theta$)

    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

end procedure
```

3.3.1 The Proposal Distribution

There are many valid choices for proposal distribution. A common choice is a Gaussian distribution $q(\theta'|\theta) = \mathcal{N}(\theta'|\theta, \Sigma)$, where Σ is the *covariance matrix.* 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 $\delta \sim \mathcal{N}(0, I)$. 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 requires evaluating the target distribution at the initial state and the proposed state.

3.3.2 Random Walk Metropolis

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 any 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 naturally lead to slow exploration of sample space. The slow exploration also leads to a Markov chain where consecutive samples embody a relatively large measure of correlation. The effect is that the effective sample size grows slowly and efficient evaluation of eq. (3.1) becomes difficult. 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 a conditional distribution as follows.

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

We may summarize this as a function in algorithm 2 which given an initial state $\theta^{(t)}$ returns a new state $\theta^{(t+1)}$ sampled according to eq. (3.8).

Algorithm 2 Gibbs sampling

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

Chapter 4

Hamiltonian Monte Carlo

In this chapter, we will study the details of the Hamiltonian Monte Carlo (HMC) method. It is a Markov Chain Monte Carlo (MCMC) sampling technique that merges Gibbs sampling, Hamiltonian dynamics with a final Metropolis-Hastings update. It avoids random walk behavior by performing gradient-informed exploration of the typical set. We will begin with a survey of Hamiltonian dynamics followed by a description of the *Leapfrog* integrator which is used to simulate Hamiltonian systems. Once these are established, we will describe how HMC is constructed.

4.1 Hamiltonian dynamics

Hamiltonian dynamics [9] plays a central part in the HMC algorithm. Let us assume that $q = (q_1, \ldots, q_d)$ are the *generalized coordinates* and $p = (p_1, \ldots, p_d)$ and *generalized momenta* of a physical system. We will further assume that the Hamiltonian can be decomposed as

$$H(q, p) = V(q) + K(p), \tag{4.1}$$

where V denotes the *potential* energy and K denotes the *kinetic* energy of the system. The time evolution of q and p are found solving *Hamilton's* equations,

$$\frac{\mathrm{d}q_i}{\mathrm{d}t} = \frac{\partial H}{\partial p_i}, \qquad \frac{\mathrm{d}p_i}{\mathrm{d}t} = -\frac{\partial H}{\partial q_i}. \tag{4.2}$$

From Hamilton's equations in eq. (4.2), we can easily show that the Hamiltonian is conserved in time t by

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

A numerical integrator must thus conserve this property. The class of numerical solvers that conserve the Hamiltonian is called *symplectic* integrators. The particular kind that HMC uses is called the *Leapfrog* integrator which we will discuss next.

4.1.1 Leapfrog integration

The Leapfrog integrator[10] is used in HMC to integrate eq. (4.2) to generate new proposal states. Assume we approximate the true coordinates and momenta by $(\hat{q}, \hat{p})$. A single Leapfrog integration step can then be written as in algorithm 4. Here ϵ represents the stepsize used in the algorithm.

Algorithm 3 Leapfrog integration (single step)

```
procedure LEAPFROG($q, p, \epsilon$)
    for $i = 1, \ldots, d$ do
        $\hat{p}_i(t + \epsilon/2) = \hat{p}_i(t) - \frac{\epsilon}{2}\, \partial V(\hat{q}(t))/\partial q_i$
        $\hat{q}_i(t + \epsilon) = \hat{q}_i(t) + \frac{\epsilon}{m_i}\hat{p}_i(t + \epsilon/2)$
    end for
    for $i = 1, \ldots, d$ do
        $\hat{p}_i(t + \epsilon) = \hat{p}_i(t + \epsilon/2) - \frac{\epsilon}{2}\, \partial V(\hat{q}(t + \epsilon))/\partial q_i$
    end for
end procedure
```

It is common to set all masses $m_i = 1$, which simplifies Hamilton's equations somewhat and allows us to straight forwardly *vectorize* our leapfrog integrator.

Algorithm 4 Leapfrog integration (single step)

```
procedure LEAPFROG($q, p, \epsilon$)
    1. $\hat{p}(t + \epsilon/2) = \hat{p}(t) - \frac{\epsilon}{2}\nabla_q V(\hat{q}(t))$
    2. $\hat{q}(t + \epsilon) = \hat{q}(t) + \epsilon\hat{p}(t + \epsilon/2)$
    3. $\hat{p}(t + \epsilon) = \hat{p}(t + \epsilon/2) - \frac{\epsilon}{2}\nabla_q V(\hat{q}(t + \epsilon))$
end procedure
```

4.2 HMC

Hamiltonian Monte Carlo is largely developed by Radford Neal [11]. We seek to sample from some target distribution which we express in terms of a canonical distribution over the generalized coordinates

$$\pi(q) = \exp\left(-V(q)\right). \tag{4.4}$$

In other words, we can generically express any potential energy function as the negative logarithm of the target distribution,

$$V(q) = -\log \pi(q). \tag{4.5}$$

To utilize the framework of Hamiltonian dynamics, we introduce *auxilliary* momentum variables p. This allows us to construct a canonical distribution over *phase-space* as

$$\pi(q, p) = \exp\left(-H(q, p)\right). \tag{4.6}$$

Using the form of the Hamiltonian introduced in eq. (4.1), we can recast the canonical distribution over phase-space as

$$\pi(q, p) = \exp\left(-V(q)\right)\exp\left(-K(p)\right). \tag{4.7}$$

The most common choice for the canonincal distribution over momentum space is a Gaussian which corresponds to the kinetic energy function from classical physics

$$K(p) = \sum_{i=1}^{d} \frac{p_i^2}{2m_i}. \tag{4.8}$$

Before we summarize the algorithm in pseudocode, we should look at conceptual description of the algorithm.

1. Given an initial state q, we randomly sample p from a Gaussian distribution.
2. The next step is to choose a direction in phase space, in which we sample $v \sim \text{Uniform}(\{-1, 1\})$.
3. Now we perform L leapfrog steps along the v direction, for a total trajectory length of $v\epsilon L$.
4. At this point, we have reached a proposal state in phase-space (q^*, p^*) which we perform Metropolis correction, that is, we feed the proposal state and the initial state through the Metropolis-Hastings algorithm and accept the new state with a given acceptance probability.

This sums up a single step in HMC which returns a position state. To sample multiple such points, we simply feed the returned position state q back in to the machinery and redo the procedure. The HMC scheme is summarized in algorithm 5.

Algorithm 5 Hamiltonian Monte Carlo

```
procedure HMCSTEP($L, q, p$)
    Sample $v \sim \text{Uniform}(\{-1, 1\})$.
    $(q^*, p^*) \leftarrow (q, p)$                              ▷ Start from initial state.
    $p^* \leftarrow \text{GIBBS}(p^*)$
    for $l = 1, ..., L$ do                                       ▷ $L$ Leapfrog steps.
        $(q^*, p^*) \leftarrow \text{LEAPFROG}(q^*, p^*, v)$
    end for
    $P = \min\left(1, \exp\left[-\left(H(q^*, p^*) - H(q, p)\right)\right]\right)$
    Sample $u \sim U(0, 1)$                                      ▷ Uniform distribution on $(0, 1)$.
    if $P \geq u$ then
        $(q, p) \leftarrow (q^*, p^*)$                           ▷ Accept proposed state.
    else
        $(q, p) \leftarrow (q, p)$                               ▷ Reject proposed state.
    end if
end procedure
```

4.3 The Potential Energy Function in Bayesian ML 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

$$p(\theta|D) \propto \exp\left(-\mathcal{L}(\theta)\right), \tag{4.9}$$

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. (4.4) 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

$$\mathcal{L} = -\log Z - \log p(D|\theta) - \log p(\theta), \tag{4.10}$$

and if we assume all datapoints are i.i.d we can recast it as eq. (2.24), that is

$$\mathcal{L} = -\log Z - \sum_{i=1}^{N} \log p(y^{(i)}|x^{(i)}, \theta) - \log p(\theta), \tag{4.11}$$

4.4 Limitations of HMC

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 a 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 to be worthwhile. Tuning these parameters requires preliminary runs for a given problem.

In the next chapter, we will study 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.

Chapter 5

The No-U-Turn Sampler

HMC is considered a state-of-the-art sampling method, but suffers the need for manual tuning of the number of leapfrog steps L and the step size ϵ. In this chapter, we will study a sampling method called *The No-U-Turn sampler* (NUTS) [12] built upon HMC that dynamically adapts the integration length ϵL. Moreover, we will look at how we can adaptively set the step size to help NUTS reach a stopping criterion faster.

5.1 Preliminary definitions

The NUTS algorithm introduces a collection of new ideas that need proper development before we delve into the algorithm itself. Given generalized coordinates $q = (q_1, ..., q_d)$ and generalized momenta $p = (p_1, ..., p_d)$, we introduce a *slice* variable u with corresponding conditional distributions

$$p(u|q,p) = \text{Uniform}\left(u; [0, \exp\left\{-H(q,p)\right\}]\right), \tag{5.1}$$

and

$$p(q,p|u) = \text{Uniform}\left(q,p; \left\{q',p' \middle| \exp\left\{-H(q,p)\right\} \geq u\right\}\right). \tag{5.2}$$

This effectively allows for definition of the joint distribution

$$p(q,p,u) \propto \mathbb{I}\left[u \in [0, \exp\left\{-H(q,p)\right\}]\right], \tag{5.3}$$

where $\mathbb{I}[\cdot]$ evaluates to 1 if its argument is true and 0 otherwise. Integrating over u yields the marginal distribution over phase-space

$$p(q,p) \propto \int p(q,p,u)\mathrm{d}u = \int_0^{\exp\{-H(q,p)\}} \mathrm{d}u = \exp\left\{-H(q,p)\right\}. \tag{5.4}$$

which is the target distribution used in HMC. We further introduce the set $\mathcal{B}$ which consists of all states (q,p) traced out by the leapfrog integrator. We will get back to the generation of its elements shortly. We introduce another set $\mathcal{C}$ of candidate states which is a subset $\mathcal{C} \subseteq \mathcal{B}$. The set $\mathcal{C}$ will be chosen deterministically from $\mathcal{B}$ such that none of its elements violates detailed balance if used to produce a transition. Generation of the sets $\mathcal{B}$ and $\mathcal{C}$ given the parameters q, p, u and ϵ defines a conditional distribution $p(\mathcal{B}, \mathcal{C}|q,p,u,\epsilon)$ on which the following properties are imposed:

1. All elements of $\mathcal{C}$ are volume perserving. This effectively translates to $p((q,p)|(q,p) \in \mathcal{C}) \propto p(q,p)$.
2. The current state must be included in $\mathcal{C}$, i.e $p\left((q,p) \in \mathcal{C}|q,p,u,\epsilon\right) = 1$.
3. Any state $(q',p') \in \mathcal{C}$, must be in the slice defined by u. Mathmetically, this is expressed as

$$p\left(u \leq \exp\{-H(q,p)\} \middle| (q',p') \in \mathcal{C}\right) = 1.$$

4. If $(q, p) \in \mathcal{C}$ and $(q', p') \in \mathcal{C}$, then for any $\mathcal{B}$ we impose $p(\mathcal{B}, \mathcal{C}|q, p, u, \epsilon) = p(\mathcal{B}, \mathcal{C}|q', p', u, \epsilon)$. Thus any point in $\mathcal{C}$ is equally likely. This can be encapsulated by introduction of the *transition kernel*

$$\frac{1}{|\mathcal{C}|} \sum_{(q,p)\in\mathcal{C}} T(q', p'|q, p, \mathcal{C}) = \frac{\mathbb{I}\left[(q', p') \in \mathcal{C}\right]}{|\mathcal{C}|}, \tag{5.5}$$

which expresses that a proposed point (q', p') is sampled uniformly from $\mathcal{C}$.

5.1.1 Generation of Candidate Points and Stopping Criterion

Conceptually, generation of $\mathcal{B}$ proceeds as follows. First a single Leapfrog step is integrated forwards or backwards in time, where the direction in time is chosen randomly. Then we repeat with two Leapfrog steps. And then we reiterate with four Leapfrog steps. And so on. The algorithm repeats this procedure until some hitherto undefined stopping criterion is reached. This effectively builds a balanced binary tree in which each node is an element in $\mathcal{B}$. The initial tree is a single node which naturally represents the initial state in phase space. Formally, we double the tree by choosing a random direction $v_j \in \text{Uniform}(\{-1, 1\})$ with $v_j = 1$ representing a trajectory forwards in time and $v_j = -1$ representing a trajectory backwards in time where j is the current depth of the tree. The initial tree node corresponds to a depth of $j = 0$. Consider a tree of depth j. NUTS will then consider the $2^j - 1$ balanced subtrees of the height j-tree with $j > 0$. Let q^- and p^- represent the leftmost leaves of one of its subtrees and q^+ and p^+ represent the rightmost leaves of the same subtree. If either

$$(q^+ - q^-) \cdot p^+ < 0 \quad \text{or} \quad (q^+ - q^-) \cdot p^- < 0, \tag{5.6}$$

the tree doubling is terminated. These conditions express that notion that the trajectory starts turning back toward regions that are already visited in phase space. A so-called “U-turn”. Thus the algorithm requires $2^{j+1} - 2$ inner products for a tree of height j (two inner products per subtree it must consider). This is an added computational cost over standard HMC per leapfrog step. The added cost is, however, negligible for sufficiently large datasets and/or complex models because the computational cost will be dominated by the computation of gradients with respect to the model parameters.

The second stopping criterion considered by NUTS terminates the tree doubling if any of the nodes in the tree of height j yields an energy difference larger than for some maximum energy difference E_{max}. More formally, the constraint follows roughly from the introduction of the slice variable u which required that

$$u \leq \exp\left\{-H(q, p)\right\}, \tag{5.7}$$

which results in

$$H(q, p) + \log u \leq 0. \tag{5.8}$$

NUTS loosens this constraint and simply requires that this sum is less than some max energy value. Thus the tree doubling is terminated if

$$H(q, p) + \log u > E_{\text{max}}, \tag{5.9}$$

is satisfied.

5.1.1.1 Choosing candidate points

We now turn to the problem of choosing which points of $\mathcal{B}$ that should be part of $\mathcal{C}$. Recall the the first property we imposed on the distribution $p(\mathcal{B}, \mathcal{C}|q, p, u, \epsilon)$ was that any point in $\mathcal{C}$ must be volume preserving. Luckily, the Leapfrog integrator is volume preserving, so this condition is automatically satisfied. The second condition was simply that the current state must be included in $\mathcal{C}$, so we simply allow the possibly to transition back to the same state to satisfy this condition. The third condition stated that any point $(q, p) \in \mathcal{C}$ must be part of the slice defined by u. Thus, if we exclude any point (q, p) that does not satisfy $u \leq \exp\{-H(q, p)\}$, we fulfill the condition. The fourth condition was $p(\mathcal{B}, \mathcal{C}|q, p, u, \epsilon) = p(\mathcal{B}, \mathcal{C}|q', p', u, \epsilon)$ for any point $(q, p) \in \mathcal{C}$. For any $(q', p') \in \mathcal{B}$, there exists at most one unique sequence $v_0, ..., v_j$ that generates all other states in $\mathcal{B}$ through the doubling process described above. To satisfy the condition, we must exclude any point that from which it is impossible to generate $\mathcal{B}$. Such a point will either satisfy eq. (5.6) or eq. (5.9) before it manages to complete the tree structure, which halts the generation of the necessary points in $\mathcal{B}$. Assume that the doubling procedure stopping because either condition was satisfied during the last iteration. The final points added to $\mathcal{B}$ must be excluded from $\mathcal{C}$ because they will stop the doubling process and thus cannot produce $\mathcal{B}$ if used as a starting point.

5.2 The Naive NUTS Algorithm

The naive NUTS implementation uses recursion to implicitly store proposal points $\mathcal{C}$ whilst building the balanced binary tree. For convenience, the algorithm is split into two pieces; a function called "BUILDTREE" which can be found in algorithm 6, and a procedure called "NaiveNUTS" in algorithm 7 which performs one step of NUTS similar to the one-step procedure of HMC we discussed in algorithm 5, producing a new point q^*. Let us discuss the computational cost of this algorithm. The algorithm demands $2^j - 1$ evaluations of $H(q, p)$ and its gradient. Moreover, an additional set of operations to determine if a stopping criterion is reached, which is of the order $\mathcal{O}(2^j)$. As argued earlier, though, the computational cost is comparable to standard HMC per leapfrog step when the model is sufficiently complex or the dataset large. However, in its current form it requires storage of 2^j positions and momenta, which for complex models or deep binary trees may results in an intractibly large storage requirement. In the next section we shall study a more efficient solution to reduce the memory footprint of the algorithm.

Algorithm 6 BuildTree function

```
function BUILDTREE($q, p, u, v, j, \epsilon$)
    if $j = 0$ then                    ▷ Initial state of the balanced binary tree. Base case.
        $(q', p') \leftarrow$ LEAPFROG$(q, p, v\epsilon)$.
        $\mathcal{C}' \leftarrow \{(q', p')\}$ if $u \leq \exp\{-H(q', p')\}$ else $\mathcal{C} \leftarrow \emptyset$.
        $s' \leftarrow \mathbb{I}[H(q, p) + \log u \leq E_{\max}]$                    ▷ Stopping criterion of eq.(5.9).
        return $q', p', q', p', \mathcal{C}', s'$.
    else                    ▷ Recursion case where $j > 0$. Builds left and right subtrees.
        $q^-, p^-, q^+, p^+, \mathcal{C}', s' \leftarrow$ BUILDTREE$(q, p, u, v, j-1, \epsilon)$
        if $v = 1$ then
            $q^-, p^-, -, -, \mathcal{C}'', s'' \leftarrow$ BUILDTREE$(q^-, p^-, u, v, j-1, \epsilon)$.
        else
            $-, -, q^+, p^+, \mathcal{C}'', s'' \leftarrow$ BUILDTREE$(q^+, p^+, u, v, j-1, \epsilon)$.
        end if
        $s' \leftarrow s's''\mathbb{I}[(q^+ - q^-) \cdot p^- \geq 0]\mathbb{I}[(q^+ - q^-) \cdot p^+ \geq 0]$.                    ▷ Stopping criterion from eq. (5.6)
        $\mathcal{C} \leftarrow \mathcal{C}' \cup \mathcal{C}''$                    ▷ Expand candidate sets
        return $q^-, p^-, q^+, p^+, \mathcal{C}', s'$.
    end if
end function
```

Algorithm 7 Naive NUTS sampler

```
procedure NaiveNUTS($(q, \epsilon)$)
    Sample $u \sim \text{Uniform}\left([0, \exp\{-H(q,p)\}]\right)$.                ▷ Slice variable.
    Initialize $s = 1$, $q^{\pm} = q$, $p^{\pm} = p$, $j = 0$, $\mathcal{C} = \{(q,p)\}$.
    Sample $p \sim \mathcal{N}(0, I)$.                                                ▷ Momenta
    while $s = 1$ do
        Sample $v_j \sim \text{Uniform}(\{-1, 1\})$                      ▷ Choose direction in phase space
        if $v_j = -1$ then
            $q^-, p^-, -, -, \mathcal{C}', s' \leftarrow \text{BuildTree}(q^-, p^-, u, v_j, j, \epsilon)$.
        else
            $-, -, q^+, p^+, \mathcal{C}', s' \leftarrow \text{BuildTree}(q^+, p^+, u, v_j, j, \epsilon)$.
        end if
        if $s' = 1$ then
            $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$         ▷ Expand set of candidate points if stopping criterion is not met.
        end if
        $s \leftarrow s' \mathbb{I}[(q^+ - q^-) \cdot p^- \geq 0] \mathbb{I}[(q^+ - q^-) \cdot p^+ \geq 0]$.      ▷ Stopping criterion of eq. (5.6)
        $j \leftarrow j + 1$                                                  ▷ Increment tree depth.
    end while
    Sample $q^*$ uniformly from $\mathcal{C}$
    return $q^*$.
end procedure
```

5.3 Efficient NUTS

The implementation resulting from algorithm 6 and 7 yields approximately the same computational cost as standard HMC for complex models or large datasets. There are several weaknesses which can be improved upon:

1. The algorithm stores 2^j positions and momentum for a tree of depth j. For sufficiently complex models or deep enough tree depth, this may require a too large memory footprint.

2. The transition kernel used in algorithm 7 produces "short" transitions in parameter space. There exist alternative transition kernels which produces larger transitions in parameter space while obeying detailed balance with respect to a uniform distribution over $\mathcal{C}$.

3. If a stopping criterion is satisfied during the final doubling iteration, the proposed set $\mathcal{C}'$ is still completely built before termination. A more efficient solution is to terminate the creation of the final proposed set by simply terminating immediately when a stopping criterion is reached.

First, consider the first and second weaknesses. We can introduce a kernel

$$T(q', p' | q, p, \mathcal{C}) = \begin{cases} \dfrac{\mathbb{I}[(q', p') \in \mathcal{C}_{\text{new}}]}{|C_{\text{new}}|} & \text{if} \quad |\mathcal{C}_{\text{new}}| > |\mathcal{C}_{\text{old}}|, \\ \\ \dfrac{|\mathcal{C}_{\text{new}}|}{|\mathcal{C}_{\text{old}}|} \dfrac{\mathbb{I}[(q', p') \in \mathcal{C}_{\text{new}}]}{|\mathcal{C}_{\text{new}}|} + \left(1 - \dfrac{|\mathcal{C}_{\text{new}}|}{|\mathcal{C}_{\text{old}}|}\right) \mathbb{I}[(q', p') = (q, p)] & \text{if} \quad |\mathcal{C}_{\text{new}}| \leq |\mathcal{C}_{\text{old}}|, \end{cases} \tag{5.10}$$

where $\mathcal{C}_{\text{new}}$ and $\mathcal{C}_{\text{old}}$ are disjoint subsets of $\mathcal{C}$ such that $\mathcal{C} = \mathcal{C}_{\text{old}} \cup \mathcal{C}_{\text{new}}$. Here $(q, p) \in \mathcal{C}_{\text{old}}$ represents elements already present in $\mathcal{C}$ before the final doubling iteration and $\mathcal{C}_{\text{new}}$ represents the set of elements added to $\mathcal{C}$ during the final doubling iteration. The transition kernel can be interpreted to describe a probability of a transition from a state in $\mathcal{C}_{\text{old}}$ to a randomly chosen state in $\mathcal{C}_{\text{new}}$. The move is accepted with probability $|\mathcal{C}_{\text{new}}|/|\mathcal{C}_{\text{old}}|$. The storage requirement can be reduced to the order $\mathcal{O}(j)$ by observing that

$$p(q, p | \mathcal{C}') = \frac{1}{|\mathcal{C}'|} = \frac{|\mathcal{C}_{\text{subtree}}|}{|\mathcal{C}'|} \frac{1}{|\mathcal{C}_{\text{subtree}}|} = p((q, p) \in \mathcal{C}_{\text{subtree}} | \mathcal{C}') P(q, p | (q, p) \in \mathcal{C}_{\text{subtree}}, \mathcal{C}'). \tag{5.11}$$

5.4 Dual-Averaging Step Size Adaptation

This section will introduce a step size adaptation scheme.

5.5 NUTS with Dual-Averaging Step Size Adaptation

This section will combine the two algorithms to the one used in most runs in this thesis.

Chapter 6

Bayesian Learning for Neural Networks

6.1 Neural Networks

In this chapter, we will introduce the mathematical formalism underpinning neural networks. For convenience, we will adopt the terminology used by Tensorflow[13] to help make the transition from the mathematics to their machine learning framework easier. 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.

6.1.1 Basic Mathematical Structure

A neural network is most generally defined as a non-linear function $f : \mathbb{R}^p \to \mathbb{R}^d$. This non-linear function is 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 through a real-valued vector $a^\ell \in \mathbb{R}^{n_\ell}$, where a^ℓ is colloquially 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 called the *output layer*, and 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 a layer.
- The complete set of neural network parameters $(W, b) = \{(W^\ell, b^\ell)\}_{\ell=1}^L$ 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 chosen 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[14]. The algorithm boils down to four equations defining a recursive algorithm that approximates the gradient with respect to the parameters of the model. Consider $\mathcal{L}$ as the loss function, quanitifying the error between the target and the model output. The first of the four equations quantifes the error in the output layer.

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

The second equation allows us to compute the error at layer ℓ given we know the error at layer $\ell + 1$,

$$\Delta_j^\ell = \left(\sum_k \Delta_k^{\ell+1} W_{kj}^{\ell+1} \right) \sigma'_\ell(z_j^\ell). \tag{6.4}$$

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

$$\frac{\partial \mathcal{L}}{\partial W_{jk}^\ell} = \frac{\partial \mathcal{L}}{\partial z_j^\ell} \frac{\partial z_j^\ell}{\partial W_{jk}^\ell} = \Delta_j^\ell a_k^{\ell-1}. \tag{6.5}$$

For the biases, the gradients are

$$\frac{\partial \mathcal{L}}{\partial b_j^\ell} = \frac{\partial \mathcal{L}}{\partial z_j^\ell} \frac{\partial z_j^\ell}{\partial b_j^\ell} = \Delta_j^\ell. \tag{6.6}$$

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 [3]). 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 can be performed with the backpropagation algorithm.

6.1.3 Loss Function for Regression

In this thesis, we are concerned with regression tasks. The activation function of the final layer σ_L is then just the identity function. The typical loss function chosen to solve regression tasks is the L_2-norm, which for a single output can be written as

$$\mathcal{L}(y, \hat{y}) = \frac{1}{2} \|y - \hat{y}\|_2^2, \tag{6.7}$$

where $\hat{y}$ denotes the model output and y the ground-truth. Now, the model output in this case is $\hat{y}_j = a_j^L = z_j^L$. Therefore,

$$\Delta_j^L = \frac{\partial \mathcal{L}}{\partial z_j^L} = a_j^L - y_j. \tag{6.8}$$

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 gradients of the loss function $\mathcal{L}$ with respect to the model parameters. The forward pass of the neural network is summarized algorithm 8.

Algorithm 8 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 9.

Algorithm 9 Backpropagation: Backward pass

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

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

6.1.4 Regularization in Neural Networks

Neural networks often end up with a large number of parameters, which makes them prone to *overfit* training data. This means that the trained parameters of the model is adjusted to capture trends in the training data which may not represent the underlying process the model tries to learn. The consequence is that it *generalizes* poorly to new unseen data, i.e its predictions are poor. A typical strategy to avoid this problem, is to introduce some form of *regularization*. A common choice is L^2-*regularization*, which for a neural network tacks on two additional sums to the loss function as follows:

$$\mathcal{L} = \frac{1}{2} \sum_i \left\| \hat{y}^{(i)} - y^{(i)} \right\|_2^2 + \frac{\lambda_W}{2} \sum_\ell \left\| W^\ell \right\|_2^2 + \frac{\lambda_b}{2} \sum_\ell \left\| b^\ell \right\|_2^2, \tag{6.9}$$

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. Consider a matrix

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

$$\|A\|_2 = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |A_{ij}|^2}. \tag{6.10}$$

L^2-regularization is sometimes called L^2-penalty because it penalizes assignment of large values to the model parameters. Its effect is thus shrinkage of the parameter space where accessible minima may reside.

6.2 Activation Functions

There are many common activation functions with various strengths used in modern neural networks. We will discuss a few common ones with an emphasis on the ones used in this thesis.

6.2.1 Sigmoid

The sigmoid activation function is given by

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{6.11}$$

It was a very common choice in neural networks early, likely due to its simple derivative. It has a significant drawback, however. Looking at eq. (6.11), we can easily deduce that $\sigma(\pm\infty) = 0$, and since its derivative is of the form $\sigma'(x) = \sigma(1 - \sigma)$, the gradient computed during backpropagation vanishes if the input to the activation function as $|x| \to \infty$. This significantly hampers the progress during optimization. A popular alternative to the sigmoid function is $\tanh(x)$. This function too, however, suffers from the same vanishing gradient problem.

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 [15] 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

$$\sigma(x) = x \cdot \mathrm{sigmoid}(x). \tag{6.13}$$

6.3 Bayesian Framework for Neural Networks

We can specialize the equations used in Bayesian inference for neural networks in the context of regression. The predictive distribution $P(y|x, \theta)$ seeks to model a function $f : \mathbb{R}^p \to \mathbb{R}^d$ that for a given $x \in \mathbb{R}^p$ produces an output $y \in \mathbb{R}^d$. In the infinite data limit, the distribution should be a Dirac delta function. For finite datasets, however, we instead seek a distribution of outputs given the input features.

Consider a set of observations $D = \{(x^{(1)}, y^{(1)}), ..., (x^{(n)}, y^{(n)})\}$, where $x^{(i)} \in \mathbb{R}^p$ are the input features and $y^{(i)} \in \mathbb{R}^d$ are the targets. The equation for the predictive distribution of y^* given an input x^* changes to

$$P(y^*|x^*, D) = \int P(y^*|x^*, \theta) P(\theta|D) \mathrm{d}\theta. \tag{6.14}$$

Assuming that the the observations in D are drawn independently, the likelihood function can be expressed as

$$P(D|\theta) = \prod_{i=1}^{n} P(y^{(i)}|x^{(i)}, \theta). \tag{6.15}$$

In the context of regression, the likelihood for a given observation (x, y) is commonly chosen to be

$$P(y|x, \theta) \propto \exp\left(-\frac{\lambda}{2}\|y - f(x;\theta)\|_2^2\right). \tag{6.16}$$

where $f(x;\theta)$ is the output of the neural network and λ is a hyperparameter typically chosen to be identical for all inputs (x, y) during training. The likelihood found eq. (6.16) is equivalent to using L^2-norm as a loss function with regularization strength λ when framed as a minimization problem, which we will see shortly.

In practice, however, we instead sample a set of network parameters $\{\theta_1, ..., \theta_n\}$ from the posterior distribution

$$P(\theta|D) \propto P(D|\theta)P(\theta), \tag{6.17}$$

from which we can produce a set of predictions $\{\hat{y}_1, \ldots \hat{y}_n\}$ from the neural network model

$$\hat{y}_i = f(x, \theta_i). \tag{6.18}$$

Given this set of predictions, we can compute the sample mean

$$\hat{y}_{\text{MLE}} = \frac{1}{n}\sum_i f(x, \theta_i), \tag{6.19}$$

which is an approximation to the *maximum likelihood estimate* (MLE). Furthermore, an estimate of the error is provided by the sample *covariance*

$$\text{Cov}(\hat{y}) = \frac{1}{n-1}\sum_i (\hat{y}^{(i)} - \hat{y}_{\text{MLE}})(\hat{y}^{(i)} - \hat{y}_{\text{MLE}})^T. \tag{6.20}$$

The diagonal terms of eq. (6.20) yields the sample *variance* of the components of $\{\hat{y}^{(i)}\}_{i=1}^n$.

6.4 Bayesian learning using HMC

To learn from the data D using HMC, we need to define a potential energy function $V(q)$ and a kinetic energy function $K(p)$. In the case of a neural network, the potential energy can be specified in the generic form

$$V(W, b) = -\log Z - \sum_\ell \log P(W^\ell) - \sum_\ell \log P(b^\ell) - \sum_i \log P(y^{(i)}|x^{(i)}, W, b). \tag{6.21}$$

where W and b collectively denotes all the kernels and biases of the model, respectively. We thus need to specify a prior for kernels W^ℓ and the biases b^ℓ, as well as a likelihood given an input (x, y). The priors are typically chosen to be Gaussian,

$$P(W^\ell) \propto \exp\left(-\frac{\lambda_W}{2}\|W^\ell\|_2^2\right), \qquad P(b^\ell) \propto \exp\left(-\frac{\lambda_b}{2}\|b^\ell\|_2^2\right). \tag{6.22}$$

The likelihood for regressions tasks was defined in eq. (6.16). Combining this and eq. (6.22), we can write down an explicit expression for the potential energy as

$$V(W, b) = -\log Z + \frac{\lambda_W}{2}\sum_\ell \|W^\ell\|_2^2 + \frac{\lambda_b}{2}\sum_\ell \|b^\ell\|_2^2 + \frac{\lambda}{2}\sum_i \left\|y^{(i)} - f(x^{(i)}; W, b)\right\|_2^2, \tag{6.23}$$

given a set of datapoints $D = \{x^{(i)}, y^{(i)}\}_{i=1}^n$. Here Z denotes the normalization constant of the distribution $P \propto \exp(-V(W, b))$, but is of no importance for the sampling procedure because it either vanishes when the gradient of V is computed, or when we compute the energy difference between the initial state (W, b) and the proposed state (W^*, b^*). Finally, we need to specify a distribution for the kinetic energy. Since we interpret the Hamiltonian as the negative log likelihood of the target distribution, we have implicitly defined the distribution as Gaussian

$$P(p) \propto \prod_{\ell} \exp\left(-\frac{(p^\ell)^T (M^\ell)^{-1} p^\ell}{2}\right), \tag{6.24}$$

where M^ℓ is the diagonal mass matrix of layer ℓ with its mass elements corresponding to each individual momentum and p^ℓ denotes the generalized momenta of layers ℓ.

With the ingredients introduced hitherto, we can proceed to sample from a network with an arbitrary network architecture. The procedure is a follows.

1. Initialize the weights of the network sampled from the priors.
2. Train the network using the backpropagation algorithm to reach an initial network state in the high probability region of the posterior. This is done to reduce the number of burn-in steps needed for the the Markov chain to reach the stationary distribution.
3. Perform a small number of burn-in steps. This step may be redundant as the network is minimized using eq. (6.23), which will place us in a high probability region once the loss is minimized.
4. Sample a set of neural network parameters from the posterior distribution.

Chapter 7

Numerical Experiments

7.1 The Dataset

7.1.1 Data Generation

7.1.2 Data Scaling and Transformations

7.2 Performance Metrics

7.3 Results

7.3.1 Benchmarks of Hyperparameters

7.3.1.1 Baseline Model

7.3.1.2 Pretraining

7.3.1.3 Burn-in length

7.3.1.4 Number of model parameters

7.3.2 Neutralino-Neutralino Cross Sections

Chapter 8

Discussion

Conclusion

Conclusion here.

Appendices

Appendix A

A.1 Appendix 1 title

Some appendix stuff.

Bibliography

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

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

[3] 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.

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

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

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

[7] 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`].

[8] 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`].

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

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

[11] R. Neal, *Handbook of Markov Chain Monte Carlo*, ch. 5. Chapman and Hall/CRC, May, 2011. `http://dx.doi.org/10.1201/b10905`. 10.1201/b10905.

[12] M. D. Hoffman and A. Gelman, *The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo*, 2011.

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

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

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